

# Residuated Transition Systems

Eugene W. Stark

Department of Computer Science  
Stony Brook University  
Stony Brook, New York 11794 USA

February 6, 2026

## Abstract

A *residuated transition system* (RTS) is a transition system that is equipped with a certain partial binary operation, called *residuation*, on transitions. Using the residuation operation, one can express nuances, such as a distinction between nondeterministic and concurrent choice, as well as partial commutativity relationships between transitions, which are not captured by ordinary transition systems. A version of residuated transition systems was introduced by the author in [10], where they were called “concurrent transition systems” in view of the original motivation for their definition from the study of concurrency. In the first part of the present article, we give a formal development that generalizes and subsumes the original presentation. We give an axiomatic definition of residuated transition systems that assumes only a single partial binary operation as given structure. From the axioms, we derive notions of “arrow” (transition), “source”, “target”, “identity”, as well as “composition” and “join” of transitions; thereby recovering structure that in the previous work was assumed as given. We formalize and generalize the result, that residuation extends from transitions to transition paths, and we systematically develop the properties of this extension. A significant generalization made in the present work is the identification of a general notion of congruence on RTS’s, along with an associated quotient construction.

In the second part of this article, we use the RTS framework to formalize several results in the theory of reduction in Church’s  $\lambda$ -calculus. Using a de Bruijn indexed syntax in which terms represent parallel reduction steps, we define residuation on terms and show that it satisfies the axioms for an RTS. An application of the results on paths from the first part of the article allows us to prove the classical Church-Rosser Theorem with little additional effort. We then use residuation to define the notion of “development” and we prove the Finite Developments Theorem, that every development is finite, formalizing and adapting to de Bruijn indices a proof by de Vrijer. We also use residuation to define the notion of a “standard reduction path”, and we prove the Standardization Theorem: that every reduction path is congruent to a standard one. As a corollary of the Standardization Theorem, we obtain the Leftmost Reduction Theorem: that leftmost reduction is a normalizing strategy.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Residuated Transition Systems</b>	<b>9</b>
2.1	Basic Definitions and Properties	9
2.1.1	Partial Magmas	9
2.1.2	Residuation	10
2.1.3	Residuated Transition System	12
2.1.4	Weakly Extensional RTS	23
2.1.5	Extensional RTS	27
2.1.6	Composites of Transitions	27
2.1.7	Joins of Transitions	31
2.1.8	Joins and Composites in a Weakly Extensional RTS	34
2.1.9	Joins and Composites in an Extensional RTS	35
2.1.10	Confluence	49
2.2	Simulations	49
2.2.1	Identity Simulation	51
2.2.2	Composite of Simulations	51
2.2.3	Simulations into a Weakly Extensional RTS	52
2.2.4	Simulations into an Extensional RTS	52
2.2.5	Simulations between Weakly Extensional RTS's	53
2.2.6	Simulations between Extensional RTS's	53
2.2.7	Transformations	54
2.3	Normal Sub-RTS's and Congruence	55
2.3.1	Normal Sub-RTS's	56
2.3.2	Semi-Congruence	57
2.3.3	Congruence	60
2.3.4	Congruence Classes	66
2.3.5	Coherent Normal Sub-RTS's	68
2.3.6	Quotient by Coherent Normal Sub-RTS	75
2.3.7	Identities form a Coherent Normal Sub-RTS	92
2.4	Paths	93
2.4.1	Residuation on Paths	97
2.4.2	Inclusion Map	134

2.4.3	Composites of Paths . . . . .	134
2.4.4	Paths in a Weakly Extensional RTS . . . . .	143
2.4.5	Paths in a Confluent RTS . . . . .	146
2.4.6	Simulations Lift to Paths . . . . .	148
2.4.7	Normal Sub-RTS's Lift to Paths . . . . .	149
2.5	Composite Completion . . . . .	167
2.5.1	Inclusion Map . . . . .	174
2.5.2	Composite Completion of a Weakly Extensional RTS . . . . .	176
2.5.3	Composite Completion of an Extensional RTS . . . . .	178
2.5.4	Freeness of Composite Completion . . . . .	178
2.6	Constructions on RTS's . . . . .	195
2.6.1	Products of RTS's . . . . .	195
2.6.2	Sub-RTS's . . . . .	202
<b>3</b>	<b>The Lambda Calculus</b> . . . . .	<b>211</b>
3.1	Syntax . . . . .	212
3.1.1	Some Orderings for Induction . . . . .	213
3.1.2	Arrows and Identities . . . . .	214
3.1.3	Raising Indices . . . . .	215
3.1.4	Substitution . . . . .	218
3.2	Lambda-Calculus as an RTS . . . . .	221
3.2.1	Residuation . . . . .	221
3.2.2	Source and Target . . . . .	223
3.2.3	Residuation and Substitution . . . . .	229
3.2.4	Residuation Determines an RTS . . . . .	231
3.2.5	Simulations from Syntactic Constructors . . . . .	242
3.2.6	Reduction and Conversion . . . . .	245
3.2.7	The Church-Rosser Theorem . . . . .	247
3.2.8	Normalization . . . . .	250
3.3	Reduction Paths . . . . .	251
3.3.1	Sources and Targets . . . . .	251
3.3.2	Mapping Constructors over Paths . . . . .	254
3.3.3	Decomposition of 'App Paths' . . . . .	259
3.3.4	Miscellaneous . . . . .	269
3.4	Developments . . . . .	271
3.4.1	Finiteness of Developments . . . . .	277
3.4.2	Complete Developments . . . . .	293
3.5	Reduction Strategies . . . . .	296
3.5.1	Parallel Reduction . . . . .	298
3.5.2	Head Reduction . . . . .	304
3.5.3	Leftmost Reduction . . . . .	316
3.6	Standard Reductions . . . . .	322
3.6.1	Standard Reduction Paths . . . . .	322
3.6.2	Standard Developments . . . . .	339

3.6.3 Standardization . . . . .	349
<b>Bibliography</b>	<b>428</b>

# Chapter 1

## Introduction

A *transition system* is a graph used to represent the dynamics of a computational process. It consists simply of nodes, called *states*, and edges, called *transitions*. Paths through a transition system correspond to possible computations. A *residuated transition system* is a transition system that is equipped with a partial binary operation, called *residuation*, on transitions, subject to certain axioms. Among other things, these axioms imply that if residuation is defined for transitions  $t$  and  $u$ , then  $t$  and  $u$  must be *coinitial*; that is, they must have a common source state. If the residuation is defined for coinitial transitions  $t$  and  $u$ , then we regard transitions  $t$  and  $u$  as *consistent*, otherwise they are *in conflict*. The residuation  $t \setminus u$  of  $t$  along  $u$  can be thought of as what remains of transition  $t$  after the portion that it has in common with  $u$  has been cancelled.

A version of residuated transition systems was introduced in [10], where I called them “concurrent transition systems”, because my motivation for the definition was to be able to have a way of representing information about concurrency and nondeterministic choice. Indeed, transitions that are in conflict can be thought of as representing a nondeterministic choice between steps that cannot occur in a single computation, whereas consistent transitions represent steps that can so occur and are therefore in some sense concurrent with each other. Whereas performing a product construction on ordinary transition system results in a transition system that records no information about commutativity of concurrent steps, with residuated transition systems the residuation operation makes it possible to represent such information.

In [10], concurrent transition systems were defined in terms of graphs, consisting of states, transitions, and a pair of functions that assign to each transition a *source* (or domain) state and a *target* (or codomain) state. In addition, the presence of transitions that are *identities* for the residuation was assumed. Identity transitions had the same source and target state, and they could be thought of as representing empty computational steps. The key axiom for concurrent transition systems is the “cube axiom”, which is a parallel moves property stating that the same result is achieved when transporting a transition by residuation along the two paths from the base to the apex of a “commuting diamond”. Using the residuation operation and the associated cube axiom, it becomes possible to define notions of “join” and “composition” of transitions. The residuation also

induces a notion of congruence of transitions; namely, transitions  $t$  and  $u$  are congruent whenever they are cointial and both  $t \setminus u$  and  $u \setminus t$  are identities. In [10], the basic definition of concurrent transition system included an axiom, called “extensionality”, which states that the congruence relation is trivial (*i.e.* coincides with equality). An advantage of the extensionality axiom is that, in its presence, joins and composites of transitions are uniquely defined when they exist. It was shown in [10] that a concurrent transition system could always be quotiented by congruence to achieve extensionality.

A focus of the basic theory developed in [10] was to show that the residuation operation  $\setminus$  on individual transitions extended in a natural way to a residuation operation  $\setminus^*$  on paths, so that a concurrent transition system could be completed to one having a composite for each “composable” pair of transitions. The construction involved quotienting by the congruence on paths obtained by declaring paths  $T$  and  $U$  to be congruent if they are cointial and both  $T \setminus^* U$  and  $U \setminus^* T$  are paths consisting only of identities. Besides collapsing paths of identities, this congruence reflects permutation relations induced by the residuation. In particular, if  $t$  and  $u$  are consistent, then the paths  $t(u \setminus t)$  and  $u(t \setminus u)$  are congruent.

Imposing the extensionality requirement as part of the basic definition of concurrent transition systems does not end up being particularly desirable, since natural examples of situations where there is a residuation on transitions (such as on reductions in the  $\lambda$ -calculus) often do not naturally satisfy the extensionality condition and can only be made to do so if a quotient construction is applied. Also, the treatment of identity transitions and quotienting in [10] was not entirely satisfactory. The definition of “strong congruence” given there was somewhat awkward and basically existed to capture the specific congruence that was induced on paths by the underlying residuation. It was clear that a more general quotient construction ought to be possible than the one used in [10], but it was not clear what the right general definition ought to be.

In the present article we revisit the notion of transition systems equipped with a residuation operation, with the idea of developing a more general theory that does not require the assumption of extensionality as part of the basic axioms, and of clarifying the general notion of congruence that applies to such structures. We use the term “residuated transition systems” to refer to the more general structures defined here, as the name is perhaps more suggestive of what the theory is about and it does not seem to limit the interpretation of the residuation operation only to settings that have something to do with concurrency.

Rather than starting out by assuming source, target, and identities as basic structure, here we develop residuated transition systems purely as a theory about a partial binary operation (residuation) that is subject to certain axioms. The axioms will allow us to introduce sources, targets, and identities as defined notions, and we will be able to recover the properties of this additional structure that in [10] were taken as axiomatic. This idea of defining residuated transition systems purely in terms of a partial binary operation of residuation is similar to the approach taken in [11], where we formalized categories purely in terms of a partial binary operation of composition.

This article comprises two parts. In the first part, we give the definition of residuated transition systems and systematically develop the basic theory. We show how sources,

composites, and identities can be defined in terms of the residuation operation. We also show how residuation can be used to define the notions of join and composite of transitions, as well as the simple notion of congruence that relates transitions  $t$  and  $u$  whenever both  $t \setminus u$  and  $u \setminus t$  are identities. We then present a much more general notion of congruence, based a definition of “coherent normal sub-RTS”, which abstracts the properties enjoyed by the sub-RTS of identity transitions. After defining this general notion of congruence, we show that it admits a quotient construction, which yields a quotient RTS having the extensionality property. After studying congruences and quotients, we consider paths in an RTS, represented as nonempty lists of transitions whose sources and targets match up in the expected “domino fashion”. We show that the residuation operation of an RTS lifts to a residuation on its paths, yielding an “RTS of paths” in which composites of paths are given by list concatenation. The collection of paths that consist entirely of identity transitions is then shown to form a coherent normal sub-RTS of the RTS of paths. The associated congruence on paths can be seen as “permutation congruence”: the least congruence respecting composition that relates the two-element lists  $[t, t \setminus u]$  and  $[u, u \setminus t]$  whenever  $t$  and  $u$  are consistent, and that relates  $[t, b]$  and  $[t]$  whenever  $b$  is an identity transition that is a target of  $t$ . Quotienting by the associated congruence results in a free “composite completion” of the original RTS. The composite completion has a composite for each pair of “composable” transitions, and it will in general exhibit nontrivial equations between composites, as a result of the congruence induced on paths by the underlying residuation. In summary, the first part of this article can be seen as a significant generalization and more satisfactory development of the results originally presented in [10].

The second part of this article applies the formal framework developed in the first part to prove various results about reduction in Church’s  $\lambda$ -calculus. Although many of these results have had machine-checked proofs given by other authors (*e.g.* the basic formalization of residuation in the  $\lambda$ -calculus given by Huet [7]), the presentation here develops a number of such results in a single formal framework: that of residuated transition systems. For the presentation of the  $\lambda$ -calculus given here we employ (as was also done in [7]) the device of de Bruijn indices [4], in order to avoid having to treat the issue of  $\alpha$ -convertibility. The terms in our syntax represent reductions in which multiple redexes are contracted in parallel; this is done to deal with the well-known fact that contractions of single redexes are not preserved by residuation, in general. We treat only  $\beta$ -reduction here; leaving the extension to the  $\beta\eta$ -calculus for future work. We define residuation on terms essentially as is done in [7] and we develop a similar series of lemmas concerning residuation, substitution, and de Bruijn indices, culminating in Lévy’s “Cube Lemma” [8], which is the key property needed to show that a residuated transition system is obtained. In this residuated transition system, the identities correspond to the usual  $\lambda$ -terms, and transitions correspond to parallel reductions, represented by  $\lambda$ -terms with “marked redexes”. The source of a transition is obtained by erasing the markings on the redexes; the target is obtained by contracting all the marked redexes.

Once having obtained an RTS whose transitions represent parallel reductions, we exploit the general results proved in the first part of this article to extend the residuation to sequences of reductions. It is then possible to prove the Church-Rosser Theorem

with very little additional effort. After that, we turn our attention to the notion of a “development”, which is a reduction sequence in which the only redexes contracted are those that are residuals of redexes in some originally marked set. We give a formal proof of the Finite Developments Theorem ([9, 6]), which states that all developments are finite. The proof here follows the one by de Vrijer [5], with the difference that here we are using de Bruijn indices, whereas de Vrijer used a classical  $\lambda$ -calculus syntax. The modifications of de Vrijer’s proof required for de Bruijn indices were not entirely straightforward to find. We then proceed to define the notion of “standard reduction path”, which is a reduction sequence that in some sense contracts redexes in a left-to-right fashion, perhaps with some jumps. We give a formal proof of the Standardization Theorem ([3]), stated in the strong form which asserts that every reduction is permutation congruent to a standard reduction. The proof presented here proceeds by stating and proving correct the definition of a recursive function that transforms a given path of parallel reductions into a standard reduction path, using a technique roughly analogous to insertion sort. Finally, as a corollary of the Standardization Theorem, we prove the Leftmost Reduction Theorem, which is the well-known result that the leftmost (or normal-order) reduction strategy is normalizing.

## Chapter 2

# Residuated Transition Systems

```
theory ResiduatedTransitionSystem
imports Main HOL-Library.FuncSet
begin
```

### 2.1 Basic Definitions and Properties

#### 2.1.1 Partial Magmas

A *partial magma* consists simply of a partial binary operation. We represent the partiality by assuming the existence of a unique value *null* that behaves as a zero for the operation.

```
locale partial-magma =
fixes OP :: 'a ⇒ 'a ⇒ 'a
assumes ex-un-null: ∃!n. ∀t. OP n t = n ∧ OP t n = n
begin
```

```
definition null :: 'a
where null = (THE n. ∀t. OP n t = n ∧ OP t n = n)
```

```
lemma null-eqI:
assumes ⋀t. OP n t = n ∧ OP t n = n
shows n = null
using assms null-def ex-un-null the1-equality [of λn. ∀t. OP n t = n ∧ OP t n = n]
by auto
```

```
lemma null-is-zero [simp]:
shows OP null t = null and OP t null = null
using null-def ex-un-null theI' [of λn. ∀t. OP n t = n ∧ OP t n = n]
by auto
```

```
end
```

## 2.1.2 Residuation

A *residuation* is a partial magma subject to three axioms. The first, *con-sym-ax*, states that the domain of a residuation is symmetric. The second, *con-imp-arr-resid*, constrains the results of residuation either to be *null*, which indicates inconsistency, or something that is self-consistent, which we will define below to be an “arrow”. The “cube axiom”, *cube-ax*, states that if  $v$  can be transported by residuation around one side of the “commuting square” formed by  $t$  and  $u \setminus t$ , then it can also be transported around the other side, formed by  $u$  and  $t \setminus u$ , with the same result.

**type-synonym**  $'a \text{ resid} = 'a \Rightarrow 'a \Rightarrow 'a$

**locale** *residuation* = *partial-magma resid*

**for** *resid* ::  $'a \text{ resid}$  (**infix**  $\langle \setminus \rangle$  70) +

**assumes** *con-sym-ax*:  $t \setminus u \neq \text{null} \Longrightarrow u \setminus t \neq \text{null}$

**and** *con-imp-arr-resid*:  $t \setminus u \neq \text{null} \Longrightarrow (t \setminus u) \setminus (t \setminus u) \neq \text{null}$

**and** *cube-ax*:  $(v \setminus t) \setminus (u \setminus t) \neq \text{null} \Longrightarrow (v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$

**begin**

The axiom *cube-ax* is equivalent to the following unconditional form. The locale assumptions use the weaker form to avoid having to treat the case  $(v \setminus t) \setminus (u \setminus t) = \text{null}$  specially for every interpretation.

**lemma** *cube*:

**shows**  $(v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$

**using** *cube-ax* **by** *metis*

We regard  $t$  and  $u$  as *consistent* if the residuation  $t \setminus u$  is defined. It is convenient to make this a definition, with associated notation.

**definition** *con* (**infix**  $\langle \frown \rangle$  50)

**where**  $t \frown u \equiv t \setminus u \neq \text{null}$

**lemma** *conI* [*intro*]:

**assumes**  $t \setminus u \neq \text{null}$

**shows**  $t \frown u$

**using** *assms con-def* **by** *blast*

**lemma** *conE* [*elim*]:

**assumes**  $t \frown u$

**and**  $t \setminus u \neq \text{null} \Longrightarrow T$

**shows**  $T$

**using** *assms con-def* **by** *simp*

**lemma** *con-sym*:

**assumes**  $t \frown u$

**shows**  $u \frown t$

**using** *assms con-def con-sym-ax* **by** *blast*

We call  $t$  an *arrow* if it is self-consistent.

**definition** *arr*

**where**  $arr\ t \equiv t \frown t$

**lemma** *arrI* [*intro*]:  
**assumes**  $t \frown t$   
**shows**  $arr\ t$   
**using** *assms arr-def* **by** *simp*

**lemma** *arrE* [*elim*]:  
**assumes**  $arr\ t$   
**and**  $t \frown t \implies T$   
**shows**  $T$   
**using** *assms arr-def* **by** *simp*

**lemma** *not-arr-null* [*simp*]:  
**shows**  $\neg arr\ null$   
**by** (*auto simp add: con-def*)

**lemma** *con-implies-arr*:  
**assumes**  $t \frown u$   
**shows**  $arr\ t$  **and**  $arr\ u$   
**using** *assms*  
**by** (*metis arrI con-def con-imp-arr-resid cube null-is-zero(2)*)**+**

**lemma** *arr-resid* [*simp*]:  
**assumes**  $t \frown u$   
**shows**  $arr\ (t \setminus u)$   
**using** *assms con-imp-arr-resid* **by** *blast*

**lemma** *arr-resid-iff-con*:  
**shows**  $arr\ (t \setminus u) \iff t \frown u$   
**by** *auto*

**lemma** *con-arr-self* [*simp*]:  
**assumes**  $arr\ f$   
**shows**  $f \frown f$   
**using** *assms arrE* **by** *auto*

**lemma** *not-con-null* [*simp*]:  
**shows**  $con\ null\ t = False$  **and**  $con\ t\ null = False$   
**by** *auto*

The residuation of an arrow along itself is the *canonical target* of the arrow.

**definition** *trg*  
**where**  $trg\ t \equiv t \setminus t$

**lemma** *resid-arr-self*:  
**shows**  $t \setminus t = trg\ t$   
**using** *trg-def* **by** *auto*

**lemma** *arr-trg-iff-arr*:  
**shows**  $\text{arr } (\text{trg } t) \longleftrightarrow \text{arr } t$   
**by** (*metis arrI arrE arr-resid-iff-con resid-arr-self*)

An *identity* is an arrow that is its own target.

**definition** *ide*  
**where**  $\text{ide } a \equiv a \frown a \wedge a \setminus a = a$

**lemma** *ideI* [*intro*]:  
**assumes**  $a \frown a$  **and**  $a \setminus a = a$   
**shows**  $\text{ide } a$   
**using** *assms ide-def* **by** *auto*

**lemma** *ideE* [*elim*]:  
**assumes**  $\text{ide } a$   
**and**  $\llbracket a \frown a; a \setminus a = a \rrbracket \Longrightarrow T$   
**shows**  $T$   
**using** *assms ide-def* **by** *blast*

**lemma** *ide-implies-arr* [*simp*]:  
**assumes**  $\text{ide } a$   
**shows**  $\text{arr } a$   
**using** *assms* **by** *blast*

**lemma** *not-ide-null* [*simp*]:  
**shows**  $\text{ide } \text{null} = \text{False}$   
**by** *auto*

**end**

### 2.1.3 Residuated Transition System

A *residuated transition system* consists of a residuation subject to additional axioms that concern the relationship between identities and residuation. These axioms make it possible to sensibly associate with each arrow certain nonempty sets of identities called the *sources* and *targets* of the arrow. Axiom *ide-trg* states that the canonical target *trg*  $t$  of an arrow  $t$  is an identity. Axiom *resid-arr-ide* states that identities are right units for residuation, when it is defined. Axiom *resid-ide-arr* states that the residuation of an identity along an arrow is again an identity, assuming that the residuation is defined. Axiom *con-imp-coinitial-ax* states that if arrows  $t$  and  $u$  are consistent, then there is an identity that is consistent with both of them (*i.e.* they have a common source). Axiom *con-target* states that an identity of the form  $t \setminus u$  (which may be regarded as a “target” of  $u$ ) is consistent with any other arrow  $v \setminus u$  obtained by residuation along  $u$ . We note that replacing the premise  $\text{ide } (t \setminus u)$  in this axiom by either  $\text{arr } (t \setminus u)$  or  $t \frown u$  would result in a strictly stronger statement.

**locale** *rts = residuation* +  
**assumes** *ide-trg* [*simp*]:  $\text{arr } t \Longrightarrow \text{ide } (\text{trg } t)$

**and** *resid-arr-ide*:  $\llbracket \text{ide } a; t \frown a \rrbracket \Longrightarrow t \setminus a = t$   
**and** *resid-ide-arr* [*simp*]:  $\llbracket \text{ide } a; a \frown t \rrbracket \Longrightarrow \text{ide } (a \setminus t)$   
**and** *con-imp-coinitial-ax*:  $t \frown u \Longrightarrow \exists a. \text{ide } a \wedge a \frown t \wedge a \frown u$   
**and** *con-target*:  $\llbracket \text{ide } (t \setminus u); u \frown v \rrbracket \Longrightarrow t \setminus u \frown v \setminus u$   
**begin**

We define the *sources* of an arrow  $t$  to be the identities that are consistent with  $t$ .

**definition** *sources*  
**where** *sources*  $t = \{a. \text{ide } a \wedge t \frown a\}$

We define the *targets* of an arrow  $t$  to be the identities that are consistent with the canonical target  $\text{trg } t$ .

**definition** *targets*  
**where** *targets*  $t = \{b. \text{ide } b \wedge \text{trg } t \frown b\}$

**lemma** *in-sourcesI* [*intro, simp*]:  
**assumes** *ide*  $a$  **and**  $t \frown a$   
**shows**  $a \in \text{sources } t$   
**using** *assms sources-def* **by** *simp*

**lemma** *in-sourcesE* [*elim*]:  
**assumes**  $a \in \text{sources } t$   
**and**  $\llbracket \text{ide } a; t \frown a \rrbracket \Longrightarrow T$   
**shows**  $T$   
**using** *assms sources-def* **by** *auto*

**lemma** *in-targetsI* [*intro, simp*]:  
**assumes** *ide*  $b$  **and**  $\text{trg } t \frown b$   
**shows**  $b \in \text{targets } t$   
**using** *assms targets-def resid-arr-self* **by** *simp*

**lemma** *in-targetsE* [*elim*]:  
**assumes**  $b \in \text{targets } t$   
**and**  $\llbracket \text{ide } b; \text{trg } t \frown b \rrbracket \Longrightarrow T$   
**shows**  $T$   
**using** *assms targets-def resid-arr-self* **by** *force*

**lemma** *trg-in-targets*:  
**assumes** *arr*  $t$   
**shows**  $\text{trg } t \in \text{targets } t$   
**using** *assms*  
**by** (*meson ideE ide-trg in-targetsI*)

**lemma** *source-is-ide*:  
**assumes**  $a \in \text{sources } t$   
**shows** *ide*  $a$   
**using** *assms* **by** *blast*

**lemma** *target-is-ide*:

**assumes**  $a \in \text{targets } t$   
**shows**  $\text{ide } a$   
**using**  $\text{assms by blast}$

Consistent arrows have a common source.

**lemma** *con-imp-common-source*:  
**assumes**  $t \frown u$   
**shows**  $\text{sources } t \cap \text{sources } u \neq \{\}$   
**using**  $\text{assms}$   
**by** ( $\text{meson disjoint-iff in-sourcesI con-imp-coinitial-ax con-sym}$ )

Arrows are characterized by the property of having a nonempty set of sources, or equivalently, by that of having a nonempty set of targets.

**lemma** *arr-iff-has-source*:  
**shows**  $\text{arr } t \iff \text{sources } t \neq \{\}$   
**using**  $\text{con-imp-common-source con-implies-arr(1) sources-def by blast}$

**lemma** *arr-iff-has-target*:  
**shows**  $\text{arr } t \iff \text{targets } t \neq \{\}$   
**using**  $\text{trg-def trg-in-targets by fastforce}$

The residuation of a source of an arrow along that arrow gives a target of the same arrow. However, it is *not* true that every target of an arrow  $t$  is of the form  $u \setminus t$  for some  $u$  with  $t \frown u$ .

**lemma** *resid-source-in-targets*:  
**assumes**  $a \in \text{sources } t$   
**shows**  $a \setminus t \in \text{targets } t$   
**by** ( $\text{metis arr-resid assms con-target con-sym resid-arr-ide ide-trg in-sourcesE resid-ide-arr in-targetsI resid-arr-self}$ )

Residuation along an identity reflects identities.

**lemma** *ide-backward-stable*:  
**assumes**  $\text{ide } a$  **and**  $\text{ide } (t \setminus a)$   
**shows**  $\text{ide } t$   
**by** ( $\text{metis assms ideE resid-arr-ide arr-resid-iff-con}$ )

**lemma** *resid-reflects-con*:  
**assumes**  $t \frown v$  **and**  $u \frown v$  **and**  $t \setminus v \frown u \setminus v$   
**shows**  $t \frown u$   
**using**  $\text{assms cube}$   
**by** ( $\text{elim conE}$ ) *auto*

**lemma** *con-transitive-on-ide*:  
**assumes**  $\text{ide } a$  **and**  $\text{ide } b$  **and**  $\text{ide } c$   
**shows**  $\llbracket a \frown b; b \frown c \rrbracket \implies a \frown c$   
**using**  $\text{assms}$   
**by** ( $\text{metis resid-arr-ide con-target con-sym}$ )

**lemma** *sources-are-con*:

**assumes**  $a \in \text{sources } t$  **and**  $a' \in \text{sources } t$   
**shows**  $a \frown a'$   
**using** *assms*  
**by** (*metis (no-types, lifting) CollectD con-target con-sym resid-ide-arr*  
*sources-def resid-reflects-con*)

**lemma** *sources-con-closed*:  
**assumes**  $a \in \text{sources } t$  **and** *ide*  $a'$  **and**  $a \frown a'$   
**shows**  $a' \in \text{sources } t$   
**using** *assms*  
**by** (*metis (no-types, lifting) con-target con-sym resid-arr-ide*  
*mem-Collect-eq sources-def*)

**lemma** *sources-eqI*:  
**assumes**  $\text{sources } t \cap \text{sources } t' \neq \{\}$   
**shows**  $\text{sources } t = \text{sources } t'$   
**using** *assms sources-def sources-are-con sources-con-closed* **by** *blast*

**lemma** *targets-are-con*:  
**assumes**  $b \in \text{targets } t$  **and**  $b' \in \text{targets } t$   
**shows**  $b \frown b'$   
**using** *assms sources-are-con sources-def targets-def* **by** *blast*

**lemma** *targets-con-closed*:  
**assumes**  $b \in \text{targets } t$  **and** *ide*  $b'$  **and**  $b \frown b'$   
**shows**  $b' \in \text{targets } t$   
**using** *assms sources-con-closed sources-def targets-def* **by** *blast*

**lemma** *targets-eqI*:  
**assumes**  $\text{targets } t \cap \text{targets } t' \neq \{\}$   
**shows**  $\text{targets } t = \text{targets } t'$   
**using** *assms targets-def targets-are-con targets-con-closed* **by** *blast*

Arrows are *coinitial* if they have a common source, and *coterminal* if they have a common target.

**definition** *coinitial*  
**where** *coinitial*  $t u \equiv \text{sources } t \cap \text{sources } u \neq \{\}$

**definition** *coterminal*  
**where** *coterminal*  $t u \equiv \text{targets } t \cap \text{targets } u \neq \{\}$

**lemma** *coinitialI* [*intro*]:  
**assumes** *arr*  $t$  **and**  $\text{sources } t = \text{sources } u$   
**shows** *coinitial*  $t u$   
**using** *assms coinitial-def arr-iff-has-source* **by** *simp*

**lemma** *coinitialE* [*elim*]:  
**assumes** *coinitial*  $t u$   
**and**  $\llbracket \text{arr } t; \text{arr } u; \text{sources } t = \text{sources } u \rrbracket \implies T$

**shows**  $T$   
**using** *assms coinitial-def sources-eqI arr-iff-has-source* **by** *auto*

**lemma** *con-imp-coinitial*:  
**assumes**  $t \frown u$   
**shows** *coinitial*  $t$   $u$   
**using** *assms*  
**by** (*simp add: coinitial-def con-imp-common-source*)

**lemma** *coinitial-iff*:  
**shows** *coinitial*  $t$   $t' \iff arr\ t \wedge arr\ t' \wedge sources\ t = sources\ t'$   
**by** (*metis arr-iff-has-source coinitial-def inf-idem sources-eqI*)

**lemma** *coterminal-iff*:  
**shows** *coterminal*  $t$   $t' \iff arr\ t \wedge arr\ t' \wedge targets\ t = targets\ t'$   
**by** (*metis arr-iff-has-target coterminal-def inf-idem targets-eqI*)

**lemma** *coterminal-iff-con-trg*:  
**shows** *coterminal*  $t$   $u \iff trg\ t \frown trg\ u$   
**by** (*metis coinitial-iff con-imp-coinitial coterminal-iff in-targetsE trg-in-targets resid-arr-self arr-resid-iff-con sources-def targets-def*)

**lemma** *coterminalI* [*intro*]:  
**assumes**  $arr\ t$  **and**  $targets\ t = targets\ u$   
**shows** *coterminal*  $t$   $u$   
**using** *assms coterminal-iff arr-iff-has-target* **by** *auto*

**lemma** *coterminalE* [*elim*]:  
**assumes** *coterminal*  $t$   $u$   
**and**  $\llbracket arr\ t; arr\ u; targets\ t = targets\ u \rrbracket \implies T$   
**shows**  $T$   
**using** *assms coterminal-iff* **by** *auto*

**lemma** *sources-resid* [*simp*]:  
**assumes**  $t \frown u$   
**shows**  $sources\ (t \setminus u) = targets\ u$   
**unfolding** *targets-def trg-def*  
**using** *assms conI conE*  
**by** (*metis con-imp-arr-resid assms coinitial-iff con-imp-coinitial cube ex-un-null sources-def*)

**lemma** *targets-resid-sym*:  
**assumes**  $t \frown u$   
**shows**  $targets\ (t \setminus u) = targets\ (u \setminus t)$   
**using** *assms*  
**apply** (*intro targets-eqI*)  
**by** (*metis (no-types, opaque-lifting) assms cube inf-idem arr-iff-has-target arr-def arr-resid-iff-con sources-resid*)

Arrows  $t$  and  $u$  are *sequential* if the set of targets of  $t$  equals the set of sources of  $u$ .

**definition** *seq*  
**where**  $seq\ t\ u \equiv arr\ t \wedge arr\ u \wedge targets\ t = sources\ u$

**lemma** *seqI* [*intro*]:  
**shows**  $\llbracket arr\ t; targets\ t = sources\ u \rrbracket \implies seq\ t\ u$   
**and**  $\llbracket arr\ u; targets\ t = sources\ u \rrbracket \implies seq\ t\ u$   
**using** *seq-def arr-iff-has-source arr-iff-has-target* **by** *metis+*

**lemma** *seqE* [*elim*]:  
**assumes**  $seq\ t\ u$   
**and**  $\llbracket arr\ t; arr\ u; targets\ t = sources\ u \rrbracket \implies T$   
**shows**  $T$   
**using** *assms seq-def* **by** *blast*

## Congruence of Transitions

Residuation induces a preorder  $\lesssim$  on transitions, defined by  $t \lesssim u$  if and only if  $t \setminus u$  is an identity.

**abbreviation** *prfx* (**infix**  $\langle \lesssim \rangle$  50)  
**where**  $t \lesssim u \equiv ide\ (t \setminus u)$

**lemma** *prfxE*:  
**assumes**  $t \lesssim u$   
**and**  $ide\ (t \setminus u) \implies T$   
**shows**  $T$   
**using** *assms* **by** *fastforce*

**lemma** *prfx-implies-con*:  
**assumes**  $t \lesssim u$   
**shows**  $t \frown u$   
**using** *assms arr-resid-iff-con* **by** *blast*

**lemma** *prfx-reflexive*:  
**assumes**  $arr\ t$   
**shows**  $t \lesssim t$   
**by** (*simp add: assms resid-arr-self*)

**lemma** *prfx-transitive* [*trans*]:  
**assumes**  $t \lesssim u$  **and**  $u \lesssim v$   
**shows**  $t \lesssim v$   
**using** *assms con-target resid-ide-arr ide-backward-stable cube conI*  
**by** *metis*

**lemma** *source-is-prfx*:  
**assumes**  $a \in sources\ t$   
**shows**  $a \lesssim t$   
**using** *assms resid-source-in-targets* **by** *blast*

The equivalence  $\sim$  associated with  $\lesssim$  is substitutive with respect to residuation.

**abbreviation** *cong* (**infix**  $\langle \sim \rangle$  50)  
**where**  $t \sim u \equiv t \lesssim u \wedge u \lesssim t$

**lemma** *congE*:  
**assumes**  $t \sim u$   
**and**  $\llbracket t \frown u; \text{ide } (t \setminus u); \text{ide } (u \setminus t) \rrbracket \implies T$   
**shows**  $T$   
**using** *assms prfx-implies-con* **by** *blast*

**lemma** *cong-reflexive*:  
**assumes** *arr*  $t$   
**shows**  $t \sim t$   
**using** *assms prfx-reflexive* **by** *simp*

**lemma** *cong-symmetric*:  
**assumes**  $t \sim u$   
**shows**  $u \sim t$   
**using** *assms* **by** *simp*

**lemma** *cong-transitive* [*trans*]:  
**assumes**  $t \sim u$  **and**  $u \sim v$   
**shows**  $t \sim v$   
**using** *assms prfx-transitive* **by** *auto*

**lemma** *cong-subst-left*:  
**assumes**  $t \sim t'$  **and**  $t \frown u$   
**shows**  $t' \frown u$  **and**  $t \setminus u \sim t' \setminus u$   
**apply** (*meson assms con-sym con-target prfx-implies-con resid-reflects-con*)  
**by** (*metis assms con-sym con-target cube prfx-implies-con resid-ide-arr resid-reflects-con*)

**lemma** *cong-subst-right*:  
**assumes**  $u \sim u'$  **and**  $t \frown u$   
**shows**  $t \frown u'$  **and**  $t \setminus u \sim t \setminus u'$   
**proof** –  
**have**  $1: t \frown u' \wedge t \setminus u' \frown u \setminus u' \wedge$   
 $(t \setminus u) \setminus (u' \setminus u) = (t \setminus u') \setminus (u \setminus u')$   
**using** *assms cube con-sym con-target cong-subst-left(1)* **by** *meson*  
**show**  $t \frown u'$   
**using**  $1$  **by** *simp*  
**show**  $t \setminus u \sim t \setminus u'$   
**by** (*metis 1 arr-resid-iff-con assms(1) cong-reflexive resid-arr-ide*)  
**qed**

**lemma** *cong-implies-coinitial*:  
**assumes**  $u \sim u'$   
**shows** *coinitial*  $u$   $u'$   
**using** *assms con-imp-coinitial prfx-implies-con* **by** *simp*

**lemma** *cong-implies-coterminal*:

**assumes**  $u \sim u'$   
**shows** *coterminal*  $u u'$   
**using** *assms*  
**by** (*metis con-implies-arr(1) coterminalI ideE prfx-implies-con sources-resid targets-resid-sym*)

**lemma** *ide-imp-con-iff-cong*:  
**assumes** *ide*  $t$  **and** *ide*  $u$   
**shows**  $t \frown u \longleftrightarrow t \sim u$   
**using** *assms*  
**by** (*metis con-sym resid-ide-arr prfx-implies-con*)

**lemma** *sources-are-cong*:  
**assumes**  $a \in \text{sources } t$  **and**  $a' \in \text{sources } t$   
**shows**  $a \sim a'$   
**using** *assms sources-are-con*  
**by** (*metis CollectD ide-imp-con-iff-cong sources-def*)

**lemma** *sources-cong-closed*:  
**assumes**  $a \in \text{sources } t$  **and**  $a \sim a'$   
**shows**  $a' \in \text{sources } t$   
**using** *assms sources-def*  
**by** (*meson in-sourcesE in-sourcesI cong-subst-right(1) ide-backward-stable*)

**lemma** *targets-are-cong*:  
**assumes**  $b \in \text{targets } t$  **and**  $b' \in \text{targets } t$   
**shows**  $b \sim b'$   
**using** *assms(1-2) sources-are-cong sources-def targets-def* **by** *blast*

**lemma** *targets-cong-closed*:  
**assumes**  $b \in \text{targets } t$  **and**  $b \sim b'$   
**shows**  $b' \in \text{targets } t$   
**using** *assms targets-def sources-cong-closed sources-def* **by** *blast*

**lemma** *targets-char*:  
**shows**  $\text{targets } t = \{b. \text{arr } t \wedge t \setminus t \sim b\}$   
**unfolding** *targets-def*  
**by** (*metis (no-types, lifting) con-def con-implies-arr(2) con-sym cong-reflexive ide-def resid-arr-ide trg-def*)

**lemma** *coinitial-ide-are-cong*:  
**assumes** *ide*  $a$  **and** *ide*  $a'$  **and** *coinitial*  $a a'$   
**shows**  $a \sim a'$   
**using** *assms coinitial-def*  
**by** (*metis ideE in-sourcesI coinitialE sources-are-cong*)

**lemma** *cong-respects-seq*:  
**assumes** *seq*  $t u$  **and** *cong*  $t t'$  **and** *cong*  $u u'$   
**shows** *seq*  $t' u'$

by (*metis* *assms* *coterminalE* *coinitialE* *cong-implies-coinitial*  
*cong-implies-coterminal* *seqE* *seqI*(1))

## Chosen Sources

In a general RTS, sources are not unique and (in contrast to the case for targets) there isn't even any canonical source. However, it is useful to choose an arbitrary source for each transition. Once we have at least weak extensionality, this will be the unique source and stronger things can be proved about it.

**definition** *src*

**where** *src*  $t \equiv$  if *arr*  $t$  then *SOME*  $a$ .  $a \in$  *sources*  $t$  else *null*

**lemma** *src-in-sources*:

**assumes** *arr*  $t$

**shows** *src*  $t \in$  *sources*  $t$

**using** *assms* *someI-ex* [of  $\lambda a$ .  $a \in$  *sources*  $t$ ] *arr-iff-has-source* *src-def*

**by** *auto*

**lemma** *src-congI*:

**assumes** *ide*  $a$  **and**  $a \frown t$

**shows** *src*  $t \sim a$

**using** *assms* *src-in-sources* *sources-are-cong*

**by** (*metis* *arr-iff-has-source* *con-sym* *emptyE* *in-sourcesI*)

**lemma** *arr-src-iff-arr*:

**shows** *arr* (*src*  $t$ )  $\longleftrightarrow$  *arr*  $t$

**by** (*metis* *arrI* *conE* *null-is-zero*(2) *sources-are-con* *arrE* *src-def* *src-in-sources*)

**lemma** *arr-src-if-arr* [*simp*]:

**assumes** *arr*  $t$

**shows** *arr* (*src*  $t$ )

**using** *assms* *arr-src-iff-arr* **by** *blast*

**lemma** *sources-char<sub>CS</sub>*:

**shows** *sources*  $t = \{a$ . *arr*  $t \wedge$  *src*  $t \sim a\}$

**unfolding** *sources-def*

**by** (*meson* *con-implies-arr*(1) *con-sym* *in-sourcesE* *sources-cong-closed*

*src-congI* *src-in-sources*)

**lemma** *targets-char'*:

**shows** *targets*  $t = \{b$ . *arr*  $t \wedge$  *trg*  $t \sim b\}$

**unfolding** *targets-def*

**using** *targets-char* *targets-def* *trg-def* **by** *presburger*

**lemma** *con-imp-cong-src*:

**assumes**  $t \frown u$

**shows** *src*  $t \sim$  *src*  $u$

**using** *assms* *con-imp-coinitial-ax* *cong-transitive* *src-congI* **by** *blast*

**lemma** *ide-src* [*simp*]:  
**assumes** *arr t*  
**shows** *ide (src t)*  
**using** *assms src-in-sources* **by** *blast*

**lemma** *src-resid*:  
**assumes**  $t \frown u$   
**shows**  $\text{src } (t \setminus u) \sim \text{trg } u$   
**using** *assms*  
**by** (*metis con-implies-arr(2) con-sym con-target ide-trg src-congI trg-def*)

**lemma** *apex-arr-prfx'*:  
**assumes** *prfx t u*  
**shows**  $\text{trg } (u \setminus t) \sim \text{trg } u$   
**and**  $\text{trg } (t \setminus u) \sim \text{trg } u$   
**using** *assms*  
**apply** (*metis (no-types, lifting) ide-def mem-Collect-eq prfx-implies-con sources-resid targets-resid-sym sources-def targets-char' trg-in-targets*)  
**by** (*metis assms ideE prfx-transitive arr-resid-iff-con src-resid*)

**lemma** *seqI<sub>CS</sub>* [*intro, simp*]:  
**shows**  $\llbracket \text{arr } t; \text{trg } t \sim \text{src } u \rrbracket \implies \text{seq } t u$   
**and**  $\llbracket \text{arr } u; \text{trg } t \sim \text{src } u \rrbracket \implies \text{seq } t u$   
**apply** (*metis ide-trg in-sourcesE not-ide-null prfx-implies-con resid-arr-ide seqI(1) sources-resid src-def src-in-sources trg-def*)  
**by** (*metis in-sourcesE prfx-implies-con resid-arr-ide seqI(2) sources-resid src-in-sources trg-def*)

**lemma** *seqE<sub>CS</sub>* [*elim*]:  
**assumes** *seq t u*  
**and**  $\llbracket \text{arr } u; \text{arr } t; \text{trg } t \sim \text{src } u \rrbracket \implies T$   
**shows** *T*  
**using** *assms*  
**by** (*metis seq-def sources-are-cong src-in-sources trg-in-targets*)

**lemma** *coinitial-iff'*:  
**shows**  $\text{coinitial } t u \iff \text{arr } t \wedge \text{arr } u \wedge \text{src } t \sim \text{src } u$   
**by** (*metis (full-types) arr-resid-iff-con coinitialE coinitialI ide-implies-arr resid-arr-ide con-imp-coinitial in-sourcesE src-in-sources*)

**lemma** *coterminal-iff'*:  
**shows**  $\text{coterminal } t u \iff \text{arr } t \wedge \text{arr } u \wedge \text{trg } t \sim \text{trg } u$   
**by** (*meson coterminalE ide-imp-con-iff-cong coterminal-iff-con-trg ide-trg*)

**lemma** *coinitialI'* [*intro*]:  
**assumes** *arr t* **and**  $\text{src } t \sim \text{src } u$   
**shows** *coinitial t u*  
**by** (*metis assms(2) coinitial-iff' not-ide-null null-is-zero(2) src-def*)

**lemma** *coinitialE'* [*elim*]:  
**assumes** *coinitial t u*  
**and**  $\llbracket \text{arr } t; \text{arr } u; \text{src } t \sim \text{src } u \rrbracket \implies T$   
**shows**  $T$   
**using** *assms coinitial-iff'* **by** *blast*

**lemma** *coterminalI'* [*intro*]:  
**assumes** *arr t* **and** *trg t  $\sim$  trg u*  
**shows** *coterminal t u*  
**by** (*simp add: assms(2) prfx-implies-con coterminal-iff-con-trg*)

**lemma** *coterminalE'* [*elim*]:  
**assumes** *coterminal t u*  
**and**  $\llbracket \text{arr } t; \text{arr } u; \text{trg } t \sim \text{trg } u \rrbracket \implies T$   
**shows**  $T$   
**using** *assms coterminal-iff'* **by** *blast*

**lemma** *src-cong-ide*:  
**assumes** *ide a*  
**shows** *src a  $\sim$  a*  
**using** *arrI assms src-congI* **by** *blast*

**lemma** *trg-ide* [*simp*]:  
**assumes** *ide a*  
**shows** *trg a = a*  
**using** *assms resid-arr-self* **by** *auto*

**lemma** *ide-iff-src-cong-self*:  
**assumes** *arr a*  
**shows** *ide a  $\longleftrightarrow$  src a  $\sim$  a*  
**by** (*metis assms ide-backward-stable in-sourcesE src-cong-ide src-in-sources*)

**lemma** *ide-iff-trg-cong-self*:  
**assumes** *arr a*  
**shows** *ide a  $\longleftrightarrow$  trg a  $\sim$  a*  
**by** (*metis assms ideE ide-backward-stable ide-trg trg-def*)

**lemma** *src-src-cong-src*:  
**assumes** *arr t*  
**shows** *src (src t)  $\sim$  src t*  
**using** *assms src-cong-ide src-in-sources* **by** *blast*

**lemma** *trg-trg-cong-trg*:  
**assumes** *arr t*  
**shows** *trg (trg t)  $\sim$  trg t*  
**using** *assms* **by** *fastforce*

**lemma** *src-trg-cong-trg*:  
**assumes** *arr t*

```

shows  $\text{src } (\text{trg } t) \sim \text{trg } t$ 
  using assms ide-trg src-cong-ide by blast

lemma trg-src-cong-src:
assumes arr t
shows  $\text{trg } (\text{src } t) \sim \text{src } t$ 
  using assms
  by (metis in-sourcesE resid-arr-self trg-ide src-in-sources)

lemma resid-ide-cong:
assumes ide a and coinitial a t
shows  $t \setminus a \sim t$  and  $a \setminus t \sim \text{trg } t$ 
  using assms
  apply (metis coinitialE cong-reflexive ideE in-sourcesE in-sourcesI resid-arr-ide)
  by (metis apex-arr-prfx'(2) assms coinitialE ideE in-sourcesI resid-arr-self
    source-is-prfx)

lemma con-arr-src [simp]:
assumes arr f
shows  $f \frown \text{src } f$  and  $\text{src } f \frown f$ 
  using assms src-in-sources con-sym by blast+

lemma resid-src-arr-cong:
assumes arr f
shows  $\text{src } f \setminus f \sim \text{trg } f$ 
  using assms
  by (meson resid-source-in-targets src-in-sources trg-in-targets targets-are-cong)

lemma resid-arr-src-cong:
assumes arr f
shows  $f \setminus \text{src } f \sim f$ 
  using assms
  by (metis cong-reflexive in-sourcesE resid-arr-ide src-in-sources)

end

```

### 2.1.4 Weakly Extensional RTS

A *weakly extensional* RTS is an RTS that satisfies the additional condition that identity arrows have trivial congruence classes. This axiom has a number of useful consequences, including that each arrow has a unique source and target.

```

locale weakly-extensional-rts = rts +
assumes weak-extensionality:  $\llbracket t \sim u; \text{ide } t; \text{ide } u \rrbracket \implies t = u$ 
begin

```

```

lemma con-ide-are-eq:
assumes ide a and ide a' and  $a \frown a'$ 
shows  $a = a'$ 
  using assms ide-imp-con-iff-cong weak-extensionality by blast

```

**lemma** *coinitial-ide-are-eq*:

**assumes** *ide a* **and** *ide a'* **and** *coinitial a a'*

**shows**  $a = a'$

**using** *assms coinitial-def con-ide-are-eq* **by** *blast*

**lemma** *arr-has-un-source*:

**assumes** *arr t*

**shows**  $\exists!a. a \in \text{sources } t$

**using** *assms*

**by** (*meson arr-iff-has-source con-ide-are-eq ex-in-conv in-sourcesE sources-are-con*)

**lemma** *arr-has-un-target*:

**assumes** *arr t*

**shows**  $\exists!b. b \in \text{targets } t$

**using** *assms*

**by** (*metis arrE arr-has-un-source arr-resid sources-resid*)

**lemma** *src-eqI*:

**assumes** *ide a* **and**  $a \frown t$

**shows**  $\text{src } t = a$

**using** *assms src-in-sources*

**by** (*metis arr-has-un-source resid-arr-ide in-sourcesI arr-resid-iff-con con-sym*)

**lemma** *sources-char<sub>WE</sub>*:

**shows**  $\text{sources } t = \{a. \text{arr } t \wedge \text{src } t = a\}$

**using** *arr-iff-has-source con-sym src-eqI* **by** *auto*

**lemma** *targets-char<sub>WE</sub>*:

**shows**  $\text{targets } t = \{b. \text{arr } t \wedge \text{trg } t = b\}$

**using** *trg-in-targets arr-has-un-target arr-iff-has-target* **by** *auto*

**lemma** *con-imp-eq-src*:

**assumes**  $t \frown u$

**shows**  $\text{src } t = \text{src } u$

**using** *assms*

**by** (*metis con-imp-coinitial-ax src-eqI*)

**lemma** *src-resid<sub>WE</sub> [simp]*:

**assumes**  $t \frown u$

**shows**  $\text{src } (t \setminus u) = \text{trg } u$

**using** *assms*

**by** (*metis arr-resid-iff-con con-implies-arr(2) arr-has-un-source trg-in-targets sources-resid src-in-sources*)

**lemma** *apex-sym*:

**shows**  $\text{trg } (t \setminus u) = \text{trg } (u \setminus t)$

**by** (*metis arr-has-un-target con-sym-ax arr-resid-iff-con conI targets-resid-sym trg-in-targets*)

**lemma** *apex-arr-prfx*<sub>WE</sub>:  
**assumes** *prfx t u*  
**shows**  $\text{trg } (u \setminus t) = \text{trg } u$   
**and**  $\text{trg } (t \setminus u) = \text{trg } u$   
**using** *assms*  
**apply** (*metis apex-sym arr-resid-iff-con ideE src-resid*<sub>WE</sub>)  
**by** (*metis arr-resid-iff-con assms ideE src-resid*<sub>WE</sub>)

**lemma** *seqI*<sub>WE</sub> [*intro, simp*]:  
**shows**  $\llbracket \text{arr } t; \text{trg } t = \text{src } u \rrbracket \implies \text{seq } t u$   
**and**  $\llbracket \text{arr } u; \text{trg } t = \text{src } u \rrbracket \implies \text{seq } t u$   
**by** (*metis arrE arr-src-iff-arr arr-trg-iff-arr in-sourcesE resid-arr-ide*  
*seqI(1) sources-resid src-in-sources trg-def*)<sup>+</sup>

**lemma** *seqE*<sub>WE</sub> [*elim*]:  
**assumes** *seq t u*  
**and**  $\llbracket \text{arr } u; \text{arr } t; \text{trg } t = \text{src } u \rrbracket \implies T$   
**shows** *T*  
**using** *assms*  
**by** (*metis arr-has-un-source seq-def src-in-sources trg-in-targets*)

**lemma** *coinitial-iff*<sub>WE</sub>:  
**shows**  $\text{coinitial } t u \iff \text{arr } t \wedge \text{arr } u \wedge \text{src } t = \text{src } u$   
**by** (*metis arr-has-un-source coinitial-def coinitial-iff disjoint-iff-not-equal*  
*src-in-sources*)

**lemma** *coterminal-iff*<sub>WE</sub>:  
**shows**  $\text{coterminal } t u \iff \text{arr } t \wedge \text{arr } u \wedge \text{trg } t = \text{trg } u$   
**by** (*metis arr-has-un-target coterminal-iff-con-trg coterminal-iff trg-in-targets*)

**lemma** *coinitialI*<sub>WE</sub> [*intro*]:  
**assumes** *arr t* **and** *src t = src u*  
**shows** *coinitial t u*  
**using** *assms coinitial-iff*<sub>WE</sub> **by** (*metis arr-src-iff-arr*)

**lemma** *coinitialE*<sub>WE</sub> [*elim*]:  
**assumes** *coinitial t u*  
**and**  $\llbracket \text{arr } t; \text{arr } u; \text{src } t = \text{src } u \rrbracket \implies T$   
**shows** *T*  
**using** *assms coinitial-iff*<sub>WE</sub> **by** *blast*

**lemma** *coterminalI*<sub>WE</sub> [*intro*]:  
**assumes** *arr t* **and** *trg t = trg u*  
**shows** *coterminal t u*  
**using** *assms coterminal-iff*<sub>WE</sub> **by** (*metis arr-trg-iff-arr*)

**lemma** *coterminalE*<sub>WE</sub> [*elim*]:  
**assumes** *coterminal t u*

**and**  $\llbracket \text{arr } t; \text{arr } u; \text{trg } t = \text{trg } u \rrbracket \implies T$   
**shows**  $T$   
**using** *assms coterminial-iff<sub>WE</sub>* **by** *blast*

**lemma** *src-ide* [*simp*]:  
**assumes** *ide a*  
**shows**  $\text{src } a = a$   
**using** *arrI assms src-eqI* **by** *blast*

**lemma** *ide-iff-src-self*:  
**assumes** *arr a*  
**shows**  $\text{ide } a \longleftrightarrow \text{src } a = a$   
**using** *assms* **by** (*metis ide-src src-ide*)

**lemma** *ide-iff-trg-self*:  
**assumes** *arr a*  
**shows**  $\text{ide } a \longleftrightarrow \text{trg } a = a$   
**using** *assms ide-def resid-arr-self ide-trg* **by** *metis*

**lemma** *src-src* [*simp*]:  
**shows**  $\text{src } (\text{src } t) = \text{src } t$   
**using** *ide-src src-def src-ide* **by** *auto*

**lemma** *trg-trg* [*simp*]:  
**shows**  $\text{trg } (\text{trg } t) = \text{trg } t$   
**by** (*metis con-def con-implies-arr(2) cong-reflexive ide-def null-is-zero(2) resid-arr-self*)

**lemma** *src-trg* [*simp*]:  
**shows**  $\text{src } (\text{trg } t) = \text{trg } t$   
**by** (*metis con-def not-arr-null src-def src-resid<sub>WE</sub> trg-def*)

**lemma** *trg-src* [*simp*]:  
**shows**  $\text{trg } (\text{src } t) = \text{src } t$   
**by** (*metis ide-src null-is-zero(2) resid-arr-self src-def trg-ide*)

**lemma** *resid-ide*:  
**assumes** *ide a* **and** *coinitial a t*  
**shows**  $t \setminus a = t$  **and**  $a \setminus t = \text{trg } t$   
**using** *assms resid-arr-ide* **apply** *blast*  
**using** *assms*  
**by** (*metis con-def con-sym-ax ideE in-sourcesE in-sourcesI resid-ide-arr coinitialE src-ide src-resid<sub>WE</sub>*)

**lemma** *resid-src-arr* [*simp*]:  
**assumes** *arr f*  
**shows**  $\text{src } f \setminus f = \text{trg } f$   
**using** *assms*  
**by** (*simp add: con-imp-coinitial resid-ide(2)*)

```

lemma resid-arr-src [simp]:
assumes arr f
shows  $f \setminus \text{src } f = f$ 
  using assms
  by (simp add: resid-arr-ide)

```

**end**

### 2.1.5 Extensional RTS

An *extensional* RTS is an RTS in which all arrows have trivial congruence classes; that is, congruent arrows are equal.

```

locale extensional-rts = rts +
assumes extensionality:  $t \sim u \implies t = u$ 
begin

```

```

  sublocale weakly-extensional-rts
    using extensionality
    by unfold-locales auto

```

```

  lemma cong-char:
shows  $t \sim u \iff \text{arr } t \wedge t = u$ 
    by (metis arrI cong-reflexive prfx-implies-con extensionality)

```

**end**

### 2.1.6 Composites of Transitions

Residuation can be used to define a notion of composite of transitions. Composites are not unique, but they are unique up to congruence.

```

context rts
begin

```

```

  definition composite-of
where composite-of  $u \ t \ v \equiv u \lesssim v \wedge v \setminus u \sim t$ 

```

```

  lemma composite-ofI [intro]:
assumes  $u \lesssim v$  and  $v \setminus u \sim t$ 
shows composite-of  $u \ t \ v$ 
    using assms composite-of-def by blast

```

```

  lemma composite-ofE [elim]:
assumes composite-of  $u \ t \ v$ 
and  $\llbracket u \lesssim v; v \setminus u \sim t \rrbracket \implies T$ 
shows  $T$ 
    using assms composite-of-def by auto

```

```

  lemma arr-composite-of:

```

**assumes** *composite-of u t v*  
**shows** *arr v*  
**using** *assms*  
**by** (*meson composite-of-def con-implies-arr(2) prfx-implies-con*)

**lemma** *composite-of-unq-upto-cong:*  
**assumes** *composite-of u t v* **and** *composite-of u t v'*  
**shows**  $v \sim v'$   
**using** *assms cube ide-backward-stable prfx-transitive*  
**by** (*elim composite-ofE*) *metis*

**lemma** *composite-of-ide-arr:*  
**assumes** *ide a*  
**shows** *composite-of a t t*  $\longleftrightarrow$   $t \frown a$   
**using** *assms*  
**by** (*metis composite-of-def con-implies-arr(1) con-sym resid-arr-ide resid-ide-arr prfx-implies-con prfx-reflexive*)

**lemma** *composite-of-arr-ide:*  
**assumes** *ide b*  
**shows** *composite-of t b t*  $\longleftrightarrow$   $t \setminus t \frown b$   
**using** *assms*  
**by** (*metis arr-resid-iff-con composite-of-def ide-imp-con-iff-cong con-implies-arr(1) prfx-implies-con prfx-reflexive*)

**lemma** *composite-of-source-arr:*  
**assumes** *arr t* **and**  $a \in \text{sources } t$   
**shows** *composite-of a t t*  
**using** *assms composite-of-ide-arr sources-def* **by** *auto*

**lemma** *composite-of-arr-target:*  
**assumes** *arr t* **and**  $b \in \text{targets } t$   
**shows** *composite-of t b t*  
**by** (*metis arrE assms composite-of-arr-ide in-sourcesE sources-resid*)

**lemma** *composite-of-ide-self:*  
**assumes** *ide a*  
**shows** *composite-of a a a*  
**using** *assms composite-of-ide-arr* **by** *blast*

**lemma** *con-prfx-composite-of:*  
**assumes** *composite-of t u w*  
**shows**  $t \frown w$  **and**  $w \frown v \implies t \frown v$   
**using** *assms* **apply** *force*  
**using** *assms composite-of-def con-target prfx-implies-con resid-reflects-con con-sym*  
**by** *meson*

**lemma** *sources-composite-of:*

**assumes** *composite-of u t v*  
**shows** *sources v = sources u*  
**using** *assms*  
**by** (*meson arr-resid-iff-con composite-of-def con-imp-coinitial cong-implies-coinitial coinitial-iff*)

**lemma** *targets-composite-of:*  
**assumes** *composite-of u t v*  
**shows** *targets v = targets t*  
**proof** –  
**have** *targets t = targets (v \ u)*  
**using** *assms composite-of-def*  
**by** (*meson cong-implies-coterminal coterminal-iff*)  
**also have** *... = targets (u \ v)*  
**using** *assms targets-resid-sym con-prfx-composite-of* **by** *metis*  
**also have** *... = targets v*  
**using** *assms composite-of-def*  
**by** (*metis prfx-implies-con sources-resid ideE*)  
**finally show** *?thesis* **by** *auto*  
**qed**

**lemma** *resid-composite-of:*  
**assumes** *composite-of t u w* **and**  $w \frown v$   
**shows**  $v \setminus t \frown w \setminus t$   
**and**  $v \setminus t \frown u$   
**and**  $v \setminus w \sim (v \setminus t) \setminus u$   
**and** *composite-of (t \ v) (u \ (v \ t)) (w \ v)*  
**proof** –  
**show** *0: v \ t \frown w \ t*  
**using** *assms con-def*  
**by** (*metis con-target composite-ofE conE con-sym cube*)  
**show** *1: v \ w \sim (v \ t) \setminus u*  
**proof** –  
**have**  $v \setminus w = (v \setminus w) \setminus (t \setminus w)$   
**using** *assms composite-of-def*  
**by** (*metis (no-types, opaque-lifting) con-target con-sym resid-arr-ide*)  
**also have**  $... = (v \setminus t) \setminus (w \setminus t)$   
**using** *assms cube* **by** *metis*  
**also have**  $... \sim (v \setminus t) \setminus u$   
**using** *assms 0 cong-subst-right(2) [of w \ t u v \ t]* **by** *blast*  
**finally show** *?thesis* **by** *blast*  
**qed**  
**show** *2: v \ t \frown u*  
**using** *assms 1* **by** *force*  
**show** *composite-of (t \ v) (u \ (v \ t)) (w \ v)*  
**proof** (*unfold composite-of-def, intro conjI*)  
**show**  $t \setminus v \lesssim w \setminus v$   
**using** *assms cube con-target composite-of-def resid-ide-arr* **by** *metis*  
**show**  $(w \setminus v) \setminus (t \setminus v) \lesssim u \setminus (v \setminus t)$

**by** (*metis* *assms*(1) 2 *composite-ofE* *con-sym* *cong-subst-left*(2) *cube*)  
**thus**  $u \setminus (v \setminus t) \lesssim (w \setminus v) \setminus (t \setminus v)$   
**using** *assms*  
**by** (*metis* *composite-of-def* *con-implies-arr*(2) *cong-subst-left*(2)  
*prfx-implies-con* *arr-resid-iff-con* *cube*)  
**qed**  
**qed**

**lemma** *con-composite-of-iff*:  
**assumes** *composite-of* *t* *u* *v*  
**shows**  $w \frown v \longleftrightarrow w \setminus t \frown u$   
**by** (*meson* *arr-resid-iff-con* *assms* *composite-ofE* *con-def* *con-implies-arr*(1)  
*con-sym-ax* *cong-subst-right*(1) *resid-composite-of*(2) *resid-reflects-con*)

**definition** *composable*  
**where** *composable* *t* *u*  $\equiv \exists v. \text{composite-of } t \ u \ v$

**lemma** *composableD* [*dest*]:  
**assumes** *composable* *t* *u*  
**shows** *arr* *t* **and** *arr* *u* **and** *targets* *t* = *sources* *u*  
**using** *assms* *arr-composite-of* *arr-iff-has-source* *composable-def* *sources-composite-of*  
*arr-composite-of* *arr-iff-has-target* *composable-def* *targets-composite-of*  
**apply** *auto*[2]  
**by** (*metis* *assms* *composable-def* *composite-ofE* *con-prfx-composite-of*(1) *con-sym*  
*cong-implies-coinitial* *coinitial-iff* *sources-resid*)

**lemma** *composable-imp-seq*:  
**assumes** *composable* *t* *u*  
**shows** *seq* *t* *u*  
**using** *assms* **by** *blast*

**lemma** *composable-permute*:  
**shows** *composable* *t* ( $u \setminus t$ )  $\longleftrightarrow$  *composable* *u* ( $t \setminus u$ )  
**unfolding** *composable-def*  
**by** (*metis* *cube* *ide-backward-stable* *ide-imp-con-iff-cong* *prfx-implies-con*  
*composite-ofE* *composite-ofI*)

**lemma** *diamond-commutes-upto-cong*:  
**assumes** *composite-of* *t* ( $u \setminus t$ ) *v* **and** *composite-of* *u* ( $t \setminus u$ ) *v'*  
**shows**  $v \sim v'$   
**using** *assms* *cube* *ide-backward-stable* *prfx-transitive*  
**by** (*elim* *composite-ofE*) *metis*

**lemma** *bounded-imp-con*:  
**assumes** *composite-of* *t* *u* *v* **and** *composite-of* *t'* *u'* *v*  
**shows** *con* *t* *t'*  
**by** (*meson* *assms* *composite-of-def* *con-prfx-composite-of* *prfx-implies-con*  
*arr-resid-iff-con* *con-implies-arr*(2))

**lemma** *composite-of-cancel-left*:  
**assumes** *composite-of t u v* **and** *composite-of t u' v*  
**shows**  $u \sim u'$   
**using** *assms composite-of-def cong-transitive* **by** *blast*

**end**

## RTS with Composites

**locale** *rts-with-composites* = *rts* +  
**assumes** *has-composites: seq t u  $\implies$  composable t u*  
**begin**

**lemma** *composable-iff-seq*:  
**shows** *composable g f  $\longleftrightarrow$  seq g f*  
**using** *composable-imp-seq has-composites* **by** *blast*

**lemma** *composableI* [*intro*]:  
**assumes** *seq g f*  
**shows** *composable g f*  
**using** *assms composable-iff-seq* **by** *auto*

**lemma** *composableE* [*elim*]:  
**assumes** *composable g f* **and** *seq g f  $\implies$  T*  
**shows** *T*  
**using** *assms composable-iff-seq* **by** *blast*

**lemma** *obtains-composite-of*:  
**assumes** *seq g f*  
**obtains** *h* **where** *composite-of g f h*  
**using** *assms has-composites composable-def* **by** *blast*

**end**

### 2.1.7 Joins of Transitions

**context** *rts*  
**begin**

Transition  $v$  is a *join* of  $u$  and  $v$  when  $v$  is the diagonal of the square formed by  $u$ ,  $v$ , and their residuals. As was the case for composites, joins in an RTS are not unique, but they are unique up to congruence.

**definition** *join-of*  
**where** *join-of t u v  $\equiv$  composite-of t (u \ t) v  $\wedge$  composite-of u (t \ u) v*

**lemma** *join-ofI* [*intro*]:  
**assumes** *composite-of t (u \ t) v* **and** *composite-of u (t \ u) v*  
**shows** *join-of t u v*  
**using** *assms join-of-def* **by** *simp*

**lemma** *join-ofE [elim]*:  
**assumes** *join-of t u v*  
**and**  $\llbracket \text{composite-of } t (u \setminus t) v; \text{ composite-of } u (t \setminus u) v \rrbracket \implies T$   
**shows**  $T$   
**using** *assms join-of-def by simp*

**definition** *joinable*  
**where** *joinable t u*  $\equiv \exists v. \text{join-of } t u v$

**lemma** *joinable-implies-con*:  
**assumes** *joinable t u*  
**shows**  $t \frown u$   
**by** (*meson assms bounded-imp-con join-of-def joinable-def*)

**lemma** *joinable-implies-coinitial*:  
**assumes** *joinable t u*  
**shows** *coinitial t u*  
**using** *assms*  
**by** (*simp add: con-imp-coinitial joinable-implies-con*)

**lemma** *joinable-iff-composable*:  
**shows** *joinable t u*  $\longleftrightarrow$  *composable t (u \setminus t)*  
**proof**  
**show** *joinable t u*  $\implies$  *composable t (u \setminus t)*  
**unfolding** *joinable-def join-of-def composable-def by auto*  
**show** *composable t (u \setminus t)*  $\implies$  *joinable t u*  
**proof** –  
**assume**  $1: \text{composable } t (u \setminus t)$   
**obtain**  $v$  **where**  $v: \text{composite-of } t (u \setminus t) v$   
**using**  $1$  *composable-def by blast*  
**have** *composite-of u (t \setminus u) v*  
**proof**  
**show**  $u \lesssim v$   
**by** (*metis v composite-ofE cube ide-backward-stable*)  
**show**  $v \setminus u \sim t \setminus u$   
**by** (*metis v \langle u \lesssim v \rangle coinitial-ide-are-cong composite-ofE con-imp-coinitial cube prfx-implies-con*)  
**qed**  
**thus** *joinable t u*  
**using**  $v$  *joinable-def by auto*  
**qed**  
**qed**

**lemma** *join-of-un-upto-cong*:  
**assumes** *join-of t u v* **and** *join-of t u v'*  
**shows**  $v \sim v'$   
**using** *assms join-of-def composite-of-unq-upto-cong by auto*

**lemma** *join-of-symmetric*:  
**assumes** *join-of t u v*  
**shows** *join-of u t v*  
**using** *assms join-of-def* **by** *simp*

**lemma** *join-of-arr-self*:  
**assumes** *arr t*  
**shows** *join-of t t t*  
**by** (*meson assms composite-of-arr-ide ideE join-of-def prfx-reflexive*)

**lemma** *join-of-arr-src*:  
**assumes** *arr t* **and**  $a \in \text{sources } t$   
**shows** *join-of a t t* **and** *join-of t a t*  
**proof** –  
**show** *join-of a t t*  
**by** (*meson assms composite-of-arr-target composite-of-def composite-of-source-arr join-of-def prfx-transitive resid-source-in-targets*)  
**thus** *join-of t a t*  
**using** *join-of-symmetric* **by** *blast*  
**qed**

**lemma** *sources-join-of*:  
**assumes** *join-of t u v*  
**shows**  $\text{sources } t = \text{sources } v$  **and**  $\text{sources } u = \text{sources } v$   
**using** *assms join-of-def sources-composite-of* **by** *blast+*

**lemma** *targets-join-of*:  
**assumes** *join-of t u v*  
**shows**  $\text{targets } (t \setminus u) = \text{targets } v$  **and**  $\text{targets } (u \setminus t) = \text{targets } v$   
**using** *assms join-of-def targets-composite-of* **by** *blast+*

**lemma** *join-of-resid*:  
**assumes** *join-of t u w* **and** *con v w*  
**shows** *join-of (t \ v) (u \ v) (w \ v)*  
**using** *assms con-sym cube join-of-def resid-composite-of(4)* **by** *fastforce*

**lemma** *con-with-join-of-iff*:  
**assumes** *join-of t u w*  
**shows**  $u \frown v \wedge v \setminus u \frown t \setminus u \implies w \frown v$   
**and**  $w \frown v \implies t \frown v \wedge v \setminus t \frown u \setminus t$   
**proof** –  
**have** \*:  $t \frown v \wedge v \setminus t \frown u \setminus t \iff u \frown v \wedge v \setminus u \frown t \setminus u$   
**by** (*metis arr-resid-iff-con con-implies-arr(1) con-sym cube*)  
**show**  $u \frown v \wedge v \setminus u \frown t \setminus u \implies w \frown v$   
**by** (*meson assms con-composite-of-iff con-sym join-of-def*)  
**show**  $w \frown v \implies t \frown v \wedge v \setminus t \frown u \setminus t$   
**by** (*meson assms con-prfx-composite-of join-of-def resid-composite-of(2)*)  
**qed**

**lemma** *join-of-respects-cong-left*:  
**assumes** *join-of t u v* **and** *cong t t'*  
**shows** *join-of t' u v*  
**using** *assms prfx-transitive cong-subst-left(2) cong-subst-right(2)*  
**apply** (*elim join-ofE composite-ofE, intro join-ofI composite-ofI*)  
**by** (*meson con-sym con-with-join-of-iff(2) prfx-implies-con*)+

**lemma** *join-of-respects-cong-right*:  
**assumes** *join-of t u v* **and** *cong u u'*  
**shows** *join-of t u' v*  
**using** *assms join-of-respects-cong-left join-of-symmetric*  
**by** *meson*

**end**

## RTS with Joins

**locale** *rts-with-joins* = *rts* +  
**assumes** *has-joins: t  $\frown$  u  $\implies$  joinable t u*

### 2.1.8 Joins and Composites in a Weakly Extensional RTS

**context** *weakly-extensional-rts*  
**begin**

**lemma** *src-composite-of*:  
**assumes** *composite-of u t v*  
**shows** *src v = src u*  
**using** *assms*  
**by** (*metis con-imp-eq-src con-prfx-composite-of(1)*)

**lemma** *trg-composite-of*:  
**assumes** *composite-of u t v*  
**shows** *trg v = trg t*  
**by** (*metis arr-composite-of arr-has-un-target arr-iff-has-target assms targets-composite-of trg-in-targets*)

**lemma** *src-join-of*:  
**assumes** *join-of t u v*  
**shows** *src t = src v* **and** *src u = src v*  
**by** (*metis assms join-ofE src-composite-of*)+

**lemma** *trg-join-of*:  
**assumes** *join-of t u v*  
**shows** *trg (t  $\setminus$  u) = trg v* **and** *trg (u  $\setminus$  t) = trg v*  
**by** (*metis assms join-of-def trg-composite-of*)+

**end**

## 2.1.9 Joins and Composites in an Extensional RTS

**context** *extensional-rts*  
**begin**

**lemma** *composite-of-unique*:  
**assumes** *composite-of t u v* **and** *composite-of t u v'*  
**shows**  $v = v'$   
**using** *assms composite-of-unq-upto-cong extensionality* **by** *fastforce*

**lemma** *divisors-of-ide*:  
**assumes** *composite-of t u v* **and** *ide v*  
**shows** *ide t* **and** *ide u*  
**proof** –  
**show** *ide t*  
**using** *assms ide-backward-stable* **by** *blast*  
**show** *ide u*  
**by** (*metis assms(1–2) composite-ofE con-ide-are-eq con-prfx-composite-of(1)*  
*ide-backward-stable*)  
**qed**

Here we define composition of transitions. Note that we compose transitions in diagram order, rather than in the order used for function composition. This may eventually lead to confusion, but here (unlike in the case of a category) transitions are typically not functions, so we don't have the constraint of having to conform to the order of function application and composition, and diagram order seems more natural.

**definition** *comp* (**infixr**  $\langle \cdot \rangle$  55)  
**where**  $t \cdot u \equiv$  *if composable t u then THE v. composite-of t u v else null*

**lemma** *comp-is-composite-of*:  
**shows** *composable t u*  $\implies$  *composite-of t u (t · u)*  
**and** *composite-of t u v*  $\implies$   $t \cdot u = v$   
**proof** –  
**show** *composable t u*  $\implies$  *composite-of t u (t · u)*  
**using** *comp-def composite-of-unique the1I2 [of composite-of t u composite-of t u]*  
*composable-def*  
**by** *metis*  
**thus** *composite-of t u v*  $\implies$   $t \cdot u = v$   
**using** *composite-of-unique composable-def* **by** *auto*  
**qed**

**lemma** *comp-null [simp]*:  
**shows**  $null \cdot t = null$  **and**  $t \cdot null = null$   
**by** (*meson composableD not-arr-null comp-def*)**+**

**lemma** *composable-iff-arr-comp*:  
**shows** *composable t u*  $\iff$  *arr (t · u)*  
**by** (*metis arr-composite-of comp-is-composite-of(2) composable-def comp-def not-arr-null*)

**lemma** *composable-iff-comp-not-null*:  
**shows**  $\text{composable } t \ u \longleftrightarrow t \cdot u \neq \text{null}$   
**by** (*metis composable-iff-arr-comp comp-def not-arr-null*)

**lemma** *comp-src-arr* [*simp*]:  
**assumes**  $\text{arr } t$  **and**  $\text{src } t = a$   
**shows**  $a \cdot t = t$   
**using** *assms comp-is-composite-of(2) composite-of-source-arr src-in-sources* **by** *blast*

**lemma** *comp-arr-trg* [*simp*]:  
**assumes**  $\text{arr } t$  **and**  $\text{trg } t = b$   
**shows**  $t \cdot b = t$   
**using** *assms comp-is-composite-of(2) composite-of-arr-target trg-in-targets* **by** *blast*

**lemma** *comp-ide-self*:  
**assumes** *ide a*  
**shows**  $a \cdot a = a$   
**using** *assms comp-is-composite-of(2) composite-of-ide-self* **by** *fastforce*

**lemma** *arr-comp* [*intro, simp*]:  
**assumes** *composable t u*  
**shows**  $\text{arr } (t \cdot u)$   
**using** *assms composable-iff-arr-comp* **by** *blast*

**lemma** *trg-comp* [*simp*]:  
**assumes** *composable t u*  
**shows**  $\text{trg } (t \cdot u) = \text{trg } u$   
**by** (*metis arr-has-un-target assms comp-is-composite-of(2) composable-def composable-imp-seq arr-iff-has-target seq-def targets-composite-of trg-in-targets*)

**lemma** *src-comp* [*simp*]:  
**assumes** *composable t u*  
**shows**  $\text{src } (t \cdot u) = \text{src } t$   
**using** *assms comp-is-composite-of arr-iff-has-source sources-composite-of src-def composable-def*  
**by** *auto*

**lemma** *con-comp-iff*:  
**shows**  $w \frown t \cdot u \longleftrightarrow \text{composable } t \ u \wedge w \setminus t \frown u$   
**by** (*meson comp-is-composite-of(1) composable-iff-arr-comp con-composite-of-iff con-implies-arr(2)*)

**lemma** *con-compI* [*intro*]:  
**assumes** *composable t u* **and**  $w \setminus t \frown u$   
**shows**  $w \frown t \cdot u$  **and**  $t \cdot u \frown w$   
**using** *assms con-comp-iff con-sym* **by** *blast+*

**lemma** *resid-comp*:  
**assumes**  $t \cdot u \frown w$

**shows**  $w \setminus (t \cdot u) = (w \setminus t) \setminus u$   
**and**  $(t \cdot u) \setminus w = (t \setminus w) \cdot (u \setminus (w \setminus t))$   
**proof** –  
   **have** 1: *composable*  $t$   $u$   
     **using** *assms composable-iff-comp-not-null* **by** *force*  
   **show**  $w \setminus (t \cdot u) = (w \setminus t) \setminus u$   
     **using** 1  
   **by** (*meson assms cong-char composable-def resid-composite-of(3) comp-is-composite-of(1)*)  
   **show**  $(t \cdot u) \setminus w = (t \setminus w) \cdot (u \setminus (w \setminus t))$   
     **using** *assms 1 composable-def comp-is-composite-of(2) resid-composite-of*  
     **by** *metis*  
**qed**

**lemma** *prfx-decomp*:  
**assumes**  $t \lesssim u$   
**shows**  $t \cdot (u \setminus t) = u$   
   **by** (*meson assms arr-resid-iff-con comp-is-composite-of(2) composite-of-def con-sym*  
     *cong-reflexive prfx-implies-con*)

**lemma** *prfx-comp*:  
**assumes** *arr*  $u$  **and**  $t \cdot v = u$   
**shows**  $t \lesssim u$   
   **by** (*metis assms comp-is-composite-of(2) composable-def composable-iff-arr-comp*  
     *composite-of-def*)

**lemma** *comp-eqI*:  
**assumes**  $t \lesssim v$  **and**  $u = v \setminus t$   
**shows**  $t \cdot u = v$   
   **by** (*metis assms prfx-decomp*)

**lemma** *comp-assoc*:  
**assumes** *composable*  $(t \cdot u)$   $v$   
**shows**  $t \cdot (u \cdot v) = (t \cdot u) \cdot v$   
**proof** –  
   **have** 1:  $t \lesssim (t \cdot u) \cdot v$   
     **by** (*meson assms composable-iff-arr-comp composableD prfx-comp*  
       *prfx-transitive*)  
   **moreover** **have**  $((t \cdot u) \cdot v) \setminus t = u \cdot v$   
   **proof** –  
     **have**  $((t \cdot u) \cdot v) \setminus t = ((t \cdot u) \setminus t) \cdot (v \setminus (t \setminus (t \cdot u)))$   
       **by** (*meson assms calculation con-sym prfx-implies-con resid-comp(2)*)  
     **also** **have**  $\dots = u \cdot v$   
     **proof** –  
       **have** 2:  $(t \cdot u) \setminus t = u$   
         **by** (*metis assms comp-is-composite-of(2) composable-def composable-iff-arr-comp*  
           *composable-imp-seq composite-of-def extensionality seqE*)  
       **moreover** **have**  $v \setminus (t \setminus (t \cdot u)) = v$   
         **using** *assms*  
         **by** (*meson 1 con-comp-iff con-sym composable-imp-seq resid-arr-ide*)

```

      prfx-implies-con prfx-comp seqE)
    ultimately show ?thesis by simp
  qed
  finally show ?thesis by blast
  qed
  ultimately show  $t \cdot (u \cdot v) = (t \cdot u) \cdot v$ 
    by (metis comp-eqI)
  qed

```

We note the following assymetry:  $\text{composable } (t \cdot u) v \implies \text{composable } u v$  is true, but  $\text{composable } t (u \cdot v) \implies \text{composable } t u$  is not.

```

lemma comp-cancel-left:
assumes  $\text{arr } (t \cdot u)$  and  $t \cdot u = t \cdot v$ 
shows  $u = v$ 
  using assms
  by (metis composable-def composable-iff-arr-comp composite-of-cancel-left extensionality comp-is-composite-of(2))

```

```

lemma comp-resid-prfx [simp]:
assumes  $\text{arr } (t \cdot u)$ 
shows  $(t \cdot u) \setminus t = u$ 
  using assms
  by (metis comp-cancel-left comp-eqI prfx-comp)

```

```

lemma bounded-imp-conE:
assumes  $t \cdot u \sim t' \cdot u'$ 
shows  $t \frown t'$ 
  by (metis arr-resid-iff-con assms con-comp-iff con-implies-arr(2) prfx-implies-con con-sym)

```

```

lemma join-of-unique:
assumes join-of  $t u v$  and join-of  $t u v'$ 
shows  $v = v'$ 
  using assms join-of-def composite-of-unique by blast

```

```

definition join (infix  $\langle \sqcup \rangle$  52)
where  $t \sqcup u \equiv \text{if joinable } t u \text{ then THE } v. \text{join-of } t u v \text{ else null}$ 

```

```

lemma join-is-join-of:
assumes joinable  $t u$ 
shows join-of  $t u (t \sqcup u)$ 
  using assms joinable-def join-def join-of-unique the1I2 [of join-of t u join-of t u]
  by force

```

```

lemma joinable-iff-arr-join:
shows joinable  $t u \iff \text{arr } (t \sqcup u)$ 
  by (metis cong-char join-is-join-of join-of-un-upto-cong not-arr-null join-def)

```

**lemma** *joinable-iff-join-not-null*:  
**shows**  $joinable\ t\ u \longleftrightarrow t \sqcup u \neq null$   
**by** (*metis join-def joinable-iff-arr-join not-arr-null*)

**lemma** *join-sym*:  
**shows**  $t \sqcup u = u \sqcup t$   
**by** (*metis join-def join-of-unique join-is-join-of join-of-symmetric joinable-def*)

**lemma** *src-join*:  
**assumes** *joinable t u*  
**shows**  $src\ (t \sqcup u) = src\ t$   
**using** *assms*  
**by** (*metis con-imp-eq-src con-prfx-composite-of(1) join-is-join-of join-of-def*)

**lemma** *trg-join*:  
**assumes** *joinable t u*  
**shows**  $trg\ (t \sqcup u) = trg\ (t \setminus u)$   
**using** *assms*  
**by** (*metis arr-resid-iff-con join-is-join-of joinable-iff-arr-join joinable-implies-con in-targetsE src-eqI targets-join-of(1) trg-in-targets*)

**lemma** *resid-join<sub>E</sub> [simp]*:  
**assumes** *joinable t u* **and**  $v \frown t \sqcup u$   
**shows**  $v \setminus (t \sqcup u) = (v \setminus u) \setminus (t \setminus u)$   
**and**  $v \setminus (t \sqcup u) = (v \setminus t) \setminus (u \setminus t)$   
**and**  $(t \sqcup u) \setminus v = (t \setminus v) \sqcup (u \setminus v)$   
**proof** –  
**show**  $1: v \setminus (t \sqcup u) = (v \setminus u) \setminus (t \setminus u)$   
**by** (*meson assms con-sym join-of-def resid-composite-of(3) extensionality join-is-join-of*)  
**show**  $v \setminus (t \sqcup u) = (v \setminus t) \setminus (u \setminus t)$   
**by** (*metis 1 cube*)  
**show**  $(t \sqcup u) \setminus v = (t \setminus v) \sqcup (u \setminus v)$   
**using** *assms joinable-def join-of-resid join-is-join-of extensionality*  
**by** (*meson join-of-unique*)  
**qed**

**lemma** *join-eqI*:  
**assumes**  $t \lesssim v$  **and**  $u \lesssim v$  **and**  $v \setminus u = t \setminus u$  **and**  $v \setminus t = u \setminus t$   
**shows**  $t \sqcup u = v$   
**using** *assms composite-of-def cube ideE join-of-def joinable-def join-of-unique join-is-join-of trg-def*  
**by** *metis*

**lemma** *comp-join*:  
**assumes** *joinable (t · u) (t · u')*  
**shows** *composable t (u · u')*  
**and**  $t \cdot (u \sqcup u') = t \cdot u \sqcup t \cdot u'$   
**proof** –

**have**  $t \lesssim t \cdot u \sqcup t \cdot u'$   
**using** *assms*  
**by** (*metis composable-def composite-of-def join-of-def join-is-join-of*  
*joinable-implies-con prfx-transitive comp-is-composite-of(2) con-comp-iff*)  
**moreover have**  $(t \cdot u \sqcup t \cdot u') \setminus t = u \sqcup u'$   
**by** (*metis arr-resid-iff-con assms calculation comp-resid-prfx con-implies-arr(2)*  
*joinable-implies-con resid-join<sub>E</sub>(3) con-implies-arr(1) ide-implies-arr*)  
**ultimately show**  $t \cdot (u \sqcup u') = t \cdot u \sqcup t \cdot u'$   
**by** (*metis comp-eqI*)  
**thus composable**  $t (u \sqcup u')$   
**by** (*metis assms joinable-iff-join-not-null comp-def*)  
**qed**

**lemma** *join-src*:  
**assumes** *arr t*  
**shows**  $\text{src } t \sqcup t = t$   
**using** *assms joinable-def join-of-arr-src join-is-join-of join-of-unique src-in-sources*  
**by** *meson*

**lemma** *join-arr-self*:  
**assumes** *arr t*  
**shows**  $t \sqcup t = t$   
**using** *assms joinable-def join-of-arr-self join-is-join-of join-of-unique* **by** *blast*

**lemma** *arr-prfx-join-self*:  
**assumes** *joinable t u*  
**shows**  $t \lesssim t \sqcup u$   
**using** *assms*  
**by** (*meson composite-of-def join-is-join-of join-of-def*)

**lemma** *con-prfx*:  
**shows**  $\llbracket t \frown u; v \lesssim u \rrbracket \implies t \frown v$   
**and**  $\llbracket t \frown u; v \lesssim t \rrbracket \implies v \frown u$   
**apply** (*metis arr-resid con-arr-src(1) ide-iff-src-self prfx-implies-con resid-reflects-con*  
*src-resid<sub>WE</sub>*)  
**by** (*metis arr-resid-iff-con comp-eqI con-comp-iff con-implies-arr(1) con-sym*)

**lemma** *join-prfx*:  
**assumes**  $t \lesssim u$   
**shows**  $t \sqcup u = u$  **and**  $u \sqcup t = u$   
**proof** –  
**show**  $t \sqcup u = u$   
**using** *assms*  
**by** (*metis (no-types, lifting) join-eqI ide-iff-src-self ide-implies-arr resid-arr-self*  
*prfx-implies-con src-resid<sub>WE</sub>*)  
**thus**  $u \sqcup t = u$   
**by** (*metis join-sym*)  
**qed**

**lemma** *con-with-join-if* [*intro, simp*]:  
**assumes** *joinable t u* **and**  $u \frown v$  **and**  $v \setminus u \frown t \setminus u$   
**shows**  $t \sqcup u \frown v$   
**and**  $v \frown t \sqcup u$   
**proof** –  
    **show**  $t \sqcup u \frown v$   
        **using** *assms con-with-join-of-iff* [*of t u join t u v*] *join-is-join-of* **by** *simp*  
    **thus**  $v \frown t \sqcup u$   
        **using** *assms con-sym* **by** *blast*  
**qed**

**lemma** *join-assocE*:  
**assumes** *arr*  $((t \sqcup u) \sqcup v)$  **and** *arr*  $(t \sqcup (u \sqcup v))$   
**shows**  $(t \sqcup u) \sqcup v = t \sqcup (u \sqcup v)$   
**proof** (*intro join-eqI*)  
    **have** *tu*: *joinable t u*  
        **by** (*metis arr-src-iff-arr assms(1) joinable-iff-arr-join src-join*)  
    **have** *uv*: *joinable u v*  
        **by** (*metis assms(2) joinable-iff-arr-join joinable-iff-join-not-null joinable-implies-con not-con-null(2)*)  
    **have** *tu-v*: *joinable (t \sqcup u) v*  
        **by** (*simp add: assms(1) joinable-iff-arr-join*)  
    **have** *t-uv*: *joinable t (u \sqcup v)*  
        **by** (*simp add: assms(2) joinable-iff-arr-join*)  
    **show**  $0$ :  $t \sqcup u \lesssim t \sqcup (u \sqcup v)$   
**proof** –  
    **have**  $(t \sqcup u) \setminus (t \sqcup (u \sqcup v)) = ((u \setminus t) \setminus (u \setminus t)) \setminus ((v \setminus t) \setminus (u \setminus t))$   
**proof** –  
    **have**  $(t \sqcup u) \setminus (t \sqcup (u \sqcup v)) = ((t \sqcup u) \setminus t) \setminus ((u \sqcup v) \setminus t)$   
        **by** (*metis t-uv tu arr-prfx-join-self conI con-with-join-if(2) join-sym joinable-iff-join-not-null not-ide-null resid-joinE(2)*)  
    **also have**  $\dots = (t \setminus t \sqcup u \setminus t) \setminus ((u \sqcup v) \setminus t)$   
        **by** (*simp add: tu con-sym joinable-implies-con*)  
    **also have**  $\dots = (t \setminus t \sqcup u \setminus t) \setminus (u \setminus t \sqcup v \setminus t)$   
        **by** (*simp add: t-uv uv joinable-implies-con*)  
    **also have**  $\dots = (u \setminus t) \setminus \text{join } (u \setminus t) (v \setminus t)$   
        **by** (*metis tu con-implies-arr(1) cong-subst-left(2) cube join-eqI join-sym joinable-iff-join-not-null joinable-implies-con prfx-reflexive trg-def trg-join*)  
    **also have**  $\dots = ((u \setminus t) \setminus (u \setminus t)) \setminus ((v \setminus t) \setminus (u \setminus t))$   
**proof** –  
    **have**  $1$ : *joinable (u \setminus t) (v \setminus t)*  
        **by** (*metis t-uv uv con-sym joinable-iff-join-not-null joinable-implies-con resid-joinE(3) conE*)  
    **moreover have**  $u \setminus t \frown u \setminus t \sqcup v \setminus t$   
        **using** *arr-prfx-join-self 1 prfx-implies-con* **by** *blast*  
    **ultimately show** *?thesis*  
        **using** *resid-joinE(2)* [*of u \setminus t v \setminus t u \setminus t*] **by** *blast*  
**qed**  
**finally show** *?thesis* **by** *blast*

**qed**  
**moreover have** *ide ...*  
by (*metis tu-v tu arr-resid-iff-con con-sym cube joinable-implies-con prfx-reflexive resid-join<sub>E</sub>(2)*)  
**ultimately show** *?thesis by simp*  
**qed**  
**show**  $1: v \lesssim t \sqcup (u \sqcup v)$   
by (*metis arr-prfx-join-self join-sym joinable-iff-join-not-null prfx-transitive t-uv uv*)  
**show**  $(t \sqcup (u \sqcup v)) \setminus v = (t \sqcup u) \setminus v$   
**proof** –  
have  $(t \sqcup (u \sqcup v)) \setminus v = t \setminus v \sqcup (u \sqcup v) \setminus v$   
by (*metis 1 assms(2) join-def not-arr-null resid-join<sub>E</sub>(3) prfx-implies-con*)  
**also have**  $\dots = t \setminus v \sqcup (u \setminus v \sqcup v \setminus v)$   
by (*metis 1 conE conI con-sym join-def resid-join<sub>E</sub>(1) resid-join<sub>E</sub>(3) null-is-zero(2) prfx-implies-con*)  
**also have**  $\dots = t \setminus v \sqcup u \setminus v$   
by (*metis arr-resid-iff-con con-sym cube cong-char join-prfx(2) joinable-implies-con uv*)  
**also have**  $\dots = (t \sqcup u) \setminus v$   
by (*metis 0 1 con-implies-arr(1) con-prfx(1) joinable-iff-arr-join resid-join<sub>E</sub>(3) prfx-implies-con*)  
**finally show** *?thesis by blast*  
**qed**  
**show**  $(t \sqcup (u \sqcup v)) \setminus (t \sqcup u) = v \setminus (t \sqcup u)$   
**proof** –  
have  $2: (t \sqcup (u \sqcup v)) \setminus (t \sqcup u) = t \setminus (t \sqcup u) \sqcup (u \sqcup v) \setminus (t \sqcup u)$   
by (*metis 0 assms(2) join-def not-arr-null resid-join<sub>E</sub>(3) prfx-implies-con*)  
**also have**  $3: \dots = (t \setminus t) \setminus (u \setminus t) \sqcup (u \sqcup v) \setminus (t \sqcup u)$   
by (*metis tu arr-prfx-join-self prfx-implies-con resid-join<sub>E</sub>(2)*)  
**also have**  $4: \dots = (u \sqcup v) \setminus (t \sqcup u)$   
**proof** –  
have  $(t \setminus t) \setminus (u \setminus t) = \text{src}((u \sqcup v) \setminus (t \sqcup u))$   
**using** *src-resid<sub>WE</sub> trg-join*  
by (*metis (full-types) t-uv tu 0 arr-resid-iff-con con-implies-arr(1) con-sym cube prfx-implies-con resid-join<sub>E</sub>(1) trg-def*)  
**thus** *?thesis*  
by (*metis tu arr-prfx-join-self conE join-src prfx-implies-con resid-join<sub>E</sub>(2) src-def*)  
**qed**  
**also have**  $\dots = u \setminus (t \sqcup u) \sqcup v \setminus (t \sqcup u)$   
by (*metis 0 2 3 4 uv conI con-sym-ax not-ide-null resid-join<sub>E</sub>(3)*)  
**also have**  $\dots = (u \setminus u) \setminus (t \setminus u) \sqcup v \setminus (t \sqcup u)$   
by (*metis tu arr-prfx-join-self join-sym joinable-iff-join-not-null prfx-implies-con resid-join<sub>E</sub>(1)*)  
**also have**  $\dots = v \setminus (t \sqcup u)$   
**proof** –  
have  $(u \setminus u) \setminus (t \setminus u) = \text{src}(v \setminus (t \sqcup u))$   
by (*metis tu-v tu con-sym cube joinable-implies-con src-resid<sub>WE</sub> trg-def trg-join apex-sym*)  
**thus** *?thesis*  
**using** *tu-v arr-resid-iff-con con-sym join-src joinable-implies-con*

by *presburger*  
 qed  
 finally show *?thesis* by *blast*  
 qed  
 qed

**lemma** *join-prfx-monotone*:

**assumes**  $t \lesssim u$  and  $u \sqcup v \frown t \sqcup v$

**shows**  $t \sqcup v \lesssim u \sqcup v$

**proof** –

have  $(t \sqcup v) \setminus (u \sqcup v) = (t \setminus u) \setminus (v \setminus u)$

**proof** –

have  $(t \sqcup v) \setminus (u \sqcup v) = t \setminus (u \sqcup v) \sqcup v \setminus (u \sqcup v)$

using *assms join-sym resid-join<sub>E</sub>(3)* [*of t v join u v*] *joinable-iff-join-not-null*

by *fastforce*

also have  $\dots = (t \setminus u) \setminus (v \setminus u) \sqcup (v \setminus u) \setminus (v \setminus u)$

by (*metis* *(full-types) assms(2) conE conI joinable-iff-join-not-null null-is-zero(1)*)  
*resid-join<sub>E</sub>(1-2) con-sym-ax*)

also have  $\dots = (t \setminus u) \setminus (v \setminus u) \sqcup \text{trg } (v \setminus u)$

using *trg-def* by *fastforce*

also have  $\dots = (t \setminus u) \setminus (v \setminus u) \sqcup \text{src } ((t \setminus u) \setminus (v \setminus u))$

by (*metis* *assms(1-2) con-implies-arr(1) con-target joinable-iff-arr-join*)  
*joinable-implies-con src-resid<sub>WE</sub>*)

also have  $\dots = (t \setminus u) \setminus (v \setminus u)$

by (*metis* *arr-resid-iff-con assms(2) con-implies-arr(1) con-sym join-def*)  
*join-src join-sym not-arr-null resid-join<sub>E</sub>(2)*)

finally show *?thesis* by *blast*

qed

moreover have *ide* ...

by (*metis* *arr-resid-iff-con assms(1-2) calculation con-sym resid-ide-arr*)

ultimately show *?thesis* by *presburger*

qed

**lemma** *join-eqI'*:

**assumes**  $t \lesssim v$  and  $u \lesssim v$  and  $v \setminus u = t \setminus u$  and  $v \setminus t = u \setminus t$

**shows**  $v = t \sqcup u$

using *assms composite-of-def cube ideE join-of-def joinable-def join-of-unique*  
*join-is-join-of trg-def*

by *metis*

We note that it is not the case that the existence of either of  $t \sqcup (u \sqcup v)$  or  $(t \sqcup u) \sqcup v$  implies that of the other. For example, if  $(t \sqcup u) \sqcup v \neq \text{null}$ , then it is not necessarily the case that  $u \sqcup v \neq \text{null}$ .

**lemma** *join-expansion*:

**assumes** *joinable t u*

**shows**  $t \sqcup u = t \cdot (u \setminus t)$  and *seq t (u \setminus t)*

apply (*metis* *assms comp-is-composite-of(2) join-is-join-of join-of-def*)

using *assms joinable-iff-composable* by *auto*

**lemma** *join3-expansion*:  
**assumes** *joinable*  $(t \sqcup u)$   $v$   
**shows**  $(t \sqcup u) \sqcup v = (t \cdot (u \setminus t)) \cdot ((v \setminus t) \setminus (u \setminus t))$   
**by** (*metis* *assms* *con-implies-arr*(1) *join-expansion*(1) *joinable-iff-arr-join*  
*joinable-implies-con* *resid-comp*(1))

**lemma** *join-comp*:  
**assumes** *joinable*  $(t \cdot u)$   $v$   
**shows**  $(t \cdot u) \sqcup v = t \cdot (v \setminus t) \cdot (u \setminus (v \setminus t))$   
**by** (*metis* *assms* *composable-iff-comp-not-null* *extensional-rts.comp-assoc*  
*extensional-rts-axioms* *join-expansion*(1) *join-sym* *joinable-iff-composable*  
*joinable-iff-join-not-null* *joinable-implies-con* *resid-iff-comp*(1))

**end**

## Extensional RTS with Joins

**locale** *extensional-rts-with-joins* =  
*rts-with-joins* +  
*extensional-rts*

**begin**

**lemma** *joinable-iff-con* [*iff*]:  
**shows** *joinable*  $t$   $u \iff t \frown u$   
**by** (*meson* *has-joins* *joinable-implies-con*)

**lemma** *joinableE* [*elim*]:  
**assumes** *joinable*  $t$   $u$  **and**  $t \frown u \implies T$   
**shows**  $T$   
**using** *assms* *joinable-iff-con* **by** *blast*

**lemma** *src-join<sub>EJ</sub>* [*simp*]:  
**assumes**  $t \frown u$   
**shows** *src*  $(t \sqcup u) = \text{src } t$   
**using** *assms*  
**by** (*meson* *has-joins* *src-join*)

**lemma** *trg-join<sub>EJ</sub>*:  
**assumes**  $t \frown u$   
**shows** *trg*  $(t \sqcup u) = \text{trg } (t \setminus u)$   
**using** *assms*  
**by** (*meson* *has-joins* *trg-join*)

**lemma** *resid-join<sub>EJ</sub>* [*simp*]:  
**assumes**  $t \frown u$  **and**  $v \frown t \sqcup u$   
**shows**  $v \setminus (t \sqcup u) = (v \setminus t) \setminus (u \setminus t)$   
**and**  $(t \sqcup u) \setminus v = (t \setminus v) \sqcup (u \setminus v)$   
**using** *assms* *has-joins* *resid-join<sub>E</sub>* [*of t u v*] **by** *blast+*

**lemma** *join-assoc*:

**shows**  $t \sqcup (u \sqcup v) = (t \sqcup u) \sqcup v$

**proof** –

**have** \*:  $\bigwedge t u v. \text{con } (t \sqcup u) v \implies t \sqcup (u \sqcup v) = (t \sqcup u) \sqcup v$

**proof** –

**fix**  $t u v$

**assume**  $1: \text{con } (t \sqcup u) v$

**have**  $vt-ut: v \setminus t \frown u \setminus t$

**using**  $1$

**by** (*metis con-with-join-of-iff(2) join-def join-is-join-of not-con-null(1)*)

**have**  $tv-uv: t \setminus v \frown u \setminus v$

**using** *vt-ut cube con-sym*

**by** (*metis arr-resid-iff-con*)

**have**  $2: (t \sqcup u) \sqcup v = (t \cdot (u \setminus t)) \cdot (v \setminus (t \cdot (u \setminus t)))$

**using**  $1$

**by** (*metis comp-is-composite-of(2) con-implies-arr(1) has-joins join-is-join-of join-of-def joinable-iff-arr-join*)

**also have**  $\dots = t \cdot ((u \setminus t) \cdot (v \setminus (t \cdot (u \setminus t))))$

**using**  $1$

**by** (*metis calculation has-joins joinable-iff-join-not-null comp-assoc comp-def*)

**also have**  $\dots = t \cdot ((u \setminus t) \cdot ((v \setminus t) \setminus (u \setminus t)))$

**using**  $1$

**by** (*metis 2 comp-null(2) con-compI(2) con-comp-iff has-joins resid-comp(1) conI joinable-iff-join-not-null*)

**also have**  $\dots = t \cdot ((v \setminus t) \sqcup (u \setminus t))$

**by** (*metis vt-ut comp-is-composite-of(2) has-joins join-of-def join-is-join-of*)

**also have**  $\dots = t \cdot ((u \setminus t) \sqcup (v \setminus t))$

**using** *join-sym by metis*

**also have**  $\dots = t \cdot ((u \sqcup v) \setminus t)$

**by** (*metis tv-uv vt-ut con-implies-arr(2) con-sym con-with-join-of-iff(1) has-joins join-is-join-of arr-resid-iff-con resid-join<sub>E</sub>(3)*)

**also have**  $\dots = t \sqcup (u \sqcup v)$

**by** (*metis comp-is-composite-of(2) comp-null(2) conI has-joins join-is-join-of join-of-def joinable-iff-join-not-null*)

**finally show**  $t \sqcup (u \sqcup v) = (t \sqcup u) \sqcup v$

**by** *simp*

**qed**

**thus** *?thesis*

**by** (*metis (full-types) has-joins joinable-iff-join-not-null joinable-implies-con con-sym*)

**qed**

**lemma** *join-is-lub*:

**assumes**  $t \lesssim v$  **and**  $u \lesssim v$

**shows**  $t \sqcup u \lesssim v$

**proof** –

**have**  $(t \sqcup u) \setminus v = (t \setminus v) \sqcup (u \setminus v)$

**using** *assms resid-join<sub>E</sub>(3) [of t u v]*

**by** (*metis arr-prfx-join-self con-target con-sym join-assoc joinable-iff-con joinable-iff-join-not-null prfx-implies-con resid-reflects-con*)

```

also have ... =  $trg\ v \sqcup\ trg\ v$ 
  using assms
  by (metis ideE prfx-implies-con src-residWE trg-ide)
also have ... =  $trg\ v$ 
  by (metis assms(2) ide-iff-src-self ide-implies-arr join-arr-self prfx-implies-con
      src-residWE)
finally have  $(t \sqcup u) \setminus v = trg\ v$  by blast
moreover have ide ( $trg\ v$ )
  using assms
  by (metis con-implies-arr(2) prfx-implies-con cong-char trg-def)
ultimately show ?thesis by simp
qed

end

```

## Extensional RTS with Composites

If an extensional RTS is assumed to have composites for all composable pairs of transitions, then the “semantic” property of transitions being composable can be replaced by the “syntactic” property of transitions being sequential. This results in simpler statements of a number of properties.

```

locale extensional-rts-with-composites =
  rts-with-composites +
  extensional-rts
begin

```

```

lemma seq-implies-arr-comp:
assumes seq t u
shows arr (t · u)
  using assms
  by (meson composable-iff-arr-comp composable-iff-seq)

```

```

lemma arr-compEC [intro, simp]:
assumes arr t and trg t = src u
shows arr (t · u)
  using assms
  by (simp add: seq-implies-arr-comp)

```

```

lemma arr-compEEC [elim]:
assumes arr (t · u)
and  $\llbracket arr\ t; arr\ u; trg\ t = src\ u \rrbracket \implies T$ 
shows T
  using assms composable-iff-arr-comp composable-iff-seq by blast

```

```

lemma trg-compEC [simp]:
assumes seq t u
shows trg (t · u) = trg u
  by (meson assms has-composites trg-comp)

```

**lemma** *src-comp*<sub>EC</sub> [*simp*]:

**assumes** *seq t u*

**shows**  $\text{src } (t \cdot u) = \text{src } t$

**using** *assms src-comp has-composites* **by** *simp*

**lemma** *con-comp-iff*<sub>EC</sub> [*simp*]:

**shows**  $w \frown t \cdot u \longleftrightarrow \text{seq } t \ u \wedge u \frown w \setminus t$

**and**  $t \cdot u \frown w \longleftrightarrow \text{seq } t \ u \wedge u \frown w \setminus t$

**using** *composable-iff-seq con-comp-iff con-sym* **by** *meson+*

**lemma** *comp-assoc*<sub>EC</sub>:

**shows**  $t \cdot (u \cdot v) = (t \cdot u) \cdot v$

**apply** (*cases seq t u*)

**apply** (*metis arr-comp comp-assoc comp-def not-arr-null arr-comp*<sub>EC</sub> *arr-comp*<sub>EC</sub> *seq-implies-arr-comp trg-comp*<sub>EC</sub>)

**by** (*metis comp-def composable-iff-arr-comp seqI*<sub>WE</sub>(1) *src-comp arr-comp*<sub>EC</sub>)

**lemma** *diamond-commutes*:

**shows**  $t \cdot (u \setminus t) = u \cdot (t \setminus u)$

**proof** (*cases t \frown u*)

**show**  $\neg t \frown u \implies ?thesis$

**by** (*metis comp-null*(2) *conI con-sym*)

**assume** *con: t \frown u*

**have**  $(t \cdot (u \setminus t)) \setminus u = (t \setminus u) \cdot ((u \setminus t) \setminus (u \setminus t))$

**using** *con*

**by** (*metis (no-types, lifting) arr-resid-iff-con con-compI*(2) *con-implies-arr*(1)

*resid-comp*(2) *con-imp-arr-resid con-sym comp-def arr-comp*<sub>EC</sub> *src-resid*<sub>WE</sub> *conI*)

**moreover** **have**  $u \lesssim t \cdot (u \setminus t)$

**by** (*metis arr-resid-iff-con calculation con cong-reflexive comp-arr-trg*

*resid-arr-self resid-comp*(1) *apex-sym*)

**ultimately** **show** *?thesis*

**by** (*metis comp-eqI con comp-arr-trg resid-arr-self arr-resid apex-sym*)

**qed**

**lemma** *mediating-transition*:

**assumes**  $t \cdot v = u \cdot w$

**shows**  $v \setminus (u \setminus t) = w \setminus (t \setminus u)$

**proof** (*cases seq t v*)

**assume** *1: seq t v*

**hence** *2: arr (u \cdot w)*

**using** *assms* **by** (*metis arr-comp*<sub>EC</sub> *seq*<sub>WE</sub>)

**have** *3: v \setminus (u \setminus t) = ((t \cdot v) \setminus t) \setminus (u \setminus t)*

**by** (*metis 2 assms comp-resid-prfx*)

**also** **have**  $\dots = (t \cdot v) \setminus (t \cdot (u \setminus t))$

**by** (*metis (no-types, lifting) 2 assms con-comp-iff*<sub>EC</sub>(2) *con-imp-eq-src*

*con-sym comp-resid-prfx prfx-comp resid-comp*(1) *arr-comp*<sub>EC</sub> *arr-comp*<sub>EC</sub>

*prfx-implies-con*)

**also** **have**  $\dots = (u \cdot w) \setminus (u \cdot (t \setminus u))$

**using** *assms diamond-commutes* **by** *presburger*

**also have**  $\dots = ((u \cdot w) \setminus u) \setminus (t \setminus u)$   
**by** (*metis 3 assms calculation cube*)  
**also have**  $\dots = w \setminus (t \setminus u)$   
**using 2 by simp**  
**finally show ?thesis by blast**  
**next**  
**assume 1:  $\neg \text{seq } t \ v$**   
**have**  $v \setminus (u \setminus t) = \text{null}$   
**using 1**  
**by** (*metis (mono-tags, lifting) arr-resid-iff-con coinital-iff<sub>WE</sub> con-imp-coinital seqI<sub>WE</sub>(2) src-resid<sub>WE</sub> conI*)  
**also have**  $\dots = w \setminus (t \setminus u)$   
**by** (*metis (no-types, lifting) 1 arr-comp<sub>EC</sub> assms composable-imp-seq con-imp-eq-src con-implies-arr(2) comp-def not-arr-null conI src-resid<sub>WE</sub>*)  
**finally show ?thesis by blast**  
**qed**

**lemma induced-arrow:**

**assumes**  $\text{seq } t \ u$  **and**  $t \cdot u = t' \cdot u'$   
**shows**  $(t' \setminus t) \cdot (u \setminus (t' \setminus t)) = u$   
**and**  $(t \setminus t') \cdot (u \setminus (t' \setminus t)) = u'$   
**and**  $(t' \setminus t) \cdot v = u \implies v = u \setminus (t' \setminus t)$   
**apply** (*metis assms comp-eqI arr-comp<sub>EC</sub> prfx-comp resid-comp(1) arr-resid-iff-con seq-implies-arr-comp*)  
**apply** (*metis assms comp-resid-prfx arr-comp<sub>EC</sub> resid-comp(2) arr-resid-iff-con seq-implies-arr-comp*)  
**by** (*metis assms(1) comp-resid-prfx seq-def*)

If an extensional RTS has composites, then it automatically has joins.

**sublocale** *extensional-rts-with-joins*

**proof**

**fix**  $t \ u$

**assume**  $\text{con: } t \ \frown \ u$

**have 1: con**  $u \ (t \cdot (u \setminus t))$

**using** *con-compI(1) [of  $t \ u \ \setminus \ t \ u$ ]*

**by** (*metis con con-implies-arr(1) con-sym diamond-commutes prfx-implies-con prfx-comp src-resid<sub>WE</sub> arr-comp<sub>EC</sub>*)

**have**  $t \sqcup u = t \cdot (u \setminus t)$

**proof** (*intro join-eqI*)

**show**  $t \lesssim t \cdot (u \setminus t)$

**by** (*metis 1 composable-def comp-is-composite-of(2) composite-of-def con-comp-iff*)

**moreover show 2:  $u \lesssim t \cdot (u \setminus t)$**

**using 1 arr-resid con con-sym prfx-reflexive resid-comp(1) by metis**

**moreover show**  $(t \cdot (u \setminus t)) \setminus u = t \setminus u$

**using 1 diamond-commutes induced-arrow(2) resid-comp(2) by force**

**ultimately show**  $(t \cdot (u \setminus t)) \setminus t = u \setminus t$

**by** (*metis con-comp-iff<sub>EC</sub>(1) con-sym prfx-implies-con resid-comp(2) induced-arrow(1)*)

```

qed
thus joinable t u
  by (metis 1 con-implies-arr(2) joinable-iff-join-not-null not-arr-null)
qed

```

```

lemma comp-joinEC:
assumes composable t u and joinable u u'
shows composable t (u  $\sqcup$  u')
and t · (u  $\sqcup$  u') = t · u  $\sqcup$  t · u'
proof –
  have 1: u  $\sqcup$  u' = u · (u' \ u)  $\wedge$  u  $\sqcup$  u' = u' · (u \ u')
    using assms joinable-implies-con diamond-commutes
    by (metis comp-is-composite-of(2) join-is-join-of join-ofE)
  show 2: composable t (u  $\sqcup$  u')
    using assms 1 composable-iff-seq arr-comp src-join arr-compEEC
      joinable-iff-arr-join seqIWE(1)
    by metis
  have con (t · u) (t · u')
    using 1 2 arr-comp arr-compEEC assms(2) comp-assocEC comp-resid-prfx
      con-comp-iff joinable-implies-con comp-def not-arr-null
    by metis
  thus t · (u  $\sqcup$  u') = t · u  $\sqcup$  t · u'
    using assms comp-join(2) joinable-iff-con by blast
qed

```

```

lemma resid-common-prefix:
assumes t · u  $\frown$  t · v
shows (t · u) \ (t · v) = u \ v
  using assms
  by (metis con-comp-iff con-sym con-comp-iffEC(2) con-implies-arr(2)
    induced-arrow(1) resid-comp(1–2) arr-resid-iff-con)

```

end

### 2.1.10 Confluence

An RTS is *confluent* if every cointial pair of transitions is consistent.

```

locale confluent-rts = rts +
assumes confluence: cointial t u  $\implies$  con t u

```

## 2.2 Simulations

*Simulations* are morphisms of residuated transition systems. They are assumed to preserve consistency and residuation.

```

locale simulation =
  A: rts A +
  B: rts B
for A :: 'a resid    (infix <\A> 70)

```

**and**  $B :: 'b \text{ resid}$     (**infix**  $\langle \setminus_B \rangle$  70)  
**and**  $F :: 'a \Rightarrow 'b +$   
**assumes** *extensionality*:  $\neg A.\text{arr } t \Longrightarrow F t = B.\text{null}$   
**and** *preserves-con* [*simp*]:  $A.\text{con } t u \Longrightarrow B.\text{con } (F t) (F u)$   
**and** *preserves-resid* [*simp*]:  $A.\text{con } t u \Longrightarrow F (t \setminus_A u) = F t \setminus_B F u$   
**begin**

**notation**  $A.\text{con}$     (**infix**  $\langle \frown_A \rangle$  50)  
**notation**  $A.\text{prfx}$    (**infix**  $\langle \lesssim_A \rangle$  50)  
**notation**  $A.\text{cong}$     (**infix**  $\langle \sim_A \rangle$  50)

**notation**  $B.\text{con}$     (**infix**  $\langle \frown_B \rangle$  50)  
**notation**  $B.\text{prfx}$    (**infix**  $\langle \lesssim_B \rangle$  50)  
**notation**  $B.\text{cong}$     (**infix**  $\langle \sim_B \rangle$  50)

**lemma** *preserves-reflects-arr* [*iff*]:  
**shows**  $B.\text{arr } (F t) \longleftrightarrow A.\text{arr } t$   
**by** (*metis*  $A.\text{arr-def } B.\text{con-implies-arr}(2) B.\text{not-arr-null extensionality preserves-con}$ )

**lemma** *preserves-ide* [*simp*]:  
**assumes**  $A.\text{ide } a$   
**shows**  $B.\text{ide } (F a)$   
**by** (*metis*  $A.\text{ideE assms preserves-con preserves-resid } B.\text{ideI}$ )

**lemma** *preserves-sources*:  
**shows**  $F \text{ ` } A.\text{sources } t \subseteq B.\text{sources } (F t)$   
**using**  $A.\text{sources-def } B.\text{sources-def preserves-con preserves-ide by auto}$

**lemma** *preserves-targets*:  
**shows**  $F \text{ ` } A.\text{targets } t \subseteq B.\text{targets } (F t)$   
**by** (*metis*  $A.\text{arrE } B.\text{arrE } A.\text{sources-resid } B.\text{sources-resid equals0D image-subset-iff } A.\text{arr-iff-has-target preserves-reflects-arr preserves-resid preserves-sources}$ )

**lemma** *preserves-trg* [*simp*]:  
**assumes**  $A.\text{arr } t$   
**shows**  $B.\text{trg } (F t) = F (A.\text{trg } t)$   
**using**  $\text{assms } A.\text{trg-def } B.\text{trg-def by auto}$

**lemma** *preserves-seq*:  
**shows**  $A.\text{seq } t u \Longrightarrow B.\text{seq } (F t) (F u)$   
**by** (*metis*  $A.\text{in-sourcesE } A.\text{seqE } B.\text{seqI}(1) B.\text{targets-composite-of preserves-con preserves-reflects-arr preserves-resid } B.\text{composite-of-arr-target } A.\text{resid-arr-ide } B.\text{sources-resid } A.\text{trg-in-targets } B.\text{trg-in-targets simulation.preserves-trg simulation-axioms}$ )

**lemma** *preserves-composites*:  
**assumes**  $A.\text{composite-of } t u v$   
**shows**  $B.\text{composite-of } (F t) (F u) (F v)$   
**using**  $\text{assms}$

by (*metis A.composite-ofE A.prfx-implies-con B.composite-of-def preserves-ide preserves-resid A.con-sym*)

**lemma** *preserves-joins*:

**assumes** *A.join-of t u v*

**shows** *B.join-of (F t) (F u) (F v)*

**using** *assms A.join-of-def B.join-of-def A.joinable-def*

**by** (*metis A.joinable-implies-con preserves-composites preserves-resid*)

**lemma** *preserves-prfx*:

**assumes**  $t \lesssim_A u$

**shows**  $F t \lesssim_B F u$

**using** *assms*

**by** (*metis A.prfx-implies-con preserves-ide preserves-resid*)

**lemma** *preserves-cong*:

**assumes**  $t \sim_A u$

**shows**  $F t \sim_B F u$

**using** *assms preserves-prfx by simp*

end

### 2.2.1 Identity Simulation

**locale** *identity-simulation* =

*rts*

**begin**

**abbreviation** *map*

**where**  $map \equiv \lambda t. \text{if arr } t \text{ then } t \text{ else null}$

**sublocale** *simulation resid resid map*

**using** *con-implies-arr con-sym arr-resid-iff-con*

**by** *unfold-locales auto*

end

### 2.2.2 Composite of Simulations

**lemma** *simulation-comp* [*intro*]:

**assumes** *simulation A B F and simulation B C G*

**shows** *simulation A C (G o F)*

**proof** –

**interpret** *F: simulation A B F using assms(1) by auto*

**interpret** *G: simulation B C G using assms(2) by auto*

**show** *simulation A C (G o F)*

**using** *F.extensionality G.extensionality by unfold-locales auto*

qed

**locale** *composite-simulation* =

```

    F: simulation A B F +
    G: simulation B C G
for A :: 'a resid
and B :: 'b resid
and C :: 'c resid
and F :: 'a ⇒ 'b
and G :: 'b ⇒ 'c
begin

    abbreviation map
    where map ≡ G o F

    sublocale simulation A C map
    using F.simulation-axioms G.simulation-axioms by blast

    lemma is-simulation:
    shows simulation A C map
    using F.simulation-axioms G.simulation-axioms by blast

end

```

### 2.2.3 Simulations into a Weakly Extensional RTS

```

locale simulation-to-weakly-extensional-rts =
  simulation +
  B: weakly-extensional-rts B
begin

    lemma preserves-src [simp]:
    shows a ∈ A.sources t ⇒ B.src (F t) = F a
    by (metis equalsOD image-subset-iff B.arr-iff-has-source
        preserves-sources B.arr-has-un-source B.src-in-sources)

    lemma preserves-trg [simp]:
    shows b ∈ A.targets t ⇒ B.trg (F t) = F b
    by (metis equalsOD image-subset-iff B.arr-iff-has-target
        preserves-targets B.arr-has-un-target B.trg-in-targets)

end

```

### 2.2.4 Simulations into an Extensional RTS

```

locale simulation-to-extensional-rts =
  simulation +
  B: extensional-rts B
begin

    notation B.comp (infixr ⟨·B⟩ 55)
    notation B.join (infixr ⟨⊔B⟩ 52)

```

**lemma** *preserves-comp*:  
**assumes** *A.composite-of t u v*  
**shows**  $F v = F t \cdot_B F u$   
**using** *assms*  
**by** (*metis preserves-composites B.comp-is-composite-of(2)*)

**lemma** *preserves-join*:  
**assumes** *A.join-of t u v*  
**shows**  $F v = F t \sqcup_B F u$   
**using** *assms preserves-joins*  
**by** (*meson B.join-is-join-of B.join-of-unique B.joinable-def*)

**end**

## 2.2.5 Simulations between Weakly Extensional RTS's

**locale** *simulation-between-weakly-extensional-rts* =  
*simulation-to-weakly-extensional-rts* +  
*A: weakly-extensional-rts A*

**begin**

**lemma** *preserves-src* [*simp*]:  
**shows**  $B.src (F t) = F (A.src t)$   
**by** (*metis A.arr-src-iff-arr A.src-in-sources extensionality image-subset-iff*  
*preserves-reflects-arr preserves-sources B.arr-has-un-source B.src-def*  
*B.src-in-sources*)

**lemma** *preserves-trg* [*simp*]:  
**shows**  $B.trg (F t) = F (A.trg t)$   
**by** (*metis A.arr-trg-iff-arr A.trg-def B.null-is-zero(2) B.trg-def*  
*extensionality preserves-resid A.arrE*)

**end**

## 2.2.6 Simulations between Extensional RTS's

**locale** *simulation-between-extensional-rts* =  
*simulation-to-extensional-rts* +  
*A: extensional-rts A*

**begin**

**sublocale** *simulation-between-weakly-extensional-rts ..*

**notation** *A.comp* (**infixr**  $\cdot_A$  55)

**notation** *A.join* (**infixr**  $\sqcup_A$  52)

**lemma** *preserves-comp*:  
**assumes** *A.composable t u*  
**shows**  $F (t \cdot_A u) = F t \cdot_B F u$   
**using** *assms*

by (*metis*  $A.arr\text{-}comp$   $A.comp\text{-}resid\text{-}prfx$   $A.composableD(2)$   $A.not\text{-}arr\text{-}null$   
 $A.prfx\text{-}comp$   $B.comp\text{-}eqI$  *preserves-prfx* *preserves-resid*  $A.conI$ )

**lemma** *preserves-join*:

**assumes**  $A.joinable$   $t$   $u$

**shows**  $F(t \sqcup_A u) = F t \sqcup_B F u$

**using** *assms*

by (*meson*  $A.join\text{-}is\text{-}join\text{-}of$   $B.joinable\text{-}def$  *preserves-joins*  $B.join\text{-}is\text{-}join\text{-}of$   
 $B.join\text{-}of\text{-}unique$ )

end

## 2.2.7 Transformations

A *transformation* is a morphism of simulations, analogously to how a natural transformation is a morphism of functors, except the normal commutativity condition for that “naturality squares” is replaced by the requirement that the arrows at the apex of such a square are given by residuation of the arrows at the base. If the codomain RTS is extensional, then this condition implies the commutativity of the square with respect to composition, as would be the case for a natural transformation between functors.

The proper way to define a transformation when the domain and codomain are general RTS’s is not yet clear to me. However, if the codomain is weakly extensional, then we have unique sources and targets, so there is no problem. The definition below is limited to that case. I do not make any attempt here to develop facts about transformations. My main reason for including this definition here is so that in the subsequent application to the  $\lambda$ -calculus, I can exhibit  $\beta$ -reduction as an example of a transformation.

**locale** *transformation* =

$A$ : *rts*  $A$  +

$B$ : *weakly-extensional-rts*  $B$  +

$F$ : *simulation*  $A$   $B$   $F$  +

$G$ : *simulation*  $A$   $B$   $G$

**for**  $A$  :: ' $a$  *resid* (infix  $\langle \backslash_A \rangle$  70)

**and**  $B$  :: ' $b$  *resid* (infix  $\langle \backslash_B \rangle$  70)

**and**  $F$  :: ' $a \Rightarrow 'b$

**and**  $G$  :: ' $a \Rightarrow 'b$

**and**  $\tau$  :: ' $a \Rightarrow 'b$  +

**assumes** *extensionality*:  $\neg A.arr$   $f \Longrightarrow \tau f = B.null$

**and** *respects-cong-ide*:  $\llbracket A.ide$   $a$ ;  $A.cong$   $a$   $a' \rrbracket \Longrightarrow \tau a = \tau a'$

**and** *preserves-src*:  $A.ide$   $f \Longrightarrow B.src$   $(\tau f) = F f$

**and** *preserves-trg*:  $A.ide$   $f \Longrightarrow B.trg$   $(\tau f) = G f$

**and** *naturality1-ax*:  $a \in A.sources$   $f \Longrightarrow \tau a \backslash_B F f = \tau (a \backslash_A f)$

**and** *naturality2-ax*:  $a \in A.sources$   $f \Longrightarrow F f \backslash_B \tau a = G f$

**and** *naturality3*:  $a \in A.sources$   $f \Longrightarrow B.join\text{-}of$   $(\tau a)$   $(F f)$   $(\tau f)$

**begin**

**notation**  $A.con$  (infix  $\langle \frown_A \rangle$  50)

**notation**  $A.prfx$  (infix  $\langle \lesssim_A \rangle$  50)

**notation**  $B.con$  (infix  $\langle \sim_B \rangle$  50)  
**notation**  $B.pfx$  (infix  $\langle \lesssim_B \rangle$  50)

**lemma** *naturality1*:

**shows**  $\tau (A.src\ f) \setminus_B F\ f = \tau (A.trg\ f)$   
**by** (*metis*  $A.arr\ trg\ iff\ arr$   $A.ide\ trg$   $A.resid\ src\ arr\ cong$   
 $A.src\ in\ sources$   $B.null\ is\ zero(2)$   $F.extensionality$   $extensionality$   
*naturality1-ax* *respects-cong-ide*)

**lemma** *naturality2*:

**shows**  $F\ f \setminus_B \tau (A.src\ f) = G\ f$   
**by** (*metis*  $A.src\ in\ sources$   $B.null\ is\ zero(1)$   $F.extensionality$   $G.extensionality$   
*naturality2-ax*)

**lemma** *respects-cong*:

**assumes**  $A.cong\ u\ u'$

**shows**  $B.cong\ (\tau\ u)\ (\tau\ u')$

**proof** –

**obtain**  $a$  **where**  $a: a \in A.sources\ u \cap A.sources\ u'$

**using** *assms*

**by** (*meson*  $A.con\ imp\ common\ source$   $A.pfx\ implies\ con$  *ex-in-conv*)

**have**  $B.cong\ (F\ u)\ (F\ u')$

**using** *assms*  $F.preserves\ cong$  **by** *blast*

**thus** *?thesis*

**using**  $a$  *naturality3*  $B.join\ of\ respects\ cong\ right$

**by** (*metis*  $A.con\ imp\ common\ source$   $A.pfx\ implies\ con$   $A.sources\ eqI$   
 $B.join\ of\ un\ upto\ cong$  *assms* *inf.idem*)

**qed**

**end**

## 2.3 Normal Sub-RTS's and Congruence

We now develop a general quotient construction on an RTS. We define a *normal sub-RTS* of an RTS to be a collection of transitions  $\mathfrak{N}$  having certain “local” closure properties. A normal sub-RTS induces an equivalence relation  $\approx_0$ , which we call *semi-congruence*, by defining  $t \approx_0 u$  to hold exactly when  $t \setminus u$  and  $u \setminus t$  are both in  $\mathfrak{N}$ . This relation generalizes the relation  $\sim$  defined for an arbitrary RTS, in the sense that  $\sim$  is obtained when  $\mathfrak{N}$  consists of all and only the identity transitions. However, in general the relation  $\approx_0$  is fully substitutive only in the left argument position of residuation; for the right argument position, a somewhat weaker property is satisfied. We then coarsen  $\approx_0$  to a relation  $\approx$ , by defining  $t \approx u$  to hold exactly when  $t$  and  $u$  can be transported by residuation along transitions in  $\mathfrak{N}$  to a common source, in such a way that the residuals are related by  $\approx_0$ . To obtain full substitutivity of  $\approx$  with respect to residuation, we need to impose an additional condition on  $\mathfrak{N}$ . This condition, which we call *coherence*, states that transporting a transition  $t$  along parallel transitions  $u$  and  $v$  in  $\mathfrak{N}$  always yields residuals  $t \setminus u$  and  $u \setminus t$  that are related by  $\approx_0$ . We show that, under the assumption

of coherence, the relation  $\approx$  is fully substitutive, and the quotient of the original RTS by this relation is an extensional RTS which has the  $\mathfrak{N}$ -connected components of the original RTS as identities. Although the coherence property has a somewhat *ad hoc* feel to it, we show that, in the context of the other conditions assumed for  $\mathfrak{N}$ , coherence is in fact equivalent to substitutivity for  $\approx$ .

### 2.3.1 Normal Sub-RTS's

```

locale normal-sub-rts =
  R: rts +
  fixes  $\mathfrak{N} :: 'a$  set
  assumes elements-are-arr:  $t \in \mathfrak{N} \implies R.arr\ t$ 
  and ide-closed:  $R.ide\ a \implies a \in \mathfrak{N}$ 
  and forward-stable:  $\llbracket u \in \mathfrak{N}; R.coinitial\ t\ u \rrbracket \implies u \setminus t \in \mathfrak{N}$ 
  and backward-stable:  $\llbracket u \in \mathfrak{N}; t \setminus u \in \mathfrak{N} \rrbracket \implies t \in \mathfrak{N}$ 
  and composite-closed-left:  $\llbracket u \in \mathfrak{N}; R.seq\ u\ t \rrbracket \implies \exists v. R.composite-of\ u\ t\ v$ 
  and composite-closed-right:  $\llbracket u \in \mathfrak{N}; R.seq\ t\ u \rrbracket \implies \exists v. R.composite-of\ t\ u\ v$ 
begin

```

```

lemma prfx-closed:
assumes  $u \in \mathfrak{N}$  and  $R.prfx\ t\ u$ 
shows  $t \in \mathfrak{N}$ 
  using assms backward-stable ide-closed by blast

```

```

lemma composite-closed:
assumes  $t \in \mathfrak{N}$  and  $u \in \mathfrak{N}$  and  $R.composite-of\ t\ u\ v$ 
shows  $v \in \mathfrak{N}$ 
  using assms backward-stable R.composite-of-def prfx-closed by blast

```

```

lemma factor-closed:
assumes  $R.composite-of\ t\ u\ v$  and  $v \in \mathfrak{N}$ 
shows  $t \in \mathfrak{N}$  and  $u \in \mathfrak{N}$ 
  apply (metis assms R.composite-of-def prfx-closed)
  by (meson assms R.composite-of-def R.con-imp-coinitial forward-stable prfx-closed
      R.prfx-implies-con)

```

```

lemma resid-along-elem-preserves-con:
assumes  $t \frown t'$  and  $R.coinitial\ t\ u$  and  $u \in \mathfrak{N}$ 
shows  $t \setminus u \frown t' \setminus u$ 
proof –
  have  $R.coinitial\ (t \setminus t')\ (u \setminus t')$ 
    by (metis assms R.arr-resid-iff-con R.coinitialI R.con-imp-common-source forward-stable
        elements-are-arr R.con-implies-arr(2) R.sources-resid R.sources-eqI)
  hence  $t \setminus t' \frown u \setminus t'$ 
    by (metis assms(3) R.coinitial-iff R.con-imp-coinitial R.con-sym elements-are-arr
        forward-stable R.arr-resid-iff-con)
  thus ?thesis
    using assms R.cube forward-stable by fastforce
qed

```

end

## Normal Sub-RTS's of an Extensional RTS with Composites

**locale** *normal-in-extensional-rts-with-composites* =

*R*: *extensional-rts* +  
*R*: *rts-with-composites* +  
*normal-sub-rts*

**begin**

**lemma** *factor-closed*<sub>EC</sub>:

**assumes**  $t \cdot u \in \mathfrak{N}$

**shows**  $t \in \mathfrak{N}$  and  $u \in \mathfrak{N}$

**using** *assms factor-closed*

**by** (*metis R.arrE R.composable-def R.comp-is-composite-of(2) R.con-comp-iff elements-are-arr*)+

**lemma** *comp-in-normal-iff*:

**shows**  $t \cdot u \in \mathfrak{N} \iff t \in \mathfrak{N} \wedge u \in \mathfrak{N} \wedge R.seq\ t\ u$

**by** (*metis R.comp-is-composite-of(2) composite-closed elements-are-arr factor-closed(1-2) R.composable-def R.has-composites R.rts-with-composites-axioms R.extensional-rts-axioms extensional-rts-with-composites.arr-compE<sub>EC</sub> extensional-rts-with-composites-def R.seqI<sub>WE</sub>(1)*)

end

### 2.3.2 Semi-Congruence

**context** *normal-sub-rts*

**begin**

We will refer to the elements of  $\mathfrak{N}$  as *normal transitions*. Generalizing identity transitions to normal transitions in the definition of congruence, we obtain the notion of *semi-congruence* of transitions with respect to a normal sub-RTS.

**abbreviation** *Cong*<sub>0</sub> (**infix**  $\langle \approx_0 \rangle$  50)

**where**  $t \approx_0 t' \equiv t \setminus t' \in \mathfrak{N} \wedge t' \setminus t \in \mathfrak{N}$

**lemma** *Cong*<sub>0</sub>-*reflexive*:

**assumes** *R.arr* *t*

**shows**  $t \approx_0 t$

**using** *assms R.cong-reflexive ide-closed by simp*

**lemma** *Cong*<sub>0</sub>-*symmetric*:

**assumes**  $t \approx_0 t'$

**shows**  $t' \approx_0 t$

**using** *assms by simp*

**lemma** *Cong*<sub>0</sub>-*transitive* [*trans*]:

**assumes**  $t \approx_0 t'$  **and**  $t' \approx_0 t''$   
**shows**  $t \approx_0 t''$   
**by** (*metis* (*full-types*) *R.arr-resid-iff-con* *assms* *backward-stable* *forward-stable*  
*elements-are-arr* *R.coinitialI* *R.cube* *R.sources-resid*)

**lemma** *Cong<sub>0</sub>-imp-con*:  
**assumes**  $t \approx_0 t'$   
**shows**  $R.con\ t\ t'$   
**using** *assms* *R.arr-resid-iff-con* *elements-are-arr* **by** *blast*

**lemma** *Cong<sub>0</sub>-imp-coinitial*:  
**assumes**  $t \approx_0 t'$   
**shows**  $R.sources\ t = R.sources\ t'$   
**using** *assms* **by** (*meson* *Cong<sub>0</sub>-imp-con* *R.coinitial-iff* *R.con-imp-coinitial*)

Semi-congruence is preserved and reflected by residuation along normal transitions.

**lemma** *Resid-along-normal-preserves-Cong<sub>0</sub>*:  
**assumes**  $t \approx_0 t'$  **and**  $u \in \mathfrak{N}$  **and**  $R.sources\ t = R.sources\ u$   
**shows**  $t \setminus u \approx_0 t' \setminus u$   
**by** (*metis* *Cong<sub>0</sub>-imp-coinitial* *R.arr-resid-iff-con* *R.coinitialI* *R.coinitial-def*  
*R.cube* *R.sources-resid* *assms* *elements-are-arr* *forward-stable*)

**lemma** *Resid-along-normal-reflects-Cong<sub>0</sub>*:  
**assumes**  $t \setminus u \approx_0 t' \setminus u$  **and**  $u \in \mathfrak{N}$   
**shows**  $t \approx_0 t'$   
**using** *assms*  
**by** (*metis* *backward-stable* *R.con-imp-coinitial* *R.cube* *R.null-is-zero(2)*  
*forward-stable* *R.conI*)

Semi-congruence is substitutive for the left-hand argument of residuation.

**lemma** *Cong<sub>0</sub>-subst-left*:  
**assumes**  $t \approx_0 t'$  **and**  $t \frown u$   
**shows**  $t' \frown u$  **and**  $t \setminus u \approx_0 t' \setminus u$   
**proof** –  
**have** 1:  $t \frown u \wedge t \frown t' \wedge u \setminus t \frown t' \setminus t$   
**using** *assms*  
**by** (*metis* *Resid-along-normal-preserves-Cong<sub>0</sub>* *Cong<sub>0</sub>-imp-con* *Cong<sub>0</sub>-reflexive* *R.con-sym*  
*R.null-is-zero(2)* *R.arr-resid-iff-con* *R.sources-resid* *R.conI*)  
**hence** 2:  $t' \frown u \wedge u \setminus t \frown t' \setminus t \wedge$   
 $(t \setminus u) \setminus (t' \setminus u) = (t \setminus t') \setminus (u \setminus t') \wedge$   
 $(t' \setminus u) \setminus (t \setminus u) = (t' \setminus t) \setminus (u \setminus t)$   
**by** (*meson* *R.con-sym* *R.cube* *R.resid-reflects-con*)  
**show**  $t' \frown u$   
**using** 2 **by** *simp*  
**show**  $t \setminus u \approx_0 t' \setminus u$   
**using** *assms* 1 2  
**by** (*metis* *R.arr-resid-iff-con* *R.con-imp-coinitial* *R.cube* *forward-stable*)  
**qed**

Semi-congruence is not exactly substitutive for residuation on the right. Instead, the

following weaker property is satisfied. Obtaining exact substitutivity on the right is the motivation for defining a coarser notion of congruence below.

**lemma** *Cong<sub>0</sub>-subst-right*:

**assumes**  $u \approx_0 u'$  **and**  $t \frown u$

**shows**  $t \frown u'$  **and**  $(t \setminus u) \setminus (u' \setminus u) \approx_0 (t \setminus u') \setminus (u \setminus u')$

**using** *assms*

**apply** (*meson Cong<sub>0</sub>-subst-left(1) R.con-sym*)

**using** *assms*

**by** (*metis R.sources-resid Cong<sub>0</sub>-imp-con Cong<sub>0</sub>-reflexive Resid-along-normal-preserves-Cong<sub>0</sub> R.arr-resid-iff-con residuation.cube R.residuation-axioms*)

**lemma** *Cong<sub>0</sub>-subst-Con*:

**assumes**  $t \approx_0 t'$  **and**  $u \approx_0 u'$

**shows**  $t \frown u \longleftrightarrow t' \frown u'$

**using** *assms*

**by** (*meson Cong<sub>0</sub>-subst-left(1) Cong<sub>0</sub>-subst-right(1)*)

**lemma** *Cong<sub>0</sub>-cancel-left*:

**assumes** *R.composite-of*  $t u v$  **and** *R.composite-of*  $t u' v'$  **and**  $v \approx_0 v'$

**shows**  $u \approx_0 u'$

**proof** –

**have**  $u \approx_0 v \setminus t$

**using** *assms(1) ide-closed* **by** *blast*

**also have**  $v \setminus t \approx_0 v' \setminus t$

**by** (*meson assms(1,3) Cong<sub>0</sub>-subst-left(2) R.composite-of-def R.con-sym R.prfx-implies-con*)

**also have**  $v' \setminus t \approx_0 u'$

**using** *assms(2) ide-closed* **by** *blast*

**finally show** *?thesis* **by** *auto*

**qed**

**lemma** *Cong<sub>0</sub>-iff*:

**shows**  $t \approx_0 t' \longleftrightarrow$

$(\exists u u' v v'. u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge v \approx_0 v' \wedge$

$R.composite-of\ t\ u\ v \wedge R.composite-of\ t'\ u'\ v')$

**proof** (*intro iffI*)

**show**  $\exists u u' v v'. u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge v \approx_0 v' \wedge$

$R.composite-of\ t\ u\ v \wedge R.composite-of\ t'\ u'\ v'$

$\implies t \approx_0 t'$

**by** (*meson Cong<sub>0</sub>-transitive R.composite-of-def ide-closed prfx-closed*)

**show**  $t \approx_0 t' \implies \exists u u' v v'. u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge v \approx_0 v' \wedge$

$R.composite-of\ t\ u\ v \wedge R.composite-of\ t'\ u'\ v'$

**by** (*metis Cong<sub>0</sub>-imp-con Cong<sub>0</sub>-transitive R.composite-of-def R.prfx-reflexive*

*R.arrI R.ideE*)

**qed**

**lemma** *diamond-commutes-upto-Cong<sub>0</sub>*:

**assumes**  $t \frown u$  **and** *R.composite-of*  $t (u \setminus t) v$  **and** *R.composite-of*  $u (t \setminus u) v'$

**shows**  $v \approx_0 v'$

**proof** –

**have**  $v \setminus v \approx_0 v' \setminus v \wedge v' \setminus v' \approx_0 v \setminus v'$   
**proof**–  
**have**  $1: (v \setminus t) \setminus (u \setminus t) \approx_0 (v' \setminus u) \setminus (t \setminus u)$   
**using** *assms(2–3) R.cube [of v t u]*  
**by** (*metis R.con-target R.composite-ofE R.ide-imp-con-iff-cong ide-closed R.conI*)  
**have**  $2: v \setminus v \approx_0 v' \setminus v$   
**proof** –  
**have**  $v \setminus v \approx_0 (v \setminus t) \setminus (u \setminus t)$   
**using** *assms R.composite-of-def ide-closed*  
**by** (*meson R.composite-of-unq-upto-cong R.prfx-implies-con R.resid-composite-of(3)*)  
**also have**  $(v \setminus t) \setminus (u \setminus t) \approx_0 (v' \setminus u) \setminus (t \setminus u)$   
**using**  $1$  **by** *simp*  
**also have**  $(v' \setminus u) \setminus (t \setminus u) \approx_0 (v' \setminus t) \setminus (u \setminus t)$   
**by** (*metis 1 Cong<sub>0</sub>-transitive R.cube*)  
**also have**  $(v' \setminus t) \setminus (u \setminus t) \approx_0 v' \setminus v$   
**using** *assms R.composite-of-def ide-closed*  
**by** (*metis 1 R.conI R.con-sym-ax R.cube R.null-is-zero(2) R.resid-composite-of(3)*)  
**finally show** *?thesis* **by** *auto*  
**qed**  
**moreover have**  $v' \setminus v' \approx_0 v \setminus v'$   
**proof** –  
**have**  $v' \setminus v' \approx_0 (v' \setminus u) \setminus (t \setminus u)$   
**using** *assms R.composite-of-def ide-closed*  
**by** (*meson R.composite-of-unq-upto-cong R.prfx-implies-con R.resid-composite-of(3)*)  
**also have**  $(v' \setminus u) \setminus (t \setminus u) \approx_0 (v \setminus t) \setminus (u \setminus t)$   
**using**  $1$  **by** *simp*  
**also have**  $(v \setminus t) \setminus (u \setminus t) \approx_0 (v \setminus u) \setminus (t \setminus u)$   
**using** *R.cube [of v t u] ide-closed*  
**by** (*metis Cong<sub>0</sub>-reflexive R.arr-resid-iff-con assms(2) R.composite-of-def R.prfx-implies-con*)  
**also have**  $(v \setminus u) \setminus (t \setminus u) \approx_0 v \setminus v'$   
**using** *assms R.composite-of-def ide-closed*  
**by** (*metis 2 R.conI elements-are-arr R.not-arr-null R.null-is-zero(2) R.resid-composite-of(3)*)  
**finally show** *?thesis* **by** *auto*  
**qed**  
**ultimately show** *?thesis* **by** *blast*  
**qed**  
**thus** *?thesis*  
**by** (*metis assms(2–3) R.composite-of-unq-upto-cong R.resid-arr-ide Cong<sub>0</sub>-imp-con*)  
**qed**

### 2.3.3 Congruence

We use semi-congruence to define a coarser relation as follows.

**definition** *Cong* (**infix**  $\langle \approx \rangle$  50)  
**where**  $Cong\ t\ t' \equiv \exists u\ u'. u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge t \setminus u \approx_0 t' \setminus u'$

**lemma** *CongI* [*intro*]:  
**assumes**  $u \in \mathfrak{N}$  **and**  $u' \in \mathfrak{N}$  **and**  $t \setminus u \approx_0 t' \setminus u'$   
**shows** *Cong*  $t t'$   
**using** *assms Cong-def* **by** *auto*

**lemma** *CongE* [*elim*]:  
**assumes**  $t \approx t'$   
**obtains**  $u u'$   
**where**  $u \in \mathfrak{N}$  **and**  $u' \in \mathfrak{N}$  **and**  $t \setminus u \approx_0 t' \setminus u'$   
**using** *assms Cong-def* **by** *auto*

**lemma** *Cong-imp-arr*:  
**assumes**  $t \approx t'$   
**shows**  $R.arr\ t$  **and**  $R.arr\ t'$   
**using** *assms Cong-def*  
**by** (*meson R.arr-resid-iff-con R.con-implies-arr(2) R.con-sym elements-are-arr*) $+$

**lemma** *Cong-reflexive*:  
**assumes**  $R.arr\ t$   
**shows**  $t \approx t$   
**by** (*metis CongI Cong<sub>0</sub>-reflexive assms R.con-imp-coinitial-ax ide-closed R.resid-arr-ide R.arrE R.con-sym*)

**lemma** *Cong-symmetric*:  
**assumes**  $t \approx t'$   
**shows**  $t' \approx t$   
**using** *assms Cong-def* **by** *auto*

The existence of composites of normal transitions is used in the following.

**lemma** *Cong-transitive* [*trans*]:  
**assumes**  $t \approx t''$  **and**  $t'' \approx t'$   
**shows**  $t \approx t'$   
**proof** –  
**obtain**  $u u''$  **where**  $uu''$ :  $u \in \mathfrak{N} \wedge u'' \in \mathfrak{N} \wedge t \setminus u \approx_0 t'' \setminus u''$   
**using** *assms Cong-def* **by** *blast*  
**obtain**  $v' v''$  **where**  $v'v''$ :  $v' \in \mathfrak{N} \wedge v'' \in \mathfrak{N} \wedge t'' \setminus v'' \approx_0 t' \setminus v'$   
**using** *assms Cong-def* **by** *blast*  
**let**  $?w = (t \setminus u) \setminus (v'' \setminus u'')$   
**let**  $?w' = (t' \setminus v') \setminus (u'' \setminus v'')$   
**let**  $?w'' = (t'' \setminus v'') \setminus (u'' \setminus v'')$   
**have**  $w''$ :  $?w'' = (t'' \setminus u'') \setminus (v'' \setminus u'')$   
**by** (*metis R.cube*)  
**have**  $u''v''$ :  $R.coinitial\ u''\ v''$   
**by** (*metis (full-types) R.coinitial-iff elements-are-arr R.con-imp-coinitial R.arr-resid-iff-con uu'' v'v''*)  
**hence**  $v''u''$ :  $R.coinitial\ v''\ u''$   
**by** (*meson R.con-imp-coinitial elements-are-arr forward-stable R.arr-resid-iff-con v'v''*)  
**have**  $1$ :  $?w \setminus ?w'' \in \mathfrak{N}$   
**proof** –

**have**  $(v'' \setminus u'') \setminus (t'' \setminus u'') \in \mathfrak{N}$   
**by** (*metis* *Cong<sub>0</sub>-transitive* *R.con-imp-coinitial* *forward-stable* *Cong<sub>0</sub>-imp-con*  
*resid-along-elem-preserves-con* *R.arrI* *R.arr-resid-iff-con* *u''v'' uu'' v'v''*)  
**thus** *?thesis*  
**by** (*metis* *Cong<sub>0</sub>-subst-left(2)* *R.con-sym* *R.null-is-zero(1)* *uu'' w'' R.conI*)  
**qed**  
**have**  $2: ?w'' \setminus ?w \in \mathfrak{N}$   
**by** (*metis* *1* *Cong<sub>0</sub>-subst-left(2)* *uu'' w'' R.conI*)  
**have**  $3: R.seq\ u\ (v'' \setminus u'')$   
**by** (*metis* (*full-types*) *2* *Cong<sub>0</sub>-imp-coinitial* *R.sources-resid*  
*Cong<sub>0</sub>-imp-con* *R.arr-resid-iff-con* *R.con-implies-arr(2)* *R.seqI(1)* *uu'' R.conI*)  
**have**  $4: R.seq\ v'\ (u'' \setminus v'')$   
**by** (*metis* *1* *Cong<sub>0</sub>-imp-coinitial* *Cong<sub>0</sub>-imp-con* *R.arr-resid-iff-con*  
*R.con-implies-arr(2)* *R.seq-def* *R.sources-resid* *v'v'' R.conI*)  
**obtain**  $x$  **where**  $x: R.composite-of\ u\ (v'' \setminus u'')$   $x$   
**using** *3* *composite-closed-left* *uu''* **by** *blast*  
**obtain**  $x'$  **where**  $x': R.composite-of\ v'\ (u'' \setminus v'')$   $x'$   
**using** *4* *composite-closed-left* *v'v''* **by** *presburger*  
**have**  $?w \approx_0 ?w'$   
**proof** –  
**have**  $?w \approx_0 ?w'' \wedge ?w' \approx_0 ?w''$   
**using** *1* *2*  
**by** (*metis* *Cong<sub>0</sub>-subst-left(2)* *R.null-is-zero(2)* *v'v'' R.conI*)  
**thus** *?thesis*  
**using** *Cong<sub>0</sub>-transitive* **by** *blast*  
**qed**  
**moreover** **have**  $x \in \mathfrak{N} \wedge ?w \approx_0 t \setminus x$   
**apply** (*intro conjI*)  
**apply** (*meson* *composite-closed* *forward-stable* *u''v'' uu'' v'v'' x*)  
**apply** (*metis* (*full-types*) *R.arr-resid-iff-con* *R.con-implies-arr(2)* *R.con-sym*  
*ide-closed* *forward-stable* *R.composite-of-def* *R.resid-composite-of(3)*  
*Cong<sub>0</sub>-subst-right(1)* *prfx-closed* *u''v'' uu'' v'v'' x R.conI*)  
**by** (*metis* (*no-types, lifting*) *1* *R.con-composite-of-iff* *ide-closed*  
*R.resid-composite-of(3)* *R.arr-resid-iff-con* *R.con-implies-arr(1)* *R.con-sym* *x R.conI*)  
**moreover** **have**  $x' \in \mathfrak{N} \wedge ?w' \approx_0 t' \setminus x'$   
**apply** (*intro conjI*)  
**apply** (*meson* *composite-closed* *forward-stable* *uu'' v''u'' v'v'' x'*)  
**apply** (*metis* (*full-types*) *Cong<sub>0</sub>-subst-right(1)* *R.composite-ofE* *R.con-sym*  
*ide-closed* *forward-stable* *R.con-imp-coinitial* *prfx-closed*  
*R.resid-composite-of(3)* *R.arr-resid-iff-con* *R.con-implies-arr(1)* *uu'' v'v'' x' R.conI*)  
**by** (*metis* (*full-types*) *Cong<sub>0</sub>-subst-left(1)* *R.composite-ofE* *R.con-sym* *ide-closed*  
*forward-stable* *R.con-imp-coinitial* *prfx-closed* *R.resid-composite-of(3)*  
*R.arr-resid-iff-con* *R.con-implies-arr(1)* *uu'' v'v'' x' R.conI*)  
**ultimately** **show**  $t \approx t'$   
**using** *Cong-def* *Cong<sub>0</sub>-transitive* **by** *metis*  
**qed**

**lemma** *Cong-closure-props*:  
**shows**  $t \approx u \implies u \approx t$

**and**  $\llbracket t \approx u; u \approx v \rrbracket \implies t \approx v$   
**and**  $t \approx_0 u \implies t \approx u$   
**and**  $\llbracket u \in \mathfrak{N}; R.sources\ t = R.sources\ u \rrbracket \implies t \approx t \setminus u$   
**proof** –  
  **show**  $t \approx u \implies u \approx t$   
    **using** *Cong-symmetric* **by** *blast*  
  **show**  $\llbracket t \approx u; u \approx v \rrbracket \implies t \approx v$   
    **using** *Cong-transitive* **by** *blast*  
  **show**  $t \approx_0 u \implies t \approx u$   
    **by** (*metis Cong<sub>0</sub>-subst-left(2) Cong-def Cong-reflexive R.con-implies-arr(1)*  
      *R.null-is-zero(2) R.conI*)  
  **show**  $\llbracket u \in \mathfrak{N}; R.sources\ t = R.sources\ u \rrbracket \implies t \approx t \setminus u$   
**proof** –  
  **assume**  $u: u \in \mathfrak{N}$  **and** *coinitial: R.sources\ t = R.sources\ u*  
  **obtain**  $a$  **where**  $a: a \in R.targets\ u$   
    **by** (*meson elements-are-arr empty-subsetI R.arr-iff-has-target subsetI subset-antisym u*)  
  **have**  $t \setminus u \approx_0 (t \setminus u) \setminus a$   
**proof** –  
  **have**  $R.arr\ t$   
    **using** *R.arr-iff-has-source coinitial elements-are-arr u* **by** *presburger*  
  **thus** *?thesis*  
    **by** (*meson u a R.arr-resid-iff-con coinitial ide-closed forward-stable*  
      *elements-are-arr R.coinitial-iff R.composite-of-arr-target R.resid-composite-of(3)*)  
  **qed**  
  **thus** *?thesis*  
    **using** *Cong-def*  
    **by** (*metis a R.composite-of-arr-target elements-are-arr factor-closed(2) u*)  
  **qed**  
**qed**

**lemma** *Cong<sub>0</sub>-implies-Cong*:  
**assumes**  $t \approx_0 t'$   
**shows**  $t \approx t'$   
  **using** *assms Cong-closure-props(3)* **by** *simp*

**lemma** *in-sources-respects-Cong*:  
**assumes**  $t \approx t'$  **and**  $a \in R.sources\ t$  **and**  $a' \in R.sources\ t'$   
**shows**  $a \approx a'$   
**proof** –  
  **obtain**  $u\ u'$  **where**  $uu': u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge t \setminus u \approx_0 t' \setminus u'$   
    **using** *assms Cong-def* **by** *blast*  
  **show**  $a \approx a'$   
**proof**  
  **show**  $u \in \mathfrak{N}$   
    **using**  $uu'$  **by** *simp*  
  **show**  $u' \in \mathfrak{N}$   
    **using**  $uu'$  **by** *simp*  
  **show**  $a \setminus u \approx_0 a' \setminus u'$   
**proof** –

**have**  $a \setminus u \in R.targets\ u$   
**by** (*metis* *Cong<sub>0</sub>-imp-con* *R.arr-resid-iff-con* *assms(2)* *R.con-imp-common-source*  
*R.con-implies-arr(1)* *R.resid-source-in-targets* *R.sources-eqI uu'*)  
**moreover have**  $a' \setminus u' \in R.targets\ u'$   
**by** (*metis* *Cong<sub>0</sub>-imp-con* *R.arr-resid-iff-con* *assms(3)* *R.con-imp-common-source*  
*R.resid-source-in-targets* *R.con-implies-arr(1)* *R.sources-eqI uu'*)  
**moreover have**  $R.targets\ u = R.targets\ u'$   
**by** (*metis* *Cong<sub>0</sub>-imp-coinitial* *Cong<sub>0</sub>-imp-con* *R.arr-resid-iff-con*  
*R.con-implies-arr(1)* *R.sources-resid uu'*)  
**ultimately show** *?thesis*  
**using** *ide-closed* *R.targets-are-cong* **by** *presburger*  
**qed**  
**qed**  
**qed**

**lemma** *in-targets-respects-Cong*:

**assumes**  $t \approx t'$  **and**  $b \in R.targets\ t$  **and**  $b' \in R.targets\ t'$

**shows**  $b \approx b'$

**proof** –

**obtain**  $u\ u'$  **where**  $uu': u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge t \setminus u \approx_0 t' \setminus u'$   
**using** *assms* *Cong-def* **by** *blast*  
**have**  $seq: R.seq\ (u \setminus t)\ ((t' \setminus u') \setminus (t \setminus u)) \wedge R.seq\ (u' \setminus t')\ ((t \setminus u) \setminus (t' \setminus u'))$   
**by** (*metis* *R.arr-iff-has-source* *R.arr-iff-has-target* *R.conI* *elements-are-arr* *R.not-arr-null*  
*R.seqI(2)* *R.sources-resid* *R.targets-resid-sym uu'*)  
**obtain**  $v$  **where**  $v: R.composite-of\ (u \setminus t)\ ((t' \setminus u') \setminus (t \setminus u))\ v$   
**using** *seq* *composite-closed-right uu'* **by** *presburger*  
**obtain**  $v'$  **where**  $v': R.composite-of\ (u' \setminus t')\ ((t \setminus u) \setminus (t' \setminus u'))\ v'$   
**using** *seq* *composite-closed-right uu'* **by** *presburger*  
**show**  $b \approx b'$   
**proof**  
**show**  $v\text{-in-}\mathfrak{N}: v \in \mathfrak{N}$   
**by** (*metis* *composite-closed* *R.con-imp-coinitial* *R.con-implies-arr(1)* *forward-stable*  
*R.composite-of-def* *R.prfx-implies-con* *R.arr-resid-iff-con* *R.con-sym uu' v*)  
**show**  $v'\text{-in-}\mathfrak{N}: v' \in \mathfrak{N}$   
**by** (*metis* *backward-stable* *R.composite-of-def* *R.con-imp-coinitial* *forward-stable*  
*R.null-is-zero(2)* *prfx-closed uu' v' R.conI*)  
**show**  $b \setminus v \approx_0 b' \setminus v'$   
**using** *assms*  $uu'\ v\ v'$   
**by** (*metis* *R.arr-resid-iff-con* *ide-closed* *R.seq-def* *R.sources-resid* *R.targets-resid-sym*  
*R.resid-source-in-targets* *seq* *R.sources-composite-of* *R.targets-are-cong*  
*R.targets-composite-of*)  
**qed**  
**qed**

**lemma** *sources-are-Cong*:

**assumes**  $a \in R.sources\ t$  **and**  $a' \in R.sources\ t$

**shows**  $a \approx a'$

**using** *assms*

**by** (*simp* *add: ide-closed* *R.sources-are-cong* *Cong-closure-props(3)*)

**lemma** *targets-are-Cong*:  
**assumes**  $b \in R.targets\ t$  **and**  $b' \in R.targets\ t$   
**shows**  $b \approx b'$   
**using** *assms*  
**by** (*simp add: ide-closed R.targets-are-cong Cong-closure-props(3)*)

It is *not* the case that sources and targets are  $\approx$ -closed; *i.e.*  $t \approx t' \implies sources\ t = sources\ t'$  and  $t \approx t' \implies targets\ t = targets\ t'$  do not hold, in general.

**lemma** *Resid-along-normal-preserves-reflects-con*:  
**assumes**  $u \in \mathfrak{N}$  **and**  $R.sources\ t = R.sources\ u$   
**shows**  $t \setminus u \frown t' \setminus u \longleftrightarrow t \frown t'$   
**by** (*metis R.arr-resid-iff-con assms R.con-implies-arr(1-2) elements-are-arr R.coinitial-iff R.resid-reflects-con resid-along-elem-preserves-con*)

We can alternatively characterize  $\approx$  as the least symmetric and transitive relation on transitions that extends  $\approx_0$  and has the property of being preserved by residuation along transitions in  $\mathfrak{N}$ .

**inductive** *Cong'*  
**where**  $\bigwedge t\ u. Cong'\ t\ u \implies Cong'\ u\ t$   
 $\mid \bigwedge t\ u\ v. \llbracket Cong'\ t\ u; Cong'\ u\ v \rrbracket \implies Cong'\ t\ v$   
 $\mid \bigwedge t\ u. t \approx_0\ u \implies Cong'\ t\ u$   
 $\mid \bigwedge t\ u. \llbracket R.arr\ t; u \in \mathfrak{N}; R.sources\ t = R.sources\ u \rrbracket \implies Cong'\ t\ (t \setminus u)$

**lemma** *Cong'-if*:  
**shows**  $\llbracket u \in \mathfrak{N}; u' \in \mathfrak{N}; t \setminus u \approx_0\ t' \setminus u' \rrbracket \implies Cong'\ t\ t'$   
**proof** –  
**assume**  $u: u \in \mathfrak{N}$  **and**  $u': u' \in \mathfrak{N}$  **and**  $1: t \setminus u \approx_0\ t' \setminus u'$   
**show**  $Cong'\ t\ t'$   
**using**  $u\ u'\ 1$   
**by** (*metis (no-types, lifting) Cong'.simps Cong<sub>0</sub>-imp-con R.arr-resid-iff-con R.coinitial-iff R.con-imp-coinitial*)

**qed**

**lemma** *Cong-char*:  
**shows**  $Cong\ t\ t' \longleftrightarrow Cong'\ t\ t'$   
**proof** –  
**have**  $Cong\ t\ t' \implies Cong'\ t\ t'$   
**using** *Cong-def Cong'-if* **by** *blast*  
**moreover** **have**  $Cong'\ t\ t' \implies Cong\ t\ t'$   
**apply** (*induction rule: Cong'.induct*)  
**using** *Cong-symmetric* **apply** *simp*  
**using** *Cong-transitive* **apply** *simp*  
**using** *Cong-closure-props(3)* **apply** *simp*  
**using** *Cong-closure-props(4)* **by** *simp*  
**ultimately** **show** *?thesis*  
**using** *Cong-def* **by** *blast*  
**qed**

**lemma** *normal-is-Cong-closed*:  
**assumes**  $t \in \mathfrak{N}$  **and**  $t \approx t'$   
**shows**  $t' \in \mathfrak{N}$   
**using** *assms*  
**by** (*metis* (*full-types*) *CongE* *R.con-imp-coinitial* *forward-stable*  
*R.null-is-zero*(2) *backward-stable* *R.conI*)

### 2.3.4 Congruence Classes

Here we develop some notions relating to the congruence classes of  $\approx$ .

**definition** *Cong-class*  $\langle \{\cdot\} \rangle$   
**where** *Cong-class*  $t \equiv \{t'. t \approx t'\}$

**definition** *is-Cong-class*  
**where** *is-Cong-class*  $\mathcal{T} \equiv \exists t. t \in \mathcal{T} \wedge \mathcal{T} = \{t\}$

**definition** *Cong-class-rep*  
**where** *Cong-class-rep*  $\mathcal{T} \equiv \text{SOME } t. t \in \mathcal{T}$

**lemma** *Cong-class-is-nonempty*:  
**assumes** *is-Cong-class*  $\mathcal{T}$   
**shows**  $\mathcal{T} \neq \{\}$   
**using** *assms* *is-Cong-class-def* *Cong-class-def* **by** *auto*

**lemma** *rep-in-Cong-class*:  
**assumes** *is-Cong-class*  $\mathcal{T}$   
**shows** *Cong-class-rep*  $\mathcal{T} \in \mathcal{T}$   
**using** *assms* *is-Cong-class-def* *Cong-class-rep-def* *someI-ex* [*of*  $\lambda t. t \in \mathcal{T}$ ]  
**by** *metis*

**lemma** *arr-in-Cong-class*:  
**assumes** *R.arr*  $t$   
**shows**  $t \in \{t\}$   
**using** *assms* *Cong-class-def* *Cong-reflexive* **by** *simp*

**lemma** *is-Cong-classI*:  
**assumes** *R.arr*  $t$   
**shows** *is-Cong-class*  $\{t\}$   
**using** *assms* *Cong-class-def* *is-Cong-class-def* *Cong-reflexive* **by** *blast*

**lemma** *is-Cong-classI'* [*intro*]:  
**assumes**  $\mathcal{T} \neq \{\}$   
**and**  $\bigwedge t t'. \llbracket t \in \mathcal{T}; t' \in \mathcal{T} \rrbracket \implies t \approx t'$   
**and**  $\bigwedge t t'. \llbracket t \in \mathcal{T}; t' \approx t \rrbracket \implies t' \in \mathcal{T}$   
**shows** *is-Cong-class*  $\mathcal{T}$   
**proof** –  
**obtain**  $t$  **where**  $t: t \in \mathcal{T}$   
**using** *assms* **by** *auto*  
**have**  $\mathcal{T} = \{t\}$

**unfolding** *Cong-class-def*  
**using** *assms(2-3) t by blast*  
**thus** *?thesis*  
**using** *is-Cong-class-def t by blast*  
**qed**

**lemma** *Cong-class-memb-is-arr*:  
**assumes** *is-Cong-class  $\mathcal{T}$  and  $t \in \mathcal{T}$*   
**shows** *R.arr t*  
**using** *assms Cong-class-def is-Cong-class-def Cong-imp-arr(2) by force*

**lemma** *Cong-class-membs-are-Cong*:  
**assumes** *is-Cong-class  $\mathcal{T}$  and  $t \in \mathcal{T}$  and  $t' \in \mathcal{T}$*   
**shows** *Cong t t'*  
**using** *assms Cong-class-def is-Cong-class-def*  
**by** (*metis CollectD Cong-closure-props(2) Cong-symmetric*)

**lemma** *Cong-class-eqI*:  
**assumes**  *$t \approx t'$*   
**shows**  $\{\!\{t\}\!\} = \{\!\{t'\}\!\}$   
**using** *assms Cong-class-def*  
**by** (*metis (full-types) Collect-cong Cong'.intros(1-2) Cong-char*)

**lemma** *Cong-class-eqI'*:  
**assumes** *is-Cong-class  $\mathcal{T}$  and is-Cong-class  $\mathcal{U}$  and  $\mathcal{T} \cap \mathcal{U} \neq \{\}$*   
**shows**  $\mathcal{T} = \mathcal{U}$   
**using** *assms is-Cong-class-def Cong-class-eqI Cong-class-membs-are-Cong Int-emptyI*  
**by** (*metis (no-types, lifting)*)

**lemma** *is-Cong-classE [elim]*:  
**assumes** *is-Cong-class  $\mathcal{T}$*   
**and**  $\llbracket \mathcal{T} \neq \{\}; \bigwedge t t'. \llbracket t \in \mathcal{T}; t' \in \mathcal{T} \rrbracket \implies t \approx t'; \bigwedge t t'. \llbracket t \in \mathcal{T}; t' \approx t \rrbracket \implies t' \in \mathcal{T} \rrbracket \implies T$   
**shows** *T*  
**proof** –  
**have**  $\mathcal{T}: \mathcal{T} \neq \{\}$   
**using** *assms Cong-class-is-nonempty by simp*  
**moreover have**  $1: \bigwedge t t'. \llbracket t \in \mathcal{T}; t' \in \mathcal{T} \rrbracket \implies t \approx t'$   
**using** *assms Cong-class-membs-are-Cong by metis*  
**moreover have**  $\bigwedge t t'. \llbracket t \in \mathcal{T}; t' \approx t \rrbracket \implies t' \in \mathcal{T}$   
**using** *assms Cong-class-def*  
**by** (*metis 1 Cong-class-eqI Cong-imp-arr(1) is-Cong-class-def arr-in-Cong-class*)  
**ultimately show** *?thesis*  
**using** *assms by blast*  
**qed**

**lemma** *Cong-class-rep [simp]*:  
**assumes** *is-Cong-class  $\mathcal{T}$*   
**shows**  $\{\!\{ \text{Cong-class-rep } \mathcal{T} \}\!\} = \mathcal{T}$   
**by** (*metis Cong-class-membs-are-Cong Cong-class-eqI assms is-Cong-class-def rep-in-Cong-class*)

**lemma** *Cong-class-memb-Cong-rep*:  
**assumes** *is-Cong-class*  $\mathcal{T}$  **and**  $t \in \mathcal{T}$   
**shows** *Cong*  $t$  (*Cong-class-rep*  $\mathcal{T}$ )  
**using** *assms Cong-class-membs-are-Cong rep-in-Cong-class* **by** *simp*

**lemma** *composite-of-normal-arr*:  
**shows**  $\llbracket R.arr\ t; u \in \mathfrak{N}; R.composite-of\ u\ t\ t' \rrbracket \implies t' \approx t$   
**by** (*meson Cong'.intros*(3) *Cong-char R.composite-of-def R.con-implies-arr*(2)  
*ide-closed R.prfx-implies-con Cong-closure-props*(2,4) *R.sources-composite-of*)

**lemma** *composite-of-arr-normal*:  
**shows**  $\llbracket arr\ t; u \in \mathfrak{N}; R.composite-of\ t\ u\ t' \rrbracket \implies t' \approx_0 t$   
**by** (*meson Cong-closure-props*(3) *R.composite-of-def ide-closed prfx-closed*)

**end**

### 2.3.5 Coherent Normal Sub-RTS's

A *coherent* normal sub-RTS is one that satisfies a parallel moves property with respect to arbitrary transitions. The congruence  $\approx$  induced by a coherent normal sub-RTS is fully substitutive with respect to consistency and residuation, and in fact coherence is equivalent to substitutivity in this context.

**locale** *coherent-normal-sub-rts = normal-sub-rts +*  
**assumes** *coherent*:  $\llbracket R.arr\ t; u \in \mathfrak{N}; u' \in \mathfrak{N}; R.sources\ u = R.sources\ u';$   
 $R.targets\ u = R.targets\ u'; R.sources\ t = R.sources\ u \rrbracket$   
 $\implies t \setminus u \approx_0 t \setminus u'$

**context** *normal-sub-rts*  
**begin**

The above “parallel moves” formulation of coherence is equivalent to the following formulation, which involves “opposing spans”.

**lemma** *coherent-iff*:  
**shows**  $(\forall t\ u\ u'. R.arr\ t \wedge u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge R.sources\ t = R.sources\ u \wedge$   
 $R.sources\ u = R.sources\ u' \wedge R.targets\ u = R.targets\ u'$   
 $\longrightarrow t \setminus u \approx_0 t \setminus u')$   
 $\iff$   
 $(\forall t\ t'\ v\ v'\ w\ w'. v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge$   
 $R.sources\ v = R.sources\ w \wedge R.sources\ v' = R.sources\ w' \wedge$   
 $R.targets\ w = R.targets\ w' \wedge t \setminus v \approx_0 t' \setminus v'$   
 $\longrightarrow t \setminus w \approx_0 t' \setminus w')$

**proof**

**assume** 1:  $\forall t\ t'\ v\ v'\ w\ w'. v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge$   
 $R.sources\ v = R.sources\ w \wedge R.sources\ v' = R.sources\ w' \wedge$   
 $R.targets\ w = R.targets\ w' \wedge t \setminus v \approx_0 t' \setminus v'$

$\longrightarrow t \setminus w \approx_0 t' \setminus w'$

**show**  $\forall t u u'. R.arr t \wedge u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge R.sources t = R.sources u \wedge$   
 $R.sources u = R.sources u' \wedge R.targets u = R.targets u'$   
 $\longrightarrow t \setminus u \approx_0 t \setminus u'$

**proof** (*intro allI impI, elim conjE*)  
**fix**  $t u u'$   
**assume**  $t: R.arr t$  **and**  $u: u \in \mathfrak{N}$  **and**  $u': u' \in \mathfrak{N}$   
**and**  $tu: R.sources t = R.sources u$  **and**  $sources: R.sources u = R.sources u'$   
**and**  $targets: R.targets u = R.targets u'$   
**show**  $t \setminus u \approx_0 t \setminus u'$   
**by** (*metis 1 Congo-reflexive Resid-along-normal-preserves-Cong<sub>0</sub> sources t targets tu u u'*)

**qed**

**next**

**assume 1:**  $\forall t u u'. R.arr t \wedge u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge R.sources t = R.sources u \wedge$   
 $R.sources u = R.sources u' \wedge R.targets u = R.targets u'$   
 $\longrightarrow t \setminus u \approx_0 t \setminus u'$

**show**  $\forall t t' v v' w w'. v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge$   
 $R.sources v = R.sources w \wedge R.sources v' = R.sources w' \wedge$   
 $R.targets w = R.targets w' \wedge t \setminus v \approx_0 t' \setminus v'$   
 $\longrightarrow t \setminus w \approx_0 t' \setminus w'$

**proof** (*intro allI impI, elim conjE*)  
**fix**  $t t' v v' w w'$   
**assume**  $v: v \in \mathfrak{N}$  **and**  $v': v' \in \mathfrak{N}$  **and**  $w: w \in \mathfrak{N}$  **and**  $w': w' \in \mathfrak{N}$   
**and**  $vw: R.sources v = R.sources w$  **and**  $v'w': R.sources v' = R.sources w'$   
**and**  $ww': R.targets w = R.targets w'$   
**and**  $tv'tv': (t \setminus v) \setminus (t' \setminus v') \in \mathfrak{N}$  **and**  $t'v'tv': (t' \setminus v') \setminus (t \setminus v) \in \mathfrak{N}$   
**show**  $t \setminus w \approx_0 t' \setminus w'$

**proof** –  
**have**  $\exists: R.sources t = R.sources v \wedge R.sources t' = R.sources v'$   
**using** *R.con-imp-coinitial*  
**by** (*meson Congo<sub>0</sub>-imp-con tv'tv' t'v'tv*  
*R.coinitial-iff R.arr-resid-iff-con*)

**have**  $2: t \setminus w \approx t' \setminus w'$   
**using** *Cong-closure-props*  
**by** (*metis tv'tv' t'v'tv 3 vw v'w' v v' w w'*)

**obtain**  $z z'$  **where**  $zz': z \in \mathfrak{N} \wedge z' \in \mathfrak{N} \wedge (t \setminus w) \setminus z \approx_0 (t' \setminus w') \setminus z'$   
**using**  $2$  **by** *auto*

**have**  $(t \setminus w) \setminus z \approx_0 (t \setminus w) \setminus z'$

**proof** –  
**have** *R.coinitial*  $((t \setminus w) \setminus z) ((t \setminus w) \setminus z')$

**proof** –  
**have**  $R.targets z = R.targets z'$   
**using**  $ww' zz'$   
**by** (*metis Congo<sub>0</sub>-imp-coinitial Congo<sub>0</sub>-imp-con R.con-sym-ax*  
*R.null-is-zero(2) R.sources-resid R.conI*)

**moreover** **have**  $R.sources ((t \setminus w) \setminus z) = R.targets z$   
**using**  $ww' zz'$   
**by** (*metis R.con-def R.not-arr-null R.null-is-zero(2)*)

```

      R.sources-resid elements-are-arr)
moreover have R.sources ((t \ w) \ z') = R.targets z'
  using ww' zz'
  by (metis Cong-closure-props(4) Cong-imp-arr(2) R.arr-resid-iff-con
      R.coinitial-iff R.con-imp-coinitial R.sources-resid)
ultimately show ?thesis
  using ww' zz'
  apply (intro R.coinitialI)
  apply auto
  by (meson R.arr-resid-iff-con R.con-implies-arr(2) elements-are-arr)
qed
thus ?thesis
  apply (intro conjI)
  by (metis 1 R.coinitial-iff R.con-imp-coinitial R.arr-resid-iff-con
      R.sources-resid zz')+
qed
hence (t \ w) \ z'  $\approx_0$  (t' \ w') \ z'
  using zz' Cong0-transitive Cong0-symmetric by blast
thus ?thesis
  using zz' Resid-along-normal-reflects-Cong0 by metis
qed
qed
qed

```

end

```

context coherent-normal-sub-rts
begin

```

The proof of the substitutivity of  $\approx$  with respect to residuation only uses coherence in the “opposing spans” form.

```

lemma coherent':
assumes v  $\in$   $\mathfrak{N}$  and v'  $\in$   $\mathfrak{N}$  and w  $\in$   $\mathfrak{N}$  and w'  $\in$   $\mathfrak{N}$ 
and R.sources v = R.sources w and R.sources v' = R.sources w'
and R.targets w = R.targets w' and t \ v  $\approx_0$  t' \ v'
shows t \ w  $\approx_0$  t' \ w'
proof
  show (t \ w) \ (t' \ w')  $\in$   $\mathfrak{N}$ 
    using assms coherent coherent-iff by meson
  show (t' \ w') \ (t \ w)  $\in$   $\mathfrak{N}$ 
    using assms coherent coherent-iff by meson
qed

```

The relation  $\approx$  is substitutive with respect to both arguments of residuation.

```

lemma Cong-subst:
assumes t  $\approx$  t' and u  $\approx$  u' and t  $\frown$  u and R.sources t' = R.sources u'
shows t'  $\frown$  u' and t \ u  $\approx$  t' \ u'
proof –
  obtain v v' where vv': v  $\in$   $\mathfrak{N}$   $\wedge$  v'  $\in$   $\mathfrak{N}$   $\wedge$  t \ v  $\approx_0$  t' \ v'

```

**using** *assms* **by** *auto*  
**obtain**  $w \ w'$  **where**  $ww'$ :  $w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge u \setminus w \approx_0 u' \setminus w'$   
**using** *assms* **by** *auto*  
**let**  $?x = t \setminus v$  **and**  $?x' = t' \setminus v'$   
**let**  $?y = u \setminus w$  **and**  $?y' = u' \setminus w'$   
**have**  $xx'$ :  $?x \approx_0 ?x'$   
**using** *assms*  $vv'$  **by** *blast*  
**have**  $yy'$ :  $?y \approx_0 ?y'$   
**using** *assms*  $ww'$  **by** *blast*  
**have**  $1$ :  $t \setminus w \approx_0 t' \setminus w'$   
**proof** –  
**have**  $R.sources \ v = R.sources \ w$   
**by** (*metis* (*no-types*, *lifting*) *Cong<sub>0</sub>-imp-con* *R.arr-resid-iff-con* *assms*(3)  
*R.con-imp-common-source* *R.con-implies-arr*(2) *R.sources-eqI*  $ww' \ xx'$ )  
**moreover** **have**  $R.sources \ v' = R.sources \ w'$   
**by** (*metis* (*no-types*, *lifting*) *assms*(4) *R.coinitial-iff* *R.con-imp-coinitial*  
*Cong<sub>0</sub>-imp-con* *R.arr-resid-iff-con*  $ww' \ xx'$ )  
**moreover** **have**  $R.targets \ w = R.targets \ w'$   
**by** (*metis* *Cong<sub>0</sub>-implies-Cong* *Cong<sub>0</sub>-imp-coinitial* *Cong-imp-arr*(1)  
*R.arr-resid-iff-con* *R.sources-resid*  $ww'$ )  
**ultimately** **show** *?thesis*  
**using** *assms*  $vv' \ ww'$   
**by** (*intro coherent'* [*of*  $v \ v' \ w \ w' \ t$ ]) *auto*  
**qed**  
**have**  $2$ :  $t' \setminus w' \frown u' \setminus w'$   
**using** *assms*  $1 \ ww'$   
**by** (*metis* *Cong<sub>0</sub>-subst-left*(1) *Cong<sub>0</sub>-subst-right*(1)  
*Resid-along-normal-preserves-reflects-con* *R.arr-resid-iff-con*  
*R.coinitial-iff* *R.con-imp-coinitial* *elements-are-arr*)  
**thus**  $3$ :  $t' \frown u'$   
**using**  $ww' \ R.cube$  **by** *force*  
**have**  $t \setminus u \approx ((t \setminus u) \setminus (w \setminus u)) \setminus (?y' \setminus ?y)$   
**proof** –  
**have**  $t \setminus u \approx (t \setminus u) \setminus (w \setminus u)$   
**by** (*metis* *Cong-closure-props*(4) *assms*(3) *R.con-imp-coinitial*  
*elements-are-arr* *forward-stable* *R.arr-resid-iff-con* *R.con-implies-arr*(1)  
*R.sources-resid*  $ww'$ )  
**also** **have**  $\dots \approx ((t \setminus u) \setminus (w \setminus u)) \setminus (?y' \setminus ?y)$   
**by** (*metis* *Cong<sub>0</sub>-imp-con* *Cong-closure-props*(4) *Cong-imp-arr*(2)  
*R.arr-resid-iff-con* *calculation* *R.con-implies-arr*(2) *R.targets-resid-sym*  
*R.sources-resid*  $ww'$ )  
**finally** **show** *?thesis* **by** *simp*  
**qed**  
**also** **have**  $\dots \approx (((t \setminus w) \setminus ?y) \setminus (?y' \setminus ?y))$   
**using**  $ww'$   
**by** (*metis* *Cong-imp-arr*(2) *Cong-reflexive* *calculation* *R.cube*)  
**also** **have**  $\dots \approx (((t' \setminus w') \setminus ?y) \setminus (?y' \setminus ?y))$   
**using**  $1 \ Cong_0\text{-subst-left}(2)$  [*of*  $t \setminus w \ (t' \setminus w')$   $?y$ ]  
*Cong<sub>0</sub>-subst-left*(2) [*of*  $(t \setminus w) \setminus ?y \ (t' \setminus w') \setminus ?y \ ?y' \setminus ?y$ ]

by (*meson* 2 *Cong<sub>0</sub>-implies-Cong* *Cong<sub>0</sub>-subst-Con* *Cong-imp-arr*(2)  
     *R.arr-resid-iff-con* *calculation* *ww'*)  
 also have ...  $\approx ((t' \setminus w') \setminus ?y') \setminus (?y \setminus ?y')$   
 using 2 *Cong<sub>0</sub>-implies-Cong* *Cong<sub>0</sub>-subst-right*(2) *ww'* by *presburger*  
 also have 4: ...  $\approx (t' \setminus u') \setminus (w' \setminus u')$   
 using 2 *ww'*  
 by (*metis* *Cong<sub>0</sub>-imp-con* *Cong-closure-props*(4) *Cong-symmetric* *R.cube* *R.sources-resid*)  
 also have ...  $\approx t' \setminus u'$   
 using *ww'* 3 4  
 by (*metis* *Cong-closure-props*(4) *Cong-imp-arr*(2) *Cong-symmetric* *R.con-imp-coinitial*  
     *R.con-implies-arr*(2) *forward-stable* *R.sources-resid* *R.arr-resid-iff-con*)  
 finally show  $t \setminus u \approx t' \setminus u'$  by *simp*  
 qed

**lemma** *Cong-subst-con*:  
 assumes *R.sources*  $t = R.sources$   $u$  and *R.sources*  $t' = R.sources$   $u'$   
 and  $t \approx t'$  and  $u \approx u'$   
 shows  $t \frown u \longleftrightarrow t' \frown u'$   
 using *assms* by (*meson* *Cong-subst*(1) *Cong-symmetric*)

**lemma** *Cong<sub>0</sub>-composite-of-arr-normal*:  
 assumes *R.composite-of*  $t$   $u$   $t'$  and  $u \in \mathfrak{N}$   
 shows  $t' \approx_0 t$   
 using *assms* *backward-stable* *R.composite-of-def* *ide-closed* by *blast*

**lemma** *Cong-composite-of-normal-arr*:  
 assumes *R.composite-of*  $u$   $t$   $t'$  and  $u \in \mathfrak{N}$   
 shows  $t' \approx t$   
 using *assms*  
 by (*meson* *Cong-closure-props*(2-4) *R.arr-composite-of* *ide-closed* *R.composite-of-def*  
     *R.sources-composite-of*)

end

**context** *normal-sub-rts*  
**begin**

Coherence is not an arbitrary property: here we show that substitutivity of congruence in residuation is equivalent to the “opposing spans” form of coherence.

**lemma** *Cong-subst-iff-coherent'*:  
 shows  $(\forall t t' u u'. t \approx t' \wedge u \approx u' \wedge t \frown u \wedge R.sources$   $t' = R.sources$   $u'$   
      $\longrightarrow t' \frown u' \wedge t \setminus u \approx t' \setminus u')$   
 $\longleftrightarrow$   
 $(\forall t t' v v' w w'. v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge$   
      $R.sources$   $v = R.sources$   $w \wedge R.sources$   $v' = R.sources$   $w' \wedge$   
      $R.targets$   $w = R.targets$   $w' \wedge t \setminus v \approx_0 t' \setminus v'$   
      $\longrightarrow t \setminus w \approx_0 t' \setminus w')$

**proof**

assume 1:  $\forall t t' u u'. t \approx t' \wedge u \approx u' \wedge t \frown u \wedge R.sources$   $t' = R.sources$   $u'$

$\longrightarrow t' \frown u' \wedge t \setminus u \approx t' \setminus u'$

**show**  $\forall t t' v v' w w'. v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge$   
 $R.sources\ v = R.sources\ w \wedge R.sources\ v' = R.sources\ w' \wedge$   
 $R.targets\ w = R.targets\ w' \wedge t \setminus v \approx_0 t' \setminus v'$   
 $\longrightarrow t \setminus w \approx_0 t' \setminus w'$

**proof** (*intro allI impI, elim conjE*)  
**fix**  $t t' v v' w w'$   
**assume**  $v: v \in \mathfrak{N}$  **and**  $v': v' \in \mathfrak{N}$  **and**  $w: w \in \mathfrak{N}$  **and**  $w': w' \in \mathfrak{N}$   
**and**  $sources-vw: R.sources\ v = R.sources\ w$   
**and**  $sources-v'w': R.sources\ v' = R.sources\ w'$   
**and**  $targets-ww': R.targets\ w = R.targets\ w'$   
**and**  $tt': (t \setminus v) \setminus (t' \setminus v') \in \mathfrak{N}$  **and**  $t't': (t' \setminus v') \setminus (t \setminus v) \in \mathfrak{N}$   
**show**  $t \setminus w \approx_0 t' \setminus w'$   
**proof** –  
**have**  $2: \wedge t t' u u'. \llbracket t \approx t'; u \approx u'; t \frown u; R.sources\ t' = R.sources\ u \rrbracket$   
 $\implies t' \frown u' \wedge t \setminus u \approx t' \setminus u'$   
**using** 1 **by** *blast*  
**have**  $3: t \setminus w \approx t \setminus v \wedge t' \setminus w' \approx t' \setminus v'$   
**by** (*metis tt' t't sources-vw sources-v'w' Cong<sub>0</sub>-subst-right(2)*)  
*Cong-closure-props(4) Cong-def R.arr-resid-iff-con Cong-closure-props(3)*  
*Cong-imp-arr(1) normal-is-Cong-closed v w v' w'*  
**have**  $(t \setminus w) \setminus (t' \setminus w') \approx (t \setminus v) \setminus (t' \setminus v')$   
**using** 2 [*of t \setminus w t \setminus v t' \setminus w' t' \setminus v'*] 3  
**by** (*metis tt' t't targets-ww' 1 Cong<sub>0</sub>-imp-con Cong-imp-arr(1) Cong-symmetric*)  
*R.arr-resid-iff-con R.sources-resid*  
**moreover** **have**  $(t' \setminus w') \setminus (t \setminus w) \approx (t' \setminus v') \setminus (t \setminus v)$   
**using** 2 3  
**by** (*metis tt' t't targets-ww' Cong<sub>0</sub>-imp-con Cong-symmetric*)  
*Cong-imp-arr(1) R.arr-resid-iff-con R.sources-resid*  
**ultimately** **show** *?thesis*  
**by** (*meson tt' t't normal-is-Cong-closed Cong-symmetric*)  
**qed**  
**qed**  
**next**  
**assume**  $1: \forall t t' v v' w w'. v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge$   
 $R.sources\ v = R.sources\ w \wedge R.sources\ v' = R.sources\ w' \wedge$   
 $R.targets\ w = R.targets\ w' \wedge t \setminus v \approx_0 t' \setminus v'$   
 $\longrightarrow t \setminus w \approx_0 t' \setminus w'$   
**show**  $\forall t t' u u'. t \approx t' \wedge u \approx u' \wedge t \frown u \wedge R.sources\ t' = R.sources\ u'$   
 $\longrightarrow t' \frown u' \wedge t \setminus u \approx t' \setminus u'$

**proof** (*intro allI impI, elim conjE, intro conjI*)  
**have**  $*$ :  $\wedge t t' v v' w w'. \llbracket v \in \mathfrak{N}; v' \in \mathfrak{N}; w \in \mathfrak{N}; w' \in \mathfrak{N};$   
 $R.sources\ v = R.sources\ w; R.sources\ v' = R.sources\ w';$   
 $R.targets\ v = R.targets\ v'; R.targets\ w = R.targets\ w';$   
 $t \setminus v \approx_0 t' \setminus v' \rrbracket$   
 $\implies t \setminus w \approx_0 t' \setminus w'$

**using** 1 **by** *metis*  
**fix**  $t t' u u'$   
**assume**  $tt': t \approx t'$  **and**  $uu': u \approx u'$  **and**  $con: t \frown u$

**and**  $t'u'$ :  $R.sources\ t' = R.sources\ u'$   
**obtain**  $v\ v'$  **where**  $vv'$ :  $v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge t \setminus v \approx_0 t' \setminus v'$   
**using**  $tt'$  **by** *auto*  
**obtain**  $w\ w'$  **where**  $ww'$ :  $w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge u \setminus w \approx_0 u' \setminus w'$   
**using**  $uu'$  **by** *auto*  
**let**  $?x = t \setminus v$  **and**  $?x' = t' \setminus v'$   
**let**  $?y = u \setminus w$  **and**  $?y' = u' \setminus w'$   
**have**  $xx'$ :  $?x \approx_0 ?x'$   
**using**  $tt'\ vv'$  **by** *blast*  
**have**  $yy'$ :  $?y \approx_0 ?y'$   
**using**  $uu'\ ww'$  **by** *blast*  
**have**  $1$ :  $t \setminus w \approx_0 t' \setminus w'$   
**proof** –  
**have**  $R.sources\ v = R.sources\ w \wedge R.sources\ v' = R.sources\ w'$   
**proof**  
**show**  $R.sources\ v' = R.sources\ w'$   
**using**  $Cong_0\text{-imp-con}\ R.arr\text{-resid-iff-con}\ R.coinitial\text{-iff}\ R.con\text{-imp-coinitial}$   
 $t'u'\ vv'\ ww'$   
**by** *metis*  
**show**  $R.sources\ v = R.sources\ w$   
**by** (*metis con elements-are-arr R.not-arr-null R.null-is-zero(2) R.conI*  
 $R.con\text{-imp-common-source}\ rts.sources\text{-eqI}\ R.rts\text{-axioms}\ vv'\ ww'$ )  
**qed**  
**moreover have**  $R.targets\ v = R.targets\ v' \wedge R.targets\ w = R.targets\ w'$   
**by** (*metis Cong<sub>0</sub>-imp-coinitial Cong<sub>0</sub>-imp-con R.arr-resid-iff-con*  
 $R.con\text{-implies-arr}(2)\ R.sources\text{-resid}\ vv'\ ww'$ )  
**ultimately show**  $?thesis$   
**using**  $vv'\ ww'\ xx'$   
**by** (*intro \* [of v v' w w' t t'] auto*)  
**qed**  
**have**  $2$ :  $t' \setminus w' \frown u' \setminus w'$   
**using**  $1\ tt'\ ww'$   
**by** (*meson Cong<sub>0</sub>-imp-con Cong<sub>0</sub>-subst-Con R.arr-resid-iff-con con R.con-imp-coinitial*  
 $R.con\text{-implies-arr}(2)\ resid\text{-along-elem-preserves-con}$ )  
**thus**  $3$ :  $t' \frown u'$   
**using**  $ww'$   $R.cube$  **by** *force*  
**have**  $t \setminus u \approx (t \setminus u) \setminus (w \setminus u)$   
**by** (*metis Cong-closure-props(4) R.arr-resid-iff-con con R.con-imp-coinitial*  
 $elements\text{-are-arr}\ forward\text{-stable}\ R.con\text{-implies-arr}(2)\ R.sources\text{-resid}\ ww'$ )  
**also have**  $(t \setminus u) \setminus (w \setminus u) \approx ((t \setminus u) \setminus (w \setminus u)) \setminus (?y' \setminus ?y)$   
**using**  $yy'$   
**by** (*metis Cong<sub>0</sub>-imp-con Cong-closure-props(4) Cong-imp-arr(2)*  
 $R.arr\text{-resid-iff-con}\ calculation\ R.con\text{-implies-arr}(2)\ R.sources\text{-resid}\ R.targets\text{-resid-sym}$ )  
**also have**  $\dots \approx (((t \setminus w) \setminus ?y) \setminus (?y' \setminus ?y))$   
**using**  $ww'$   
**by** (*metis Cong-imp-arr(2) Cong-reflexive calculation R.cube*)  
**also have**  $\dots \approx (((t' \setminus w') \setminus ?y) \setminus (?y' \setminus ?y))$   
**proof** –  
**have**  $((t \setminus w) \setminus ?y) \setminus (?y' \setminus ?y) \approx_0 ((t' \setminus w') \setminus ?y) \setminus (?y' \setminus ?y)$

```

    using 1 2 Cong0-subst-left(2)
    by (meson Cong0-subst-Con calculation Cong-imp-arr(2) R.arr-resid-iff-con ww')
  thus ?thesis
    using Cong0-implies-Cong by presburger
qed
also have ... ≈ ((t' \ w') \ ?y') \ (?y \ ?y')
  by (meson 2 Cong0-implies-Cong Cong0-subst-right(2) ww')
also have 4: ... ≈ (t' \ u') \ (w' \ u')
  using 2 ww'
  by (metis Cong0-imp-con Cong-closure-props(4) Cong-symmetric R.cube R.sources-resid)
also have ... ≈ t' \ u'
  using ww' 2 3 4
  by (metis Cong'.intros(1) Cong'.intros(4) Cong-char Cong-imp-arr(2)
      R.arr-resid-iff-con forward-stable R.con-imp-coinitial R.sources-resid
      R.con-implies-arr(2))
  finally show t \ u ≈ t' \ u' by simp
qed
qed
end

```

### 2.3.6 Quotient by Coherent Normal Sub-RTS

We now define the quotient of an RTS by a coherent normal sub-RTS and show that it is an extensional RTS.

```

locale quotient-by-coherent-normal =
  R: rts +
  N: coherent-normal-sub-rts
begin

  definition Resid (infix ⟨\⟩ 70)
  where  $\mathcal{T} \{\!\|\} \mathcal{U} \equiv$ 
    if  $N.is-Cong-class \mathcal{T} \wedge N.is-Cong-class \mathcal{U} \wedge (\exists t u. t \in \mathcal{T} \wedge u \in \mathcal{U} \wedge t \frown u)$ 
    then  $N.Cong-class$ 
      (fst (SOME tu. fst tu  $\in \mathcal{T} \wedge snd tu \in \mathcal{U} \wedge fst tu \frown snd tu) \setminus$ 
       snd (SOME tu. fst tu  $\in \mathcal{T} \wedge snd tu \in \mathcal{U} \wedge fst tu \frown snd tu))$ 
    else {}

  sublocale partial-magma Resid
    using N.Cong-class-is-nonempty Resid-def
    by unfold-locales metis

  lemma is-partial-magma:
  shows partial-magma Resid
  ..

  lemma null-char:
  shows null = {}
    using N.Cong-class-is-nonempty Resid-def

```

by (*metis null-is-zero(2)*)

**lemma** *Resid-by-members*:

**assumes** *N.is-Cong-class*  $\mathcal{T}$  **and** *N.is-Cong-class*  $\mathcal{U}$  **and**  $t \in \mathcal{T}$  **and**  $u \in \mathcal{U}$  **and**  $t \frown u$   
**shows**  $\mathcal{T} \{\!\!\backslash\!\!\} \mathcal{U} = \{t \setminus u\}$   
**using** *assms Resid-def someI-ex* [of  $\lambda tu. \text{fst } tu \in \mathcal{T} \wedge \text{snd } tu \in \mathcal{U} \wedge \text{fst } tu \frown \text{snd } tu$ ]  
**apply** *simp*  
**by** (*meson N.Cong-class-membs-are-Cong N.Cong-class-eqI N.Cong-subst(2)*  
*R.coinitial-iff R.con-imp-coinitial*)

**abbreviation** *Con* (**infix**  $\langle\!\!\frown\!\!\rangle$  50)

**where**  $\mathcal{T} \{\!\!\frown\!\!\} \mathcal{U} \equiv \mathcal{T} \{\!\!\backslash\!\!\} \mathcal{U} \neq \{\}$

**lemma** *Con-char*:

**shows**  $\mathcal{T} \{\!\!\frown\!\!\} \mathcal{U} \longleftrightarrow$   
 $N.is-Cong-class \mathcal{T} \wedge N.is-Cong-class \mathcal{U} \wedge (\exists t u. t \in \mathcal{T} \wedge u \in \mathcal{U} \wedge t \frown u)$   
**by** (*metis (no-types, opaque-lifting) N.Cong-class-is-nonempty N.is-Cong-classI*  
*Resid-def Resid-by-members R.arr-resid-iff-con*)

**lemma** *Con-sym*:

**assumes** *Con*  $\mathcal{T} \mathcal{U}$   
**shows** *Con*  $\mathcal{U} \mathcal{T}$   
**using** *assms Con-char R.con-sym* **by** *meson*

**lemma** *is-Cong-class-Resid*:

**assumes**  $\mathcal{T} \{\!\!\frown\!\!\} \mathcal{U}$   
**shows** *N.is-Cong-class* ( $\mathcal{T} \{\!\!\backslash\!\!\} \mathcal{U}$ )  
**using** *assms Con-char Resid-by-members R.arr-resid-iff-con N.is-Cong-classI* **by** *auto*

**lemma** *Con-witnesses*:

**assumes**  $\mathcal{T} \{\!\!\frown\!\!\} \mathcal{U}$  **and**  $t \in \mathcal{T}$  **and**  $u \in \mathcal{U}$   
**shows**  $\exists v w. v \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge t \setminus v \frown u \setminus w$   
**proof** –  
**have** 1:  $N.is-Cong-class \mathcal{T} \wedge N.is-Cong-class \mathcal{U} \wedge (\exists t u. t \in \mathcal{T} \wedge u \in \mathcal{U} \wedge t \frown u)$   
**using** *assms Con-char* **by** *simp*  
**obtain**  $t' u'$  **where**  $t' u'$ :  $t' \in \mathcal{T} \wedge u' \in \mathcal{U} \wedge t' \frown u'$   
**using** 1 **by** *auto*  
**have** 2:  $t' \approx t \wedge u' \approx u$   
**using** *assms 1 t'u' N.Cong-class-membs-are-Cong* **by** *auto*  
**obtain**  $v v'$  **where**  $vv'$ :  $v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge t' \setminus v \approx_0 t \setminus v'$   
**using** 2 **by** *auto*  
**obtain**  $w w'$  **where**  $ww'$ :  $w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge u' \setminus w \approx_0 u \setminus w'$   
**using** 2 **by** *auto*  
**have** 3:  $w \frown v$   
**by** (*metis R.arr-resid-iff-con R.con-def R.con-imp-coinitial R.ex-un-null*  
*N.elements-are-arr R.null-is-zero(2) N.resid-along-elem-preserves-con t'u' vv' ww'*)  
**have** *R.seq*  $v$  ( $w \setminus v$ )  
**by** (*simp add: N.elements-are-arr R.seq-def 3 vv'*)  
**obtain**  $x$  **where**  $x$ : *R.composite-of*  $v$  ( $w \setminus v$ )  $x$

**using**  $N.composite-closed-left \langle R.seq\ v\ (w \setminus v) \rangle\ vv'$  **by** *blast*  
**obtain**  $x'$  **where**  $x': R.composite-of\ v'\ (w \setminus v)\ x'$   
**using**  $x\ vv'$   $N.composite-closed-left$   
**by** (*metis*  $N.Cong_0-implies-Cong\ N.Cong_0-imp-coinitial\ N.Cong-imp-arr(1)$   
 $R.composable-def\ R.composable-imp-seq\ R.con-implies-arr(2)$   
 $R.seq-def\ R.sources-resid\ R.arr-resid-iff-con$ )  
**have**  $*$ :  $t' \setminus x \approx_0 t \setminus x'$   
**by** (*metis*  $N.coherent'\ N.composite-closed\ N.forward-stable\ R.con-imp-coinitial$   
 $R.targets-composite-of\ \exists\ R.con-sym\ R.sources-composite-of\ vv'\ ww'\ x\ x'$ )  
**obtain**  $y$  **where**  $y: R.composite-of\ w\ (v \setminus w)\ y$   
**using**  $x\ vv'\ ww'$   
**by** (*metis*  $R.arr-resid-iff-con\ R.composable-def\ R.composable-imp-seq$   
 $R.con-imp-coinitial\ R.seq-def\ R.sources-resid\ N.elements-are-arr$   
 $N.forward-stable\ N.composite-closed-left$ )  
**obtain**  $y'$  **where**  $y': R.composite-of\ w'\ (v \setminus w)\ y'$   
**using**  $y\ ww'$   
**by** (*metis*  $N.Cong_0-imp-coinitial\ N.Cong-closure-props(3)\ N.Cong-imp-arr(1)$   
 $R.composable-def\ R.composable-imp-seq\ R.con-implies-arr(2)\ R.seq-def$   
 $R.sources-resid\ N.composite-closed-left\ R.arr-resid-iff-con$ )  
**have**  $**$ :  $u' \setminus y \approx_0 u \setminus y'$   
**by** (*metis*  $N.composite-closed\ N.forward-stable\ R.con-imp-coinitial\ R.targets-composite-of$   
 $\langle w \frown v \rangle\ N.coherent'\ R.sources-composite-of\ vv'\ ww'\ y\ y'$ )  
**have**  $\downarrow$ :  $x \in \mathfrak{N} \wedge y \in \mathfrak{N}$   
**using**  $x\ y\ vv'\ ww'\ * \ **$   
**by** (*metis*  $\exists\ N.composite-closed\ N.forward-stable\ R.con-imp-coinitial\ R.con-sym$ )  
**have**  $t \setminus x' \frown u \setminus y'$   
**proof** –  
**have**  $t \setminus x' \approx_0 t' \setminus x$   
**using**  $*$  **by** *simp*  
**moreover** **have**  $t' \setminus x \frown u' \setminus y$   
**proof** –  
**have**  $t' \setminus x \frown u' \setminus x$   
**using**  $t'u'\ vv'\ ww'\ \downarrow\ *$   
**by** (*metis*  $N.Resid-along-normal-preserved-reflects-con\ N.elements-are-arr$   
 $R.coinitial-iff\ R.con-imp-coinitial\ R.arr-resid-iff-con$ )  
**moreover** **have**  $u' \setminus x \approx_0 u' \setminus y$   
**using**  $ww'\ x\ y$   
**by** (*metis*  $\downarrow\ N.Cong_0-imp-coinitial\ N.Cong_0-imp-con\ N.Cong_0-transitive$   
 $N.coherent'\ N.factor-closed(2)\ R.sources-composite-of$   
 $R.targets-composite-of\ R.targets-resid-sym$ )  
**ultimately** **show** *?thesis*  
**using**  $N.Cong_0-subst-right$  **by** *blast*  
**qed**  
**moreover** **have**  $u' \setminus y \approx_0 u \setminus y'$   
**using**  $**\ R.con-sym$  **by** *simp*  
**ultimately** **show** *?thesis*  
**using**  $N.Cong_0-subst-Con$  **by** *auto*  
**qed**  
**moreover** **have**  $x' \in \mathfrak{N} \wedge y' \in \mathfrak{N}$

**using**  $x' y' vv' ww'$   
**by** (*metis*  $N.Cong\text{-}composite\text{-}of\text{-}normal\text{-}arr$   $N.Cong\text{-}imp\text{-}arr(2)$   $N.composite\text{-}closed$   
 $R.con\text{-}imp\text{-}cointial$   $N.forward\text{-}stable$   $R.arr\text{-}resid\text{-}iff\text{-}con$ )  
**ultimately show** *?thesis* **by** *auto*  
**qed**

**abbreviation**  $Arr$   
**where**  $Arr \mathcal{T} \equiv Con \mathcal{T} \mathcal{T}$

**lemma**  $Arr\text{-}Resid$ :  
**assumes**  $Con \mathcal{T} \mathcal{U}$   
**shows**  $Arr (\mathcal{T} \{\!\!\}\ \mathcal{U})$   
**by** (*metis*  $Con\text{-}char$   $N.Cong\text{-}class\text{-}memb\text{-}is\text{-}arr$   $R.arrE$   $N.rep\text{-}in\text{-}Cong\text{-}class$   
 $assms$   $is\text{-}Cong\text{-}class\text{-}Resid$ )

**lemma**  $Cube$ :  
**assumes**  $Con (\mathcal{V} \{\!\!\}\ \mathcal{T}) (\mathcal{U} \{\!\!\}\ \mathcal{T})$   
**shows**  $(\mathcal{V} \{\!\!\}\ \mathcal{T}) \{\!\!\}\ (\mathcal{U} \{\!\!\}\ \mathcal{T}) = (\mathcal{V} \{\!\!\}\ \mathcal{U}) \{\!\!\}\ (\mathcal{T} \{\!\!\}\ \mathcal{U})$   
**proof** –  
**obtain**  $t u$  **where**  $tu$ :  $t \in \mathcal{T} \wedge u \in \mathcal{U} \wedge t \frown u \wedge \mathcal{T} \{\!\!\}\ \mathcal{U} = \{\!\!\}t \ \ u\{\!\!\}$   
**using**  $assms$   
**by** (*metis*  $Con\text{-}char$   $N.Cong\text{-}class\text{-}is\text{-}nonempty$   $R.con\text{-}sym$   $Resid\text{-}by\text{-}members$ )  
**obtain**  $t' v$  **where**  $t'v$ :  $t' \in \mathcal{T} \wedge v \in \mathcal{V} \wedge t' \frown v \wedge \mathcal{T} \{\!\!\}\ \mathcal{V} = \{\!\!\}t' \ \ v\{\!\!\}$   
**using**  $assms$   
**by** (*metis*  $Con\text{-}char$   $N.Cong\text{-}class\text{-}is\text{-}nonempty$   $Resid\text{-}by\text{-}members$   $Con\text{-}sym$ )  
**have**  $tt'$ :  $t \approx t'$   
**using**  $assms$   
**by** (*metis*  $N.Cong\text{-}class\text{-}membs\text{-}are\text{-}Cong$   $N.Cong\text{-}class\text{-}is\text{-}nonempty$   $Resid\text{-}def$   $t'v$   $tu$ )  
**obtain**  $w w'$  **where**  $ww'$ :  $w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge t \setminus w \approx_0 t' \setminus w'$   
**using**  $tu$   $t'v$   $tt'$  **by** *auto*  
**have**  $1$ :  $\mathcal{U} \{\!\!\}\ \mathcal{T} = \{\!\!\}u \ \ t\{\!\!\} \wedge \mathcal{V} \{\!\!\}\ \mathcal{T} = \{\!\!\}v \ \ t'\{\!\!\}$   
**by** (*metis*  $Con\text{-}char$   $N.Cong\text{-}class\text{-}is\text{-}nonempty$   $R.con\text{-}sym$   $Resid\text{-}by\text{-}members$   $assms$   $t'v$   $tu$ )  
**obtain**  $x x'$  **where**  $xx'$ :  $x \in \mathfrak{N} \wedge x' \in \mathfrak{N} \wedge (u \setminus t) \setminus x \frown (v \setminus t') \setminus x'$   
**using**  $1$   $Con\text{-}witnesses$  [*of*  $\mathcal{U} \{\!\!\}\ \mathcal{T}$   $\mathcal{V} \{\!\!\}\ \mathcal{T}$   $u \setminus t$   $v \setminus t'$ ]  
**by** (*metis*  $N.arr\text{-}in\text{-}Cong\text{-}class$   $R.con\text{-}sym$   $t'v$   $tu$   $assms$   $Con\text{-}sym$   $R.arr\text{-}resid\text{-}iff\text{-}con$ )  
**have**  $R.seq$   $t$   $x$   
**by** (*metis*  $R.arr\text{-}resid\text{-}iff\text{-}con$   $R.cointial\text{-}iff$   $R.con\text{-}imp\text{-}cointial$   $R.seqI(2)$   
 $R.sources\text{-}resid$   $xx'$ )  
**have**  $R.seq$   $t'$   $x'$   
**by** (*metis*  $R.arr\text{-}resid\text{-}iff\text{-}con$   $R.sources\text{-}resid$   $R.cointialE$   $R.con\text{-}imp\text{-}cointial$   
 $R.seqI(2)$   $xx'$ )  
**obtain**  $tx$  **where**  $tx$ :  $R.composite\text{-}of$   $t$   $x$   $tx$   
**using**  $xx'$   $\langle R.seq$   $t$   $x \rangle$   $N.composite\text{-}closed\text{-}right$  [*of*  $x$   $t$ ]  $R.composable\text{-}def$  **by** *auto*  
**obtain**  $t'x'$  **where**  $t'x'$ :  $R.composite\text{-}of$   $t'$   $x'$   $t'x'$   
**using**  $xx'$   $\langle R.seq$   $t'$   $x' \rangle$   $N.composite\text{-}closed\text{-}right$  [*of*  $x'$   $t'$ ]  $R.composable\text{-}def$  **by** *auto*  
**let**  $?tx\text{-}w = tx \setminus w$  **and**  $?t'x'\text{-}w' = t'x' \setminus w'$   
**let**  $?w\text{-}tx = (w \setminus t) \setminus x$  **and**  $?w'\text{-}t'x' = (w' \setminus t') \setminus x'$   
**let**  $?u\text{-}tx = (u \setminus t) \setminus x$  **and**  $?v\text{-}t'x' = (v \setminus t') \setminus x'$   
**let**  $?u\text{-}w = u \setminus w$  **and**  $?v\text{-}w' = v \setminus w'$

**let**  $?w-u = w \setminus u$  **and**  $?w'-v = w' \setminus v$   
**have**  $w-tx$ -in- $\mathfrak{N}$ :  $?w-tx \in \mathfrak{N}$   
**using**  $tx$   $ww'$   $xx'$   $R$ .con-composite-of-iff [of  $t$   $x$   $tx$   $w$ ]  
**by** (*metis* (*full-types*)  $N$ .Cong<sub>0</sub>-composite-of-arr-normal  $N$ .Cong<sub>0</sub>-subst-left(1)  
 $N$ .forward-stable  $R$ .null-is-zero(2)  $R$ .con-imp-coinitial  $R$ .conI  $R$ .con-sym)  
**have**  $w'-t'x'$ -in- $\mathfrak{N}$ :  $?w'-t'x' \in \mathfrak{N}$   
**using**  $t'x'$   $ww'$   $xx'$   $R$ .con-composite-of-iff [of  $t'$   $x'$   $t'x'$   $w'$ ]  
**by** (*metis* (*full-types*)  $N$ .Cong<sub>0</sub>-composite-of-arr-normal  $N$ .Cong<sub>0</sub>-subst-left(1)  
 $R$ .con-sym  $N$ .forward-stable  $R$ .null-is-zero(2)  $R$ .con-imp-coinitial  $R$ .conI)  
**have** 2:  $?tx-w \approx_0 ?t'x'-w'$   
**proof** –  
**have**  $?tx-w \approx_0 t \setminus w$   
**using**  $t'x'$   $tx$   $ww'$   $xx'$   $N$ .Cong<sub>0</sub>-composite-of-arr-normal [of  $t$   $x$   $tx$ ]  $N$ .Cong<sub>0</sub>-subst-left(2)  
**by** (*metis*  $N$ .Cong<sub>0</sub>-transitive  $R$ .conI)  
**also have**  $t \setminus w \approx_0 t' \setminus w'$   
**using**  $ww'$  **by** *blast*  
**also have**  $t' \setminus w' \approx_0 ?t'x'-w'$   
**using**  $t'x'$   $tx$   $ww'$   $xx'$   $N$ .Cong<sub>0</sub>-composite-of-arr-normal [of  $t'$   $x'$   $t'x'$ ]  $N$ .Cong<sub>0</sub>-subst-left(2)  
**by** (*metis*  $N$ .Cong<sub>0</sub>-transitive  $R$ .conI)  
**finally show** *?thesis* **by** *blast*  
**qed**  
**obtain**  $z$  **where**  $z$ :  $R$ .composite-of  $?tx-w$  ( $?t'x'-w' \setminus ?tx-w$ )  $z$   
**by** (*metis* 2  $R$ .arr-resid-iff-con  $R$ .con-implies-arr(2)  $N$ .elements-are-arr  
 $N$ .composite-closed-right  $R$ .seqI(1)  $R$ .sources-resid)  
**obtain**  $z'$  **where**  $z'$ :  $R$ .composite-of  $?t'x'-w'$  ( $?tx-w \setminus ?t'x'-w'$ )  $z'$   
**by** (*metis* 2  $R$ .arr-resid-iff-con  $R$ .con-implies-arr(2)  $N$ .elements-are-arr  
 $N$ .composite-closed-right  $R$ .seqI(1)  $R$ .sources-resid)  
**have** 3:  $z \approx_0 z'$   
**using** 2  $N$ .diamond-commutes-upto-Cong<sub>0</sub>  $N$ .Cong<sub>0</sub>-imp-con  $z$   $z'$  **by** *blast*  
**have**  $R$ .targets  $z = R$ .targets  $z'$   
**by** (*metis*  $R$ .targets-resid-sym  $z$   $z'$   $R$ .targets-composite-of  $R$ .conI)  
**have**  $Con$ - $z$ - $uw$ :  $z \frown ?u-w$   
**proof** –  
**have**  $?tx-w \frown ?u-w$   
**by** (*meson* 3  $N$ .Cong<sub>0</sub>-composite-of-arr-normal  $N$ .Cong<sub>0</sub>-subst-left(1)  
 $R$ .bounded-imp-con  $R$ .con-implies-arr(1)  $R$ .con-imp-coinitial  
 $N$ .resid-along-elem-preserves-con  $tu$   $tx$   $ww'$   $xx'$   $z$   $z'$   $R$ .arr-resid-iff-con)  
**thus** *?thesis*  
**using** 2  $N$ .Cong<sub>0</sub>-composite-of-arr-normal  $N$ .Cong<sub>0</sub>-subst-left(1)  $z$  **by** *blast*  
**qed**  
**moreover have**  $Con$ - $z'$ - $vw'$ :  $z' \frown ?v-w'$   
**proof** –  
**have**  $?t'x'-w' \frown ?v-w'$   
**by** (*meson* 3  $N$ .Cong<sub>0</sub>-composite-of-arr-normal  $N$ .Cong<sub>0</sub>-subst-left(1)  
 $R$ .bounded-imp-con  $t'v$   $t'x'$   $ww'$   $xx'$   $z$   $z'$   $R$ .con-imp-coinitial  
 $N$ .resid-along-elem-preserves-con  $R$ .arr-resid-iff-con  $R$ .con-implies-arr(1))  
**thus** *?thesis*  
**by** (*meson* 2  $N$ .Cong<sub>0</sub>-composite-of-arr-normal  $N$ .Cong<sub>0</sub>-subst-left(1)  $z'$ )  
**qed**

**moreover have**  $Con\text{-}z\text{-}vw'$ :  $z \frown ?v\text{-}w'$   
**using**  $\mathcal{B}$   $Con\text{-}z'\text{-}vw'$   $N.Cong_0\text{-}subst\text{-}left(1)$  **by** *blast*  
**moreover have**  $*$ :  $?u\text{-}w \setminus z \frown ?v\text{-}w' \setminus z$   
**proof** –  
**obtain**  $y$  **where**  $y$ :  $R.composite\text{-}of (w \setminus tx) (?t'x'\text{-}w' \setminus ?tx\text{-}w) y$   
**by** (*metis*  $\mathcal{B}$   $R.arr\text{-}resid\text{-}iff\text{-}con$   $R.composable\text{-}def$   $R.composable\text{-}imp\text{-}seq$   
 $R.con\text{-}imp\text{-}coinitial$   $N.elements\text{-}are\text{-}arr$   $N.composite\text{-}closed\text{-}right$   
 $R.seq\text{-}def$   $R.targets\text{-}resid\text{-}sym$   $ww' z N.forward\text{-}stable$ )  
**obtain**  $y'$  **where**  $y'$ :  $R.composite\text{-}of (w' \setminus t'x') (?tx\text{-}w \setminus ?t'x'\text{-}w') y'$   
**by** (*metis*  $\mathcal{B}$   $R.arr\text{-}resid\text{-}iff\text{-}con$   $R.composable\text{-}def$   $R.composable\text{-}imp\text{-}seq$   
 $R.con\text{-}imp\text{-}coinitial$   $N.elements\text{-}are\text{-}arr$   $N.composite\text{-}closed\text{-}right$   
 $R.targets\text{-}resid\text{-}sym$   $ww' z' R.seq\text{-}def$   $N.forward\text{-}stable$ )  
**have**  $y\text{-}comp$ :  $R.composite\text{-}of (w \setminus tx) ((t'x' \setminus w') \setminus (tx \setminus w)) y$   
**using**  $y$  **by** *simp*  
**have**  $y\text{-}in\text{-}normal$ :  $y \in \mathfrak{N}$   
**by** (*metis*  $\mathcal{B}$   $Con\text{-}z\text{-}uw$   $R.arr\text{-}iff\text{-}has\text{-}source$   $R.arr\text{-}resid\text{-}iff\text{-}con$   $N.composite\text{-}closed$   
 $R.con\text{-}imp\text{-}coinitial$   $R.con\text{-}implies\text{-}arr(1)$   $N.forward\text{-}stable$   
 $R.sources\text{-}composite\text{-}of$   $ww' y\text{-}comp z$ )  
**have**  $y\text{-}coinitial$ :  $R.coinitial y (u \setminus tx)$   
**using**  $y$   $R.arr\text{-}composite\text{-}of$   $R.sources\text{-}composite\text{-}of$   
**apply** (*intro*  $R.coinitialI$ )  
**apply** *auto*  
**apply** (*metis*  $N.Cong_0\text{-}composite\text{-}of\text{-}arr\text{-}normal$   $N.Cong_0\text{-}subst\text{-}right(1)$   
 $R.composite\text{-}of\text{-}cancel\text{-}left$   $R.con\text{-}sym$   $R.not\text{-}ide\text{-}null$   $R.null\text{-}is\text{-}zero(2)$   
 $R.sources\text{-}resid$   $R.conI$   $tu$   $tx$   $xx'$ )  
**by** (*metis*  $R.arr\text{-}iff\text{-}has\text{-}source$   $R.not\text{-}arr\text{-}null$   $R.sources\text{-}resid$   $empty\text{-}iff$   $R.conI$ )  
**have**  $y\text{-}con$ :  $y \frown u \setminus tx$   
**using**  $y\text{-}in\text{-}normal$   $y\text{-}coinitial$   
**by** (*metis*  $R.coinitial\text{-}iff$   $N.elements\text{-}are\text{-}arr$   $N.forward\text{-}stable$   
 $R.arr\text{-}resid\text{-}iff\text{-}con$ )  
**have**  $A$ :  $?u\text{-}w \setminus z \sim (u \setminus tx) \setminus y$   
**proof** –  
**have**  $(u \setminus tx) \setminus y \sim ((u \setminus tx) \setminus (w \setminus tx)) \setminus (?t'x'\text{-}w' \setminus ?tx\text{-}w)$   
**using**  $y\text{-}comp$   $y\text{-}con$   
 $R.resid\text{-}composite\text{-}of(\mathcal{B})$  [*of*  $w \setminus tx$   $?t'x'\text{-}w' \setminus ?tx\text{-}w$   $y$   $u \setminus tx$ ]  
**by** *simp*  
**also have**  $((u \setminus tx) \setminus (w \setminus tx)) \setminus (?t'x'\text{-}w' \setminus ?tx\text{-}w) \sim ?u\text{-}w \setminus z$   
**by** (*metis*  $Con\text{-}z\text{-}uw$   $R.resid\text{-}composite\text{-}of(\mathcal{B})$   $z$   $R.cube$ )  
**finally show**  $?thesis$  **by** *blast*  
**qed**  
**have**  $y'\text{-}comp$ :  $R.composite\text{-}of (w' \setminus t'x') (?tx\text{-}w \setminus ?t'x'\text{-}w') y'$   
**using**  $y'$  **by** *simp*  
**have**  $y'\text{-}in\text{-}normal$ :  $y' \in \mathfrak{N}$   
**by** (*metis*  $\mathcal{B}$   $Con\text{-}z'\text{-}vw'$   $R.arr\text{-}iff\text{-}has\text{-}source$   $R.arr\text{-}resid\text{-}iff\text{-}con$   
 $N.composite\text{-}closed$   $R.con\text{-}imp\text{-}coinitial$   $R.con\text{-}implies\text{-}arr(1)$   
 $N.forward\text{-}stable$   $R.sources\text{-}composite\text{-}of$   $ww' y'\text{-}comp z'$ )  
**have**  $y'\text{-}coinitial$ :  $R.coinitial y' (v \setminus t'x')$   
**using**  $y'$   $R.coinitial\text{-}def$   
**by** (*metis*  $Con\text{-}z'\text{-}vw'$   $R.arr\text{-}resid\text{-}iff\text{-}con$   $R.composite\text{-}ofE$   $R.con\text{-}imp\text{-}coinitial$ )

$R.con\text{-implies}\text{-arr}(1)$   $R.cube$   $R.prfx\text{-implies}\text{-con}$   $R.resid\text{-composite}\text{-of}(1)$   
 $R.sources\text{-resid}$   $z'$   
**have**  $y'\text{-con}$ :  $y' \frown v \setminus t'x'$   
**using**  $y'\text{-in}\text{-normal}$   $y'\text{-coinitial}$   
**by** ( $metis$   $R.coinitial\text{-iff}$   $N.elements\text{-are}\text{-arr}$   $N.forward\text{-stable}$   
 $R.arr\text{-resid}\text{-iff}\text{-con}$ )  
**have**  $B$ :  $?v\text{-}w' \setminus z' \sim (v \setminus t'x') \setminus y'$   
**proof** –  
**have**  $(v \setminus t'x') \setminus y' \sim ((v \setminus t'x') \setminus (w' \setminus t'x')) \setminus (?tx\text{-}w \setminus ?t'x'\text{-}w')$   
**using**  $y'\text{-comp}$   $y'\text{-con}$   
 $R.resid\text{-composite}\text{-of}(3)$  [ $of$   $w' \setminus t'x'$   $?tx\text{-}w \setminus ?t'x'\text{-}w'$   $y' v \setminus t'x'$ ]  
**by**  $blast$   
**also have**  $((v \setminus t'x') \setminus (w' \setminus t'x')) \setminus (?tx\text{-}w \setminus ?t'x'\text{-}w') \sim ?v\text{-}w' \setminus z'$   
**by** ( $metis$   $Con\text{-}z'\text{-}vw'$   $R.cube$   $R.resid\text{-composite}\text{-of}(3)$   $z'$ )  
**finally show**  $?thesis$  **by**  $blast$   
**qed**  
**have**  $C$ :  $u \setminus tx \frown v \setminus t'x'$   
**using**  $tx$   $t'x'$   $xx'$   $R.con\text{-sym}$   $R.cong\text{-subst}\text{-right}(1)$   $R.resid\text{-composite}\text{-of}(3)$   
**by** ( $meson$   $R.coinitial\text{-iff}$   $R.arr\text{-resid}\text{-iff}\text{-con}$   $y'\text{-coinitial}$   $y\text{-coinitial}$ )  
**have**  $D$ :  $y \approx_0 y'$   
**proof** –  
**have**  $y \approx_0 w \setminus tx$   
**using**  $2$   $N.Cong_0\text{-composite}\text{-of}\text{-arr}\text{-normal}$   $y\text{-comp}$  **by**  $blast$   
**also have**  $w \setminus tx \approx_0 w' \setminus t'x'$   
**proof** –  
**have**  $w \setminus tx \in \mathfrak{N} \wedge w' \setminus t'x' \in \mathfrak{N}$   
**using**  $N.factor\text{-closed}(1)$   $y\text{-comp}$   $y\text{-in}\text{-normal}$   $y'\text{-comp}$   $y'\text{-in}\text{-normal}$  **by**  $blast$   
**moreover have**  $R.coinitial$   $(w \setminus tx)$   $(w' \setminus t'x')$   
**by** ( $metis$   $C$   $R.coinitial\text{-def}$   $R.con\text{-implies}\text{-arr}(2)$   $N.elements\text{-are}\text{-arr}$   
 $R.sources\text{-resid}$   $calculation$   $R.con\text{-imp}\text{-coinitial}$   $R.arr\text{-resid}\text{-iff}\text{-con}$   $y\text{-con}$ )  
**ultimately show**  $?thesis$   
**by** ( $meson$   $R.arr\text{-resid}\text{-iff}\text{-con}$   $R.con\text{-imp}\text{-coinitial}$   $N.forward\text{-stable}$   
 $N.elements\text{-are}\text{-arr}$ )  
**qed**  
**also have**  $w' \setminus t'x' \approx_0 y'$   
**using**  $2$   $N.Cong_0\text{-composite}\text{-of}\text{-arr}\text{-normal}$   $y'\text{-comp}$  **by**  $blast$   
**finally show**  $?thesis$  **by**  $blast$   
**qed**  
**have**  $par\text{-}y\text{-}y'$ :  $R.sources$   $y = R.sources$   $y' \wedge R.targets$   $y = R.targets$   $y'$   
**using**  $D$   $N.Cong_0\text{-imp}\text{-coinitial}$   $R.targets\text{-composite}\text{-of}$   $y'\text{-comp}$   $y\text{-comp}$   $z$   $z'$   
 $\langle R.targets$   $z = R.targets$   $z' \rangle$   
**by**  $presburger$   
**have**  $E$ :  $(u \setminus tx) \setminus y \frown (v \setminus t'x') \setminus y'$   
**proof** –  
**have**  $(u \setminus tx) \setminus y \frown (v \setminus t'x') \setminus y$   
**using**  $C$   $N.Resid\text{-along}\text{-normal}\text{-preserves}\text{-reflects}\text{-con}$   $R.coinitial\text{-iff}$   
 $y\text{-coinitial}$   $y\text{-in}\text{-normal}$   
**by**  $presburger$   
**moreover have**  $(v \setminus t'x') \setminus y \approx_0 (v \setminus t'x') \setminus y'$

using *par-y-y' N.coherent R.coinitial-iff y'-coinitial y'-in-normal y-in-normal*  
 by *presburger*  
 ultimately show *?thesis*  
 using *N.Cong<sub>0</sub>-subst-right(1)* by *blast*  
 qed  
 hence  $?u-w \setminus z \frown ?v-w' \setminus z'$   
 proof –  
 have  $(u \setminus tx) \setminus y \sim ?u-w \setminus z$   
 using *A* by *simp*  
 moreover have  $(u \setminus tx) \setminus y \frown (v \setminus t'x') \setminus y'$   
 using *E* by *blast*  
 moreover have  $(v \setminus t'x') \setminus y' \sim ?v-w' \setminus z'$   
 using *B R.cong-symmetric* by *blast*  
 moreover have  $R.sources ((u \setminus w) \setminus z) = R.sources ((v \setminus w') \setminus z')$   
 by *(simp add: Con-z'-vw' Con-z-uw R.con-sym <R.targets z = R.targets z'>)*  
 ultimately show *?thesis*  
 by *(meson N.Cong<sub>0</sub>-subst-Con N.ide-closed)*  
 qed  
 moreover have  $?v-w' \setminus z' \approx ?v-w' \setminus z$   
 by *(meson 3 Con-z-vw' N.CongI N.Cong<sub>0</sub>-subst-right(2) R.con-sym)*  
 moreover have  $R.sources ((v \setminus w') \setminus z) = R.sources ((u \setminus w) \setminus z)$   
 by *(metis R.con-implies-arr(1) R.sources-resid calculation(1) calculation(2) N.Cong-imp-arr(2) R.arr-resid-iff-con)*  
 ultimately show *?thesis*  
 by *(metis N.Cong-reflexive N.Cong-subst(1) R.con-implies-arr(1))*  
 qed  
 ultimately have \*\*:  $?v-w' \setminus z \frown ?u-w \setminus z \wedge$   
 $(?v-w' \setminus z) \setminus (?u-w \setminus z) = (?v-w' \setminus ?u-w) \setminus (z \setminus ?u-w)$   
 by *(meson R.con-sym R.cube)*  
 have *Cong-t-z: t ≈ z*  
 by *(metis 2 N.Cong<sub>0</sub>-composite-of-arr-normal N.Cong-closure-props(2–3) N.Cong-closure-props(4) N.Cong-imp-arr(2) R.coinitial-iff R.con-imp-coinitial tx ww' xx' z R.arr-resid-iff-con)*  
 have *Cong-u-uw: u ≈ ?u-w*  
 by *(meson Con-z-uw N.Cong-closure-props(4) R.coinitial-iff R.con-imp-coinitial ww' R.arr-resid-iff-con)*  
 have *Cong-v-vw': v ≈ ?v-w'*  
 by *(meson Con-z-vw' N.Cong-closure-props(4) R.coinitial-iff ww' R.con-imp-coinitial R.arr-resid-iff-con)*  
 have *T: N.is-Cong-class T ∧ z ∈ T*  
 by *(metis (no-types, lifting) Cong-t-z N.Cong-class-eqI N.Cong-class-is-nonempty N.Cong-class-memb-Cong-rep N.Cong-class-rep N.Cong-imp-arr(2) N.arr-in-Cong-class tu assms Con-char)*  
 have *U: N.is-Cong-class U ∧ ?u-w ∈ U*  
 by *(metis Con-char Con-z-uw Cong-u-uw Int-iff N.Cong-class-eqI' N.Cong-class-eqI N.arr-in-Cong-class R.con-implies-arr(2) N.is-Cong-classI tu assms empty-iff)*  
 have *V: N.is-Cong-class V ∧ ?v-w' ∈ V*  
 by *(metis Con-char Con-z-vw' Cong-v-vw' Int-iff N.Cong-class-eqI' N.Cong-class-eqI N.arr-in-Cong-class R.con-implies-arr(2) N.is-Cong-classI t'v assms empty-iff)*

**show**  $(\mathcal{V} \{\!\!\}\mathcal{T}) \{\!\!\}\mathcal{U} \{\!\!\}\mathcal{T} = (\mathcal{V} \{\!\!\}\mathcal{U}) \{\!\!\}\mathcal{T} \{\!\!\}\mathcal{U}$   
**proof** –  
**have**  $(\mathcal{V} \{\!\!\}\mathcal{T}) \{\!\!\}\mathcal{U} \{\!\!\}\mathcal{T} = \{!(?v-w' \setminus z) \setminus (?u-w \setminus z)\}$   
**using**  $\mathcal{T} \mathcal{U} \mathcal{V} * \text{Resid-by-members}$   
**by** (*metis*  $** \text{Con-char } N.\text{arr-in-Cong-class } R.\text{arr-resid-iff-con } \text{assms } R.\text{con-implies-arr}(2)$ )  
**moreover have**  $(\mathcal{V} \{\!\!\}\mathcal{U}) \{\!\!\}\mathcal{T} \{\!\!\}\mathcal{U} = \{!(?v-w' \setminus ?u-w) \setminus (z \setminus ?u-w)\}$   
**using**  $\text{Resid-by-members [of } \mathcal{V} \mathcal{U} ?v-w' ?u-w] \text{Resid-by-members [of } \mathcal{T} \mathcal{U} z ?u-w]$   
 $\text{Resid-by-members [of } \mathcal{V} \{\!\!\}\mathcal{U} \mathcal{T} \{\!\!\}\mathcal{U} ?v-w' \setminus ?u-w z \setminus ?u-w]$   
**by** (*metis*  $\mathcal{T} \mathcal{U} \mathcal{V} * ** N.\text{arr-in-Cong-class } R.\text{con-implies-arr}(2) N.\text{is-Cong-classI}$   
 $R.\text{resid-reflects-con } R.\text{arr-resid-iff-con}$ )  
**ultimately show** *?thesis*  
**using**  $** \text{by simp}$   
**qed**  
**qed**

**sublocale** *residuation Resid*  
**using** *null-char Con-sym Arr-Resid Cube*  
**by** *unfold-locales metis+*

**lemma** *is-residuation:*  
**shows** *residuation Resid*  
**..**

**lemma** *arr-char:*  
**shows**  $\text{arr } \mathcal{T} \longleftrightarrow N.\text{is-Cong-class } \mathcal{T}$   
**by** (*metis*  $N.\text{is-Cong-class-def } \text{arrI } \text{not-arr-null } \text{null-char } N.\text{Cong-class-memb-is-arr}$   
 $\text{Con-char } R.\text{arrE } \text{arrE } \text{arr-resid } \text{conI}$ )

**lemma** *ide-char:*  
**shows**  $\text{ide } \mathcal{U} \longleftrightarrow \text{arr } \mathcal{U} \wedge \mathcal{U} \cap \mathfrak{N} \neq \{\}$   
**proof**  
**show**  $\text{ide } \mathcal{U} \Longrightarrow \text{arr } \mathcal{U} \wedge \mathcal{U} \cap \mathfrak{N} \neq \{\}$   
**apply** (*elim ideE*)  
**by** (*metis*  $\text{Con-char } N.\text{Cong}_0\text{-reflexive } \text{Resid-by-members } \text{disjoint-iff } \text{null-char}$   
 $N.\text{arr-in-Cong-class } R.\text{arrE } R.\text{arr-resid } \text{arr-resid } \text{conE}$ )  
**show**  $\text{arr } \mathcal{U} \wedge \mathcal{U} \cap \mathfrak{N} \neq \{\} \Longrightarrow \text{ide } \mathcal{U}$   
**proof** –  
**assume**  $\mathcal{U}: \text{arr } \mathcal{U} \wedge \mathcal{U} \cap \mathfrak{N} \neq \{\}$   
**obtain**  $u$  **where**  $u: R.\text{arr } u \wedge u \in \mathcal{U} \cap \mathfrak{N}$   
**using**  $\mathcal{U} \text{arr-char}$   
**by** (*metis*  $\text{IntI } N.\text{Cong-class-memb-is-arr } \text{disjoint-iff}$ )  
**show** *?thesis*  
**by** (*metis*  $\text{IntD1 } \text{IntD2 } N.\text{Cong-class-eqI } N.\text{Cong-closure-props}(4) N.\text{arr-in-Cong-class}$   
 $N.\text{is-Cong-classI } \text{Resid-by-members } \mathcal{U} \text{arrE } \text{arr-char } \text{disjoint-iff } \text{ideI}$   
 $N.\text{Cong-class-eqI}' R.\text{arrE } u$ )  
**qed**  
**qed**

**lemma** *ide-char':*

**shows**  $\text{ide } \mathcal{A} \iff \text{arr } \mathcal{A} \wedge \mathcal{A} \subseteq \mathfrak{N}$   
**by** (*metis Int-absorb2 Int-emptyI N.Cong-class-memb-Cong-rep N.Cong-closure-props(1) ide-char not-arr-null null-char N.normal-is-Cong-closed arr-char subsetI*)

**lemma** *con-char<sub>QCN</sub>*:

**shows**  $\text{con } \mathcal{T} \mathcal{U} \iff$   
 $N.\text{is-Cong-class } \mathcal{T} \wedge N.\text{is-Cong-class } \mathcal{U} \wedge (\exists t u. t \in \mathcal{T} \wedge u \in \mathcal{U} \wedge t \frown u)$   
**by** (*metis Con-char conE conI null-char*)

**lemma** *con-imp-coinitial-members-are-con*:

**assumes**  $\text{con } \mathcal{T} \mathcal{U}$  **and**  $t \in \mathcal{T}$  **and**  $u \in \mathcal{U}$  **and**  $R.\text{sources } t = R.\text{sources } u$   
**shows**  $t \frown u$   
**by** (*meson assms N.Cong-subst(1) N.is-Cong-classE con-char<sub>QCN</sub>*)

**sublocale** *rts Resid*

**proof**

**show**  $1: \bigwedge \mathcal{A} \mathcal{T}. \llbracket \text{ide } \mathcal{A}; \text{con } \mathcal{T} \mathcal{A} \rrbracket \implies \mathcal{T} \{\!\!\}\ \mathcal{A} = \mathcal{T}$

**proof** –

**fix**  $\mathcal{A} \mathcal{T}$

**assume**  $\mathcal{A}$ : *ide*  $\mathcal{A}$  **and** *con*:  $\text{con } \mathcal{T} \mathcal{A}$

**obtain**  $t a$  **where**  $ta$ :  $t \in \mathcal{T} \wedge a \in \mathcal{A} \wedge R.\text{con } t a \wedge \mathcal{T} \{\!\!\}\ \mathcal{A} = \{t \setminus a\}$

**using** *con con-char<sub>QCN</sub> Resid-by-members* **by** *auto*

**have**  $a \in \mathfrak{N}$

**using**  $\mathcal{A}$   $ta$  *ide-char'* **by** *auto*

**hence**  $t \setminus a \approx t$

**by** (*meson N.Cong-closure-props(4) N.Cong-symmetric R.coinitialE R.con-imp-coinitial ta*)

**thus**  $\mathcal{T} \{\!\!\}\ \mathcal{A} = \mathcal{T}$

**using**  $ta$

**by** (*metis N.Cong-class-eqI N.Cong-class-memb-Cong-rep N.Cong-class-rep con con-char<sub>QCN</sub>*)

**qed**

**show**  $\bigwedge \mathcal{T}. \text{arr } \mathcal{T} \implies \text{ide } (\text{trg } \mathcal{T})$

**by** (*metis N.Cong<sub>0</sub>-reflexive Resid-by-members disjoint-iff ide-char N.Cong-class-memb-is-arr N.arr-in-Cong-class N.is-Cong-class-def arr-char R.arrE R.arr-resid resid-arr-self*)

**show**  $\bigwedge \mathcal{A} \mathcal{T}. \llbracket \text{ide } \mathcal{A}; \text{con } \mathcal{A} \mathcal{T} \rrbracket \implies \text{ide } (\mathcal{A} \{\!\!\}\ \mathcal{T})$

**by** (*metis 1 arrE arr-resid con-sym ideE ideI cube*)

**show**  $\bigwedge \mathcal{T} \mathcal{U}. \text{con } \mathcal{T} \mathcal{U} \implies \exists \mathcal{A}. \text{ide } \mathcal{A} \wedge \text{con } \mathcal{A} \mathcal{T} \wedge \text{con } \mathcal{A} \mathcal{U}$

**proof** –

**fix**  $\mathcal{T} \mathcal{U}$

**assume**  $\mathcal{T}\mathcal{U}$ :  $\text{con } \mathcal{T} \mathcal{U}$

**obtain**  $t u$  **where**  $tu$ :  $\mathcal{T} = \{t\} \wedge \mathcal{U} = \{u\} \wedge t \frown u$

**using**  $\mathcal{T}\mathcal{U}$  *con-char<sub>QCN</sub> arr-char*

**by** (*metis N.Cong-class-memb-Cong-rep N.Cong-class-eqI N.Cong-class-rep*)

**obtain**  $a$  **where**  $a$ :  $a \in R.\text{sources } t$

**using**  $\mathcal{T}\mathcal{U}$   $tu$  *R.con-implies-arr(1) R.arr-iff-has-source* **by** *blast*

**have**  $\text{ide } \{a\} \wedge \text{con } \{a\} \mathcal{T} \wedge \text{con } \{a\} \mathcal{U}$

**proof** (*intro conjI*)

**have** 2:  $a \in \mathfrak{N}$   
**using**  $\mathcal{T}\mathcal{U}$   $tu$   $a$  *arr-char*  $N$ .*ide-closed*  $R$ .*sources-def* **by** *force*  
**show** 3: *ide*  $\{\!\{a\}\!\}$   
**using**  $\mathcal{T}\mathcal{U}$   $tu$  2  $a$  *ide-char* *arr-char* *con-char* $_{QCN}$   
**by** (*metis*  $IntI$   $N$ .*arr-in-Cong-class*  $N$ .*is-Cong-classI* *empty-iff*  $N$ .*elements-are-arr*)  
**show** *con*  $\{\!\{a\}\!\}$   $\mathcal{T}$   
**using**  $\mathcal{T}\mathcal{U}$   $tu$  2 3  $a$  *ide-char* *arr-char* *con-char* $_{QCN}$   
**by** (*metis*  $N$ .*arr-in-Cong-class*  $R$ .*composite-of-source-arr*  $R$ .*composite-of-def*  $R$ .*prfx-implies-con*  $R$ .*con-implies-arr*(1))  
**show** *con*  $\{\!\{a\}\!\}$   $\mathcal{U}$   
**using**  $\mathcal{T}\mathcal{U}$   $tu$   $a$  *ide-char* *arr-char* *con-char* $_{QCN}$   
**by** (*metis*  $N$ .*arr-in-Cong-class*  $R$ .*composite-of-source-arr*  $R$ .*con-prfx-composite-of*  $N$ .*is-Cong-classI*  $R$ .*con-implies-arr*(1)  $R$ .*con-implies-arr*(2))  
**qed**  
**thus**  $\exists A$ . *ide*  $\mathcal{A} \wedge$  *con*  $\mathcal{A} \mathcal{T} \wedge$  *con*  $\mathcal{A} \mathcal{U}$  **by** *auto*  
**qed**  
**show**  $\bigwedge \mathcal{T} \mathcal{U} \mathcal{V}$ .  $\llbracket \textit{ide} (\mathcal{T} \{\!\{\!\} \mathcal{U}\}\!\} ); \textit{con} \mathcal{U} \mathcal{V} \rrbracket \implies \textit{con} (\mathcal{T} \{\!\{\!\} \mathcal{U}\}\!\} ) (\mathcal{V} \{\!\{\!\} \mathcal{U}\}\!\} )$   
**proof** –  
**fix**  $\mathcal{T} \mathcal{U} \mathcal{V}$   
**assume**  $\mathcal{T}\mathcal{U}$ : *ide*  $(\mathcal{T} \{\!\{\!\} \mathcal{U}\}\!\} )$   
**assume**  $\mathcal{U}\mathcal{V}$ : *con*  $\mathcal{U} \mathcal{V}$   
**obtain**  $t u$  **where**  $tu$ :  $t \in \mathcal{T} \wedge u \in \mathcal{U} \wedge t \frown u \wedge \mathcal{T} \{\!\{\!\} \mathcal{U}\}\!\} = \{\!\{t \setminus u\}\!\}$   
**using**  $\mathcal{T}\mathcal{U}$   
**by** (*meson*  $Resid$ -*by-members* *ide-implies-arr* *con-char* $_{QCN}$  *arr-resid-iff-con*)  
**obtain**  $v u'$  **where**  $vu'$ :  $v \in \mathcal{V} \wedge u' \in \mathcal{U} \wedge v \frown u' \wedge \mathcal{V} \{\!\{\!\} \mathcal{U}\}\!\} = \{\!\{v \setminus u'\}\!\}$   
**by** (*meson*  $R$ .*con-sym*  $Resid$ -*by-members*  $\mathcal{U}\mathcal{V}$  *con-char* $_{QCN}$ )  
**have** 1:  $u \approx u'$   
**using**  $\mathcal{U}\mathcal{V}$   $tu$   $vu'$   
**by** (*meson*  $N$ .*Cong-class-membs-are-Cong* *con-char* $_{QCN}$ )  
**obtain**  $w w'$  **where**  $ww'$ :  $w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge u \setminus w \approx_0 u' \setminus w'$   
**using** 1 **by** *auto*  
**have** 2:  $((t \setminus u) \setminus (w \setminus u)) \setminus ((u' \setminus w') \setminus (u \setminus w)) \frown ((v \setminus u') \setminus (w' \setminus u')) \setminus ((u \setminus w) \setminus (u' \setminus w'))$   
**proof** –  
**have**  $((t \setminus u) \setminus (w \setminus u)) \setminus ((u' \setminus w') \setminus (u \setminus w)) \in \mathfrak{N}$   
**proof** –  
**have**  $t \setminus u \in \mathfrak{N}$   
**using**  $tu$   $N$ .*arr-in-Cong-class*  $R$ .*arr-resid-iff-con*  $\mathcal{T}\mathcal{U}$  *ide-char'* **by** *blast*  
**hence**  $(t \setminus u) \setminus (w \setminus u) \in \mathfrak{N}$   
**by** (*metis*  $N$ .*Cong-closure-props*(4)  $N$ .*forward-stable*  $R$ .*null-is-zero*(2)  $R$ .*con-imp-coinitial*  $R$ .*sources-resid*  $N$ .*Cong-imp-arr*(2)  $R$ .*arr-resid-iff-con*  $tu$   $ww'$   $R$ .*conI*)  
**thus** *?thesis*  
**by** (*metis*  $N$ .*Cong-closure-props*(4)  $N$ .*normal-is-Cong-closed*  $R$ .*sources-resid*  $R$ .*targets-resid-sym*  $N$ .*elements-are-arr*  $R$ .*arr-resid-iff-con*  $ww'$   $R$ .*conI*)  
**qed**  
**moreover** **have**  $R$ .*sources*  $((((t \setminus u) \setminus (w \setminus u)) \setminus ((u' \setminus w') \setminus (u \setminus w))) = R$ .*sources*  $((((v \setminus u') \setminus (w' \setminus u')) \setminus ((u \setminus w) \setminus (u' \setminus w'))))$   
**proof** –

**have**  $R.sources ((t \setminus u) \setminus (w \setminus u)) \setminus ((u' \setminus w') \setminus (u \setminus w)) =$   
 $R.targets ((u' \setminus w') \setminus (u \setminus w))$   
**using**  $R.arr-resid-iff-con$   $N.elements-are-arr$   $R.sources-resid$  **calculation** **by**  $blast$   
**also have**  $\dots = R.targets ((u \setminus w) \setminus (u' \setminus w'))$   
**by**  $(metis\ R.targets-resid-sym\ R.conI)$   
**also have**  $\dots = R.sources (((v \setminus u') \setminus (w' \setminus u')) \setminus ((u \setminus w) \setminus (u' \setminus w')))$   
**using**  $R.arr-resid-iff-con$   $N.elements-are-arr$   $R.sources-resid$   
**by**  $(metis\ N.Cong-closure-props(4)\ N.Cong-imp-arr(2)\ R.con-implies-arr(1)\ R.con-imp-coinitial\ N.forward-stable\ R.targets-resid-sym\ vu'\ ww')$   
**finally show**  $?thesis$  **by**  $simp$   
**qed**  
**ultimately show**  $?thesis$   
**by**  $(metis\ (no-types,\ lifting)\ N.Cong_0-imp-con\ N.Cong-closure-props(4)\ N.Cong-imp-arr(2)\ R.arr-resid-iff-con\ R.con-imp-coinitial\ N.forward-stable\ R.null-is-zero(2)\ R.conI)$   
**qed**  
**moreover have**  $t \setminus u \approx ((t \setminus u) \setminus (w \setminus u)) \setminus ((u' \setminus w') \setminus (u \setminus w))$   
**by**  $(metis\ (no-types,\ opaque-lifting)\ N.Cong-closure-props(4)\ N.Cong-transitive\ N.forward-stable\ R.arr-resid-iff-con\ R.con-imp-coinitial\ R.rts-axioms\ calculation\ rts.coinitial-iff\ ww')$   
**moreover have**  $v \setminus u' \approx ((v \setminus u') \setminus (w' \setminus u')) \setminus ((u \setminus w) \setminus (u' \setminus w'))$   
**proof** –  
**have**  $w' \setminus u' \in \mathfrak{N}$   
**by**  $(meson\ R.con-implies-arr(2)\ R.con-imp-coinitial\ N.forward-stable\ ww'\ N.Cong_0-imp-con\ R.arr-resid-iff-con)$   
**moreover have**  $(u \setminus w) \setminus (u' \setminus w') \in \mathfrak{N}$   
**using**  $ww'$  **by**  $blast$   
**ultimately show**  $?thesis$   
**by**  $(meson\ 2\ N.Cong-closure-props(2)\ N.Cong-closure-props(4)\ R.arr-resid-iff-con\ R.coinitial-iff\ R.con-imp-coinitial)$   
**qed**  
**ultimately show**  $con\ (\mathcal{T}\ \{\!\!\}\ \mathcal{U})\ (\mathcal{V}\ \{\!\!\}\ \mathcal{U})$   
**using**  $con-char_{QC_N}$   $N.Cong-class-def$   $N.is-Cong-classI$   $tu\ vu'$   $R.arr-resid-iff-con$   
**by**  $auto$   
**qed**  
**qed**

**lemma**  $is-rts$ :  
**shows**  $rts\ Resid$   
**..**

**sublocale**  $extensional-rts\ Resid$   
**proof**  
**fix**  $\mathcal{T}\ \mathcal{U}$   
**assume**  $\mathcal{T}\mathcal{U}$ :  $cong\ \mathcal{T}\ \mathcal{U}$   
**show**  $\mathcal{T} = \mathcal{U}$   
**proof** –  
**obtain**  $t\ u$  **where**  $tu$ :  $\mathcal{T} = \{\!\!\}t\ \wedge\ \mathcal{U} = \{\!\!\}u\ \wedge\ t \frown u$   
**by**  $(metis\ Con-char\ N.Cong-class-eqI\ N.Cong-class-memb-Cong-rep\ N.Cong-class-rep)$

```

     $\mathcal{T}\mathcal{U}$  ide-char not-arr-null null-char)
  have  $t \approx_0 u$ 
  proof
    show  $t \setminus u \in \mathfrak{N}$ 
      using  $tu \mathcal{T}\mathcal{U}$  Resid-by-members [of  $\mathcal{T} \mathcal{U} t u$ ]
      by (metis (full-types) N.arr-in-Cong-class R.con-implies-arr(1-2)
        N.is-Cong-classI ide-char' R.arr-resid-iff-con subset-iff)
    show  $u \setminus t \in \mathfrak{N}$ 
      using  $tu \mathcal{T}\mathcal{U}$  Resid-by-members [of  $\mathcal{U} \mathcal{T} u t$ ] R.con-sym
      by (metis (full-types) N.arr-in-Cong-class R.con-implies-arr(1-2)
        N.is-Cong-classI ide-char' R.arr-resid-iff-con subset-iff)
  qed
  hence  $t \approx u$ 
    using N.Cong0-implies-Cong by simp
  thus  $\mathcal{T} = \mathcal{U}$ 
    by (simp add: N.Cong-class-eqI tu)
  qed
  qed

```

**theorem** *is-extensional-rts:*  
**shows** *extensional-rts Resid*

..

**lemma** *sources-char<sub>QCN</sub>:*

**shows**  $\text{sources } \mathcal{T} = \{\mathcal{A}. \text{arr } \mathcal{T} \wedge \mathcal{A} = \{a. \exists t a'. t \in \mathcal{T} \wedge a' \in R.\text{sources } t \wedge a' \approx a\}\}$

**proof** –

let  $?A = \{a. \exists t a'. t \in \mathcal{T} \wedge a' \in R.\text{sources } t \wedge a' \approx a\}$

have 1:  $\text{arr } \mathcal{T} \implies \text{ide } ?A$

**proof** (*unfold ide-char', intro conjI*)

assume  $\mathcal{T}: \text{arr } \mathcal{T}$

show  $?A \subseteq \mathfrak{N}$

using *N.ide-closed N.normal-is-Cong-closed* by blast

show  $\text{arr } ?A$

**proof** –

have *N.is-Cong-class ?A*

**proof**

show  $?A \neq \{\}$

by (*metis (mono-tags, lifting) Collect-empty-eq N.Cong-class-def N.Cong-imp-arr(1)*  
*N.is-Cong-class-def N.sources-are-Cong R.arr-iff-has-source R.sources-def*  
 $\mathcal{T}$  *arr-char mem-Collect-eq*)

show  $\bigwedge a a'. \llbracket a \in ?A; a' \approx a \rrbracket \implies a' \in ?A$

using *N.Cong-transitive* by blast

show  $\bigwedge a a'. \llbracket a \in ?A; a' \in ?A \rrbracket \implies a \approx a'$

**proof** –

fix  $a a'$

assume  $a: a \in ?A$  and  $a': a' \in ?A$

obtain  $t b$  where  $b: t \in \mathcal{T} \wedge b \in R.\text{sources } t \wedge b \approx a$

using  $a$  by blast

obtain  $t' b'$  where  $b': t' \in \mathcal{T} \wedge b' \in R.\text{sources } t' \wedge b' \approx a'$

```

    using a' by blast
  have b ≈ b'
    using  $\mathcal{T}$  arr-char b b'
    by (meson IntD1 N.Cong-class-membs-are-Cong N.in-sources-respects-Cong)
  thus a ≈ a'
    by (meson N.Cong-symmetric N.Cong-transitive b b')
qed
qed
thus ?thesis
  using arr-char by auto
qed
qed
moreover have arr  $\mathcal{T} \implies$  con  $\mathcal{T}$  ?A
proof -
  assume  $\mathcal{T}$ : arr  $\mathcal{T}$ 
  obtain t a where a: t ∈  $\mathcal{T} \wedge a \in R.sources$  t
    using  $\mathcal{T}$  arr-char
    by (metis N.Cong-class-is-nonempty R.arr-iff-has-source empty-subsetI
        N.Cong-class-memb-is-arr subsetI subset-antisym)
  have t ∈  $\mathcal{T} \wedge a \in \{a. \exists t a'. t \in \mathcal{T} \wedge a' \in R.sources$  t  $\wedge a' \approx a\} \wedge t \frown a$ 
    using a N.Cong-reflexive R.sources-def R.con-implies-arr(2) by fast
  thus ?thesis
    using  $\mathcal{T}$  1 arr-char con-charQCN [of  $\mathcal{T}$  ?A] by auto
qed
ultimately have arr  $\mathcal{T} \implies$  ?A ∈ sources  $\mathcal{T}$ 
  using sources-def by blast
thus ?thesis
  using 1 ide-char sources-charWE by auto
qed

lemma targets-charQCN:
shows targets  $\mathcal{T} = \{\mathcal{B}. arr \mathcal{T} \wedge \mathcal{B} = \mathcal{T} \setminus \setminus \mathcal{T}\}$ 
proof -
  have targets  $\mathcal{T} = \{\mathcal{B}. ide \mathcal{B} \wedge con (\mathcal{T} \setminus \setminus \mathcal{T}) \mathcal{B}\}$ 
    by (simp add: targets-def trg-def)
  also have ... =  $\{\mathcal{B}. arr \mathcal{T} \wedge ide \mathcal{B} \wedge (\exists t u. t \in \mathcal{T} \setminus \setminus \mathcal{T} \wedge u \in \mathcal{B} \wedge t \frown u)\}$ 
    using arr-resid-iff-con con-charQCN arr-char arr-def by auto
  also have ... =  $\{\mathcal{B}. arr \mathcal{T} \wedge ide \mathcal{B} \wedge$ 
     $(\exists t t' b u. t \in \mathcal{T} \wedge t' \in \mathcal{T} \wedge t \frown t' \wedge b \in \{t \setminus t'\} \wedge u \in \mathcal{B} \wedge b \frown u)\}$ 
    apply auto
    apply (metis (full-types) Resid-by-members cong-char not-ide-null null-char Con-char)
    by (metis Resid-by-members arr-char)
  also have ... =  $\{\mathcal{B}. arr \mathcal{T} \wedge ide \mathcal{B} \wedge$ 
     $(\exists t t' b. t \in \mathcal{T} \wedge t' \in \mathcal{T} \wedge t \frown t' \wedge b \in \{t \setminus t'\} \wedge b \in \mathcal{B})\}$ 
  proof -
    have  $\bigwedge \mathcal{B} t t' b. \llbracket arr \mathcal{T}; ide \mathcal{B}; t \in \mathcal{T}; t' \in \mathcal{T}; t \frown t'; b \in \{t \setminus t'\} \rrbracket$ 
       $\implies (\exists u. u \in \mathcal{B} \wedge b \frown u) \iff b \in \mathcal{B}$ 
    proof -
      fix  $\mathcal{B} t t' b$ 

```

**assume**  $\mathcal{T}$ : *arr*  $\mathcal{T}$  **and**  $\mathcal{B}$ : *ide*  $\mathcal{B}$  **and**  $t$ :  $t \in \mathcal{T}$  **and**  $t'$ :  $t' \in \mathcal{T}$   
**and**  $tt'$ :  $t \frown t'$  **and**  $b$ :  $b \in \{\!|t \setminus t'|\!\}$   
**have**  $0$ :  $b \in \mathfrak{N}$   
**by** (*metis Resid-by-members*  $\mathcal{T}$   $b$  *ide-char'* *ide-trg* *arr-char* *subsetD*  $t$   $t'$  *trg-def*  $tt'$ )  
**show**  $(\exists u. u \in \mathcal{B} \wedge b \frown u) \longleftrightarrow b \in \mathcal{B}$   
**using**  $0$   
**by** (*meson*  $N$ .*Cong-closure-props*( $\mathcal{B}$ )  $N$ .*forward-stable*  $N$ .*elements-are-arr*  
 $\mathcal{B}$  *arr-char*  $R$ .*con-imp-coinitial*  $N$ .*is-Cong-classE* *ide-char'*  $R$ .*arrE*  
 $R$ .*con-sym* *subsetD*)  
**qed**  
**thus** *?thesis*  
**using** *ide-char* *arr-char*  
**by** (*metis* (*no-types*, *lifting*))  
**qed**  
**also have**  $\dots = \{\mathcal{B}. \text{arr } \mathcal{T} \wedge \text{ide } \mathcal{B} \wedge (\exists t t'. t \in \mathcal{T} \wedge t' \in \mathcal{T} \wedge t \frown t' \wedge \{\!|t \setminus t'|\!\} \subseteq \mathcal{B})\}$   
**proof** –  
**have**  $\bigwedge \mathcal{B} t t' b. \llbracket \text{arr } \mathcal{T}; \text{ide } \mathcal{B}; t \in \mathcal{T}; t' \in \mathcal{T}; t \frown t' \rrbracket$   
 $\implies (\exists b. b \in \{\!|t \setminus t'|\!\} \wedge b \in \mathcal{B}) \longleftrightarrow \{\!|t \setminus t'|\!\} \subseteq \mathcal{B}$   
**using** *ide-char* *arr-char*  
**apply** (*intro iffI*)  
**apply** (*metis*  $IntI$   $N$ .*Cong-class-eqI'*  $R$ .*arr-resid-iff-con*  $N$ .*is-Cong-classI* *empty-iff*  
*set-eq-subset*)  
**by** (*meson*  $N$ .*arr-in-Cong-class*  $R$ .*arr-resid-iff-con* *subsetD*)  
**thus** *?thesis*  
**using** *ide-char* *arr-char*  
**by** (*metis* (*no-types*, *lifting*))  
**qed**  
**also have**  $\dots = \{\mathcal{B}. \text{arr } \mathcal{T} \wedge \text{ide } \mathcal{B} \wedge \mathcal{T} \{\!|\setminus|\!\} \mathcal{T} \subseteq \mathcal{B}\}$   
**using** *arr-char* *ide-char* *Resid-by-members* [*of*  $\mathcal{T}$   $\mathcal{T}$ ]  
**by** (*metis* (*no-types*, *opaque-lifting*) *arrE* *con-char* $_{QCN}$ )  
**also have**  $\dots = \{\mathcal{B}. \text{arr } \mathcal{T} \wedge \mathcal{B} = \mathcal{T} \{\!|\setminus|\!\} \mathcal{T}\}$   
**by** (*metis* (*no-types*, *lifting*) *arr-has-un-target* *calculation* *con-ide-are-eq*  
*cong-reflexive* *mem-Collect-eq* *targets-def* *trg-def*)  
**finally show** *?thesis* **by** *blast*  
**qed**  
  
**lemma** *src-char* $_{QCN}$ :  
**shows**  $\text{src } \mathcal{T} = \{a. \text{arr } \mathcal{T} \wedge (\exists t a'. t \in \mathcal{T} \wedge a' \in R.\text{sources } t \wedge a' \approx a)\}$   
**using** *sources-char* $_{QCN}$  [*of*  $\mathcal{T}$ ]  
**by** (*simp* *add*: *null-char* *src-def*)  
  
**lemma** *trg-char* $_{QCN}$ :  
**shows**  $\text{trg } \mathcal{T} = \mathcal{T} \{\!|\setminus|\!\} \mathcal{T}$   
**unfolding** *trg-def* **by** *blast*

## Quotient Map

**abbreviation** *quot*  
**where** *quot*  $t \equiv \{\!|t|\!\}$

**sublocale** *quot: simulation-to-extensional-rts resid Resid quot*  
**proof**  
 show  $\bigwedge t. \neg R.arr\ t \implies \{t\} = null$   
 using *N.Cong-class-def N.Cong-imp-arr(1) null-char* **by** *force*  
 show  $\bigwedge t\ u. t \frown u \implies con\ \{t\}\ \{u\}$   
 by (*meson N.arr-in-Cong-class N.is-Cong-classI R.con-implies-arr(1-2) con-char<sub>QCN</sub>*)  
 show  $\bigwedge t\ u. t \frown u \implies \{t \setminus u\} = \{t\} \setminus \{u\}$   
 by (*metis N.arr-in-Cong-class N.is-Cong-classI R.con-implies-arr(1-2) Resid-by-members*)  
**qed**

**lemma** *quotient-is-simulation:*  
**shows** *simulation resid Resid quot*  
 ..

**lemma** *ide-quot-normal:*  
**assumes**  $t \in \mathfrak{N}$   
**shows** *ide (quot t)*  
 using *assms*  
 by (*metis IntI N.arr-in-Cong-class N.elements-are-arr empty-iff quot.preserves-reflects-arr ide-char*)

If a simulation  $F$  from  $R$  to an extensional RTS  $B$  maps every element of  $\mathfrak{N}$  to an identity, then it has a unique extension along the quotient map.

**lemma** *is-couniversal:*  
**assumes** *extensional-rts B*  
**and** *simulation resid B F*  
**and**  $\bigwedge t. t \in \mathfrak{N} \implies residuation.ide\ B\ (F\ t)$   
**shows**  $\exists! F'. simulation\ Resid\ B\ F' \wedge F' \circ quot = F$   
**proof** –  
 interpret  $B: extensional-rts\ B$   
 using *assms(1) simulation.axioms(2)* **by** *blast*  
 interpret  $F: simulation\ resid\ B\ F$   
 using *assms* **by** *blast*  
 have 1:  $\bigwedge t\ u. t \approx u \implies F\ t = F\ u$   
**proof** –  
 fix  $t\ u$   
 assume *Cong: t ≈ u*  
 obtain  $v\ w$  **where**  $vw: v \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge t \setminus v \approx_0 u \setminus w$   
 using *Cong* **by** *blast*  
 have  $B.cong\ (F\ t)\ (F\ u)$   
 by (*metis assms(3) vw B.cong-char F.preserves-reflects-arr F.preserves-resid N.elements-are-arr R.arr-resid-iff-con B.arr-resid-iff-con B.resid-arr-ide*)  
 thus  $F\ t = F\ u$   
 using *B.extensionality* **by** *blast*  
**qed**  
 let  $?F' = \lambda \mathcal{T}. if\ arr\ \mathcal{T}\ then\ F\ (N.Cong-class-rep\ \mathcal{T})\ else\ B.null$   
 interpret  $F': simulation\ Resid\ B\ ?F'$   
**proof**

```

show  $\bigwedge \mathcal{T}. \neg \text{arr } \mathcal{T} \implies ?F' \mathcal{T} = B.\text{null}$ 
  by argo
fix  $\mathcal{T} \mathcal{U}$ 
assume con: con  $\mathcal{T} \mathcal{U}$ 
show  $B.\text{con} (?F' \mathcal{T}) (?F' \mathcal{U})$ 
  using con
  by (metis (full-types) 1 F.preserves-con N.Cong-class-memb-Cong-rep arr-char
      con-charQCN)
show  $?F' (\mathcal{T} \setminus \setminus \mathcal{U}) = B (?F' \mathcal{T}) (?F' \mathcal{U})$ 
proof -
  have 2: N.is-Cong-class  $\mathcal{T} \wedge \textit{N.is-Cong-class } \mathcal{U}$ 
    using con con-charQCN by auto
  obtain  $t \ u$  where  $tu$ :  $t \in \mathcal{T} \wedge u \in \mathcal{U} \wedge t \frown u$ 
    using con con-charQCN by force
  have  $?F' (\mathcal{T} \setminus \setminus \mathcal{U}) = ?F' \{t \setminus u\}$ 
    using  $tu$  2 Resid-by-members by force
  also have  $\dots = F (t \setminus u)$ 
    by (metis  $tu$  N.Cong-class-memb-Cong-rep N.arr-in-Cong-class N.is-Cong-classI
        R.arr-resid  $\langle \bigwedge y \ x. x \approx y \implies F \ x = F \ y \rangle$  quot.preserves-reflects-arr)
  also have  $\dots = B (F \ t) (F \ u)$ 
    by (simp add:  $tu$ )
  also have  $\dots = B (?F' \mathcal{T}) (?F' \mathcal{U})$ 
    by (metis (full-types)  $tu$  1 2 N.Cong-class-memb-Cong-rep con
        con-implies-arr(1-2))
  finally show ?thesis by blast
qed
qed
have simulation Resid  $B \ ?F' \wedge ?F' \circ \text{quot} = F$ 
proof -
  have  $?F' \circ \text{quot} = F$ 
  proof
  fix  $t$ 
  have  $?F' (\text{quot } t) = F \ t$ 
    by (metis 1 F.extensionality N.Cong-class-memb-Cong-rep N.arr-in-Cong-class
        arr-char quot.preserves-reflects-arr)
  thus  $(?F' \circ \text{quot}) \ t = F \ t$ 
    by auto
  qed
  thus ?thesis
    using F'.simulation-axioms by blast
  qed
moreover have  $\bigwedge F''. \llbracket \text{simulation } \text{Resid } B \ F''; F'' \circ \text{quot} = F \rrbracket \implies F'' = ?F'$ 
  using simulation.extensionality arr-char by force
ultimately show ?thesis by blast
qed

```

**definition** *ext-to-quotient*

where *ext-to-quotient*  $B \ F \equiv \text{THE } F'. \text{simulation } \text{Resid } B \ F' \wedge F' \circ \text{quot} = F$

**lemma** *ext-to-quotient-props*:  
**assumes** *extensional-rts B*  
**and** *simulation resid B F*  
**and**  $\bigwedge t. t \in \mathfrak{N} \implies \text{residuation.ide } B (F t)$   
**shows** *simulation Resid B (ext-to-quotient B F) and ext-to-quotient B F o quot = F*  
**proof** –  
**have** *simulation Resid B (ext-to-quotient B F)  $\wedge$  ext-to-quotient B F o quot = F*  
**unfolding** *ext-to-quotient-def*  
**using** *assms is-couniversal [of B F]*  
*theI' [of  $\lambda F'. \text{simulation } (\{\!\!\!|\} \backslash \{\!\!\!|\}) B F' \wedge F' \circ \text{quot} = F$ ]*  
**by** *fastforce*  
**thus** *simulation Resid B (ext-to-quotient B F) and ext-to-quotient B F o quot = F*  
**by** *auto*  
**qed**

end

### 2.3.7 Identities form a Coherent Normal Sub-RTS

We now show that the collection of identities of an RTS form a coherent normal sub-RTS, and that the associated congruence  $\approx$  coincides with  $\sim$ . Thus, every RTS can be factored by the relation  $\sim$  to obtain an extensional RTS. Although we could have shown that fact much earlier, we have delayed proving it so that we could simply obtain it as a special case of our general quotient result without redundant work.

**context** *rts*  
**begin**

**interpretation** *normal-sub-rts resid <Collect ide>*  
**proof**  
**show**  $\bigwedge t. t \in \text{Collect ide} \implies \text{arr } t$   
**by** *blast*  
**show**  $1: \bigwedge a. \text{ide } a \implies a \in \text{Collect ide}$   
**by** *blast*  
**show**  $\bigwedge u t. \llbracket u \in \text{Collect ide}; \text{coinitial } t u \rrbracket \implies u \setminus t \in \text{Collect ide}$   
**by** (*metis 1 CollectD arr-def coinitial-iff*  
*con-sym in-sourcesE in-sourcesI resid-ide-arr*)  
**show**  $\bigwedge u t. \llbracket u \in \text{Collect ide}; t \setminus u \in \text{Collect ide} \rrbracket \implies t \in \text{Collect ide}$   
**using** *ide-backward-stable* **by** *blast*  
**show**  $\bigwedge u t. \llbracket u \in \text{Collect ide}; \text{seq } u t \rrbracket \implies \exists v. \text{composite-of } u t v$   
**by** (*metis composite-of-source-arr ide-def in-sourcesI mem-Collect-eq seq-def*  
*resid-source-in-targets*)  
**show**  $\bigwedge u t. \llbracket u \in \text{Collect ide}; \text{seq } t u \rrbracket \implies \exists v. \text{composite-of } t u v$   
**by** (*metis arrE composite-of-arr-target in-sourcesI seqE mem-Collect-eq*)  
**qed**

**lemma** *identities-form-normal-sub-rts*:  
**shows** *normal-sub-rts resid (Collect ide)*

..

```

interpretation coherent-normal-sub-rts resid ‹Collect ide›
  apply unfold-locales
  by (metis CollectD Cong0-reflexive Cong-closure-props(4) Cong-imp-arr(2)
      arr-resid-iff-con resid-arr-ide)

```

```

lemma identities-form-coherent-normal-sub-rts:
shows coherent-normal-sub-rts resid (Collect ide)
  ..

```

```

lemma Cong-iff-cong:
shows Cong t u  $\longleftrightarrow$  t ~ u
  by (metis CollectD Cong-def ide-closed resid-arr-ide
      Cong-closure-props(3) Cong-imp-arr(2) arr-resid-iff-con)

```

**end**

## 2.4 Paths

A *path* in an RTS is a nonempty list of arrows such that the set of targets of each arrow suitably matches the set of sources of its successor. The residuation on the given RTS extends inductively to a residuation on paths, so that paths also form an RTS. The append operation on lists yields a composite for each pair of compatible paths.

```

locale paths-in-rts =
  R: rts
begin

  type-synonym 'b arr = 'b list

  fun Srcs
  where Srcs [] = {}
        | Srcs [t] = R.sources t
        | Srcs (t # T) = R.sources t

  fun Trgs
  where Trgs [] = {}
        | Trgs [t] = R.targets t
        | Trgs (t # T) = Trgs T

  fun Arr
  where Arr [] = False
        | Arr [t] = R.arr t
        | Arr (t # T) = (R.arr t  $\wedge$  Arr T  $\wedge$  R.targets t  $\subseteq$  Srcs T)

  fun Ide
  where Ide [] = False
        | Ide [t] = R.ide t
        | Ide (t # T) = (R.ide t  $\wedge$  Ide T  $\wedge$  R.targets t  $\subseteq$  Srcs T)

```

**lemma** *Arr-induct*:  
**assumes**  $\bigwedge t. \text{Arr } [t] \implies P [t]$   
**and**  $\bigwedge t U. \llbracket \text{Arr } (t \# U); U \neq []; P U \rrbracket \implies P (t \# U)$   
**shows**  $\text{Arr } T \implies P T$   
**proof** (*induct T*)  
  **show**  $\text{Arr } [] \implies P []$   
  **using** *Arr.simps(1)* **by** *blast*  
  **show**  $\bigwedge t U. \llbracket \text{Arr } U \implies P U; \text{Arr } (t \# U) \rrbracket \implies P (t \# U)$   
  **by** (*metis assms Arr.simps(2-3) list.exhaust*)  
**qed**

**lemma** *Ide-induct*:  
**assumes**  $\bigwedge t. R.\text{ide } t \implies P [t]$   
**and**  $\bigwedge t T. \llbracket R.\text{ide } t; R.\text{targets } t \subseteq \text{Srcs } T; P T \rrbracket \implies P (t \# T)$   
**shows**  $\text{Ide } T \implies P T$   
**proof** (*induct T*)  
  **show**  $\text{Ide } [] \implies P []$   
  **using** *Ide.simps(1)* **by** *blast*  
  **show**  $\bigwedge t T. \llbracket \text{Ide } T \implies P T; \text{Ide } (t \# T) \rrbracket \implies P (t \# T)$   
  **by** (*metis assms Ide.simps(2-3) list.exhaust*)  
**qed**

**lemma** *set-Arr-subset-arr*:  
**assumes**  $\text{Arr } T$   
**shows**  $\text{set } T \subseteq \text{Collect } R.\text{arr}$   
  **apply** (*induct T rule: Arr-induct*)  
  **using** *assms Arr.elims(2)* **by** *auto*

**lemma** *Arr-imp-arr-hd [simp]*:  
**assumes**  $\text{Arr } T$   
**shows**  $R.\text{arr } (\text{hd } T)$   
  **using** *assms*  
  **by** (*metis Arr.simps(1) CollectD hd-in-set set-Arr-subset-arr subset-code(1)*)

**lemma** *Arr-imp-arr-last [simp]*:  
**assumes**  $\text{Arr } T$   
**shows**  $R.\text{arr } (\text{last } T)$   
  **using** *assms*  
  **by** (*metis Arr.simps(1) CollectD in-mono last-in-set set-Arr-subset-arr*)

**lemma** *Arr-imp-Arr-tl [simp]*:  
**assumes**  $\text{Arr } T$  **and**  $\text{tl } T \neq []$   
**shows**  $\text{Arr } (\text{tl } T)$   
  **using** *assms*  
  **by** (*metis Arr.simps(3) list.exhaust-sel list.sel(2)*)

**lemma** *set-Ide-subset-ide*:  
**assumes**  $\text{Ide } T$

**shows**  $set\ T \subseteq Collect\ R.ide$   
**apply** (*induct*  $T$  *rule: Ide-induct*)  
**using** *assms* **by** *auto*

**lemma** *Ide-imp-Ide-hd* [*simp*]:  
**assumes**  $Ide\ T$   
**shows**  $R.ide\ (hd\ T)$   
**using** *assms*  
**by** (*metis*  $Ide.simps(1)$  *CollectD hd-in-set set-Ide-subset-ide subset-code(1)*)

**lemma** *Ide-imp-Ide-last* [*simp*]:  
**assumes**  $Ide\ T$   
**shows**  $R.ide\ (last\ T)$   
**using** *assms*  
**by** (*metis*  $Ide.simps(1)$  *CollectD in-mono last-in-set set-Ide-subset-ide*)

**lemma** *Ide-imp-Ide-tl* [*simp*]:  
**assumes**  $Ide\ T$  **and**  $tl\ T \neq []$   
**shows**  $Ide\ (tl\ T)$   
**using** *assms*  
**by** (*metis*  $Ide.simps(3)$  *list.exhaust-sel list.sel(2)*)

**lemma** *Ide-implies-Arr*:  
**assumes**  $Ide\ T$   
**shows**  $Arr\ T$   
**apply** (*induct*  $T$  *rule: Ide-induct*)  
**using** *assms*  
**apply** *auto*[3]  
**by** (*metis*  $Arr.elims(2)$   $Arr.simps(3)$  *R.ide-implies-arr*)

**lemma** *const-ide-is-Ide*:  
**shows**  $[T \neq []; R.ide\ (hd\ T); set\ T \subseteq \{hd\ T\}] \implies Ide\ T$   
**apply** (*induct*  $T$ )  
**apply** *auto*[2]  
**by** (*metis*  $Ide.simps(2-3)$   $R.ideE$   $R.sources-resid$   $Srcs.simps(2-3)$  *empty-iff insert-iff list.exhaust-sel list.set-sel(1) order-refl subset-singletonD*)

**lemma** *Ide-char*:  
**shows**  $Ide\ T \longleftrightarrow Arr\ T \wedge set\ T \subseteq Collect\ R.ide$   
**apply** (*induct*  $T$ )  
**apply** *auto*[1]  
**by** (*metis*  $Arr.simps(3)$   $Ide.simps(2-3)$  *Ide-implies-Arr empty-subsetI insert-subset list.simps(15) mem-Collect-eq neq-Nil-conv set-empty*)

**lemma** *IdeI* [*intro*]:  
**assumes**  $Arr\ T$  **and**  $set\ T \subseteq Collect\ R.ide$   
**shows**  $Ide\ T$   
**using** *assms* *Ide-char* **by** *force*

**lemma** *Arr-has-Src*:  
**shows**  $Arr\ T \implies Srcs\ T \neq \{\}$   
**apply** (*cases*  $T$ )  
**apply** *simp*  
**by** (*metis*  $R.arr\text{-}iff\text{-}has\text{-}source\ Srcs.elims\ Arr.elims(2)\ list.distinct(1)\ list.sel(1)$ )

**lemma** *Arr-has-Trg*:  
**shows**  $Arr\ T \implies Trgs\ T \neq \{\}$   
**using**  $R.arr\text{-}iff\text{-}has\text{-}target$   
**apply** (*induct*  $T$ )  
**apply** *simp*  
**by** (*metis*  $Arr.simps(2)\ Arr.simps(3)\ Trgs.simps(2-3)\ list.exhaust\text{-}sel$ )

**lemma** *Srcs-are-ide*:  
**shows**  $Srcs\ T \subseteq Collect\ R.ide$   
**apply** (*cases*  $T$ )  
**apply** *simp*  
**by** (*metis* (*no-types, lifting*)  $Srcs.elims\ list.distinct(1)\ mem\text{-}Collect\text{-}eq\ R.sources\text{-}def\ subsetI$ )

**lemma** *Trgs-are-ide*:  
**shows**  $Trgs\ T \subseteq Collect\ R.ide$   
**apply** (*induct*  $T$ )  
**apply** *simp*  
**by** (*metis*  $R.arr\text{-}iff\text{-}has\text{-}target\ R.sources\text{-}resid\ Srcs.simps(2)\ Trgs.simps(2-3)\ Srcs\text{-}are\text{-}ide\ empty\text{-}subsetI\ list.exhaust\ R.arrE$ )

**lemma** *Srcs-are-con*:  
**assumes**  $a \in Srcs\ T$  **and**  $a' \in Srcs\ T$   
**shows**  $a \frown a'$   
**using** *assms*  
**by** (*metis*  $Srcs.elims\ empty\text{-}iff\ R.sources\text{-}are\text{-}con$ )

**lemma** *Srcs-con-closed*:  
**assumes**  $a \in Srcs\ T$  **and**  $R.ide\ a'$  **and**  $a \frown a'$   
**shows**  $a' \in Srcs\ T$   
**using** *assms*  $R.sources\text{-}con\text{-}closed$   
**apply** (*cases*  $T$ , *auto*)  
**by** (*metis*  $Srcs.simps(2-3)\ neq\text{-}Nil\text{-}conv$ )

**lemma** *Srcs-eqI*:  
**assumes**  $Srcs\ T \cap Srcs\ T' \neq \{\}$   
**shows**  $Srcs\ T = Srcs\ T'$   
**using** *assms*  $R.sources\text{-}eqI$   
**apply** (*cases*  $T$ ; *cases*  $T'$ )  
**apply** *auto*  
**apply** (*metis*  $IntI\ Srcs.simps(2-3)\ empty\text{-}iff\ neq\text{-}Nil\text{-}conv$ )  
**by** (*metis*  $Srcs.simps(2-3)\ assms\ neq\text{-}Nil\text{-}conv$ )

**lemma** *Trgs-are-con*:  
**shows**  $\llbracket b \in \text{Trgs } T; b' \in \text{Trgs } T \rrbracket \implies b \frown b'$   
**apply** (*induct* *T*)  
**apply** *auto*  
**by** (*metis* *R.targets-are-con* *Trgs.simps(2-3)* *list.exhaust-sel*)

**lemma** *Trgs-con-closed*:  
**shows**  $\llbracket b \in \text{Trgs } T; R.\text{ide } b'; b \frown b' \rrbracket \implies b' \in \text{Trgs } T$   
**apply** (*induct* *T*)  
**apply** *auto*  
**by** (*metis* *R.targets-con-closed* *Trgs.simps(2-3)* *neq-Nil-conv*)

**lemma** *Trgs-eqI*:  
**assumes**  $\text{Trgs } T \cap \text{Trgs } T' \neq \{\}$   
**shows**  $\text{Trgs } T = \text{Trgs } T'$   
**using** *assms* *Trgs-are-ide* *Trgs-are-con* *Trgs-con-closed* **by** *blast*

**lemma** *Srcs-simpP*:  
**assumes** *Arr* *T*  
**shows**  $\text{Srcs } T = R.\text{sources } (\text{hd } T)$   
**using** *assms*  
**by** (*metis* *Arr-has-Src* *Srcs.simps(1)* *Srcs.simps(2)* *Srcs.simps(3)* *list.exhaust-sel*)

**lemma** *Trgs-simpP*:  
**shows**  $\text{Arr } T \implies \text{Trgs } T = R.\text{targets } (\text{last } T)$   
**apply** (*induct* *T*)  
**apply** *simp*  
**by** (*metis* *Arr.simps(3)* *Trgs.simps(2)* *Trgs.simps(3)* *last-ConsL* *last-ConsR* *neq-Nil-conv*)

### 2.4.1 Residuation on Paths

It was more difficult than I thought to get a correct formal definition for residuation on paths and to prove things from it. Straightforward attempts to write a single recursive definition ran into problems with being able to prove termination, as well as getting the cases correct so that the domain of definition was symmetric. Eventually I found the definition below, which simplifies the termination proof to some extent through the use of two auxiliary functions, and which has a symmetric form that makes symmetry easier to prove. However, there was still some difficulty in proving the recursive expansions with respect to cons and append that I needed.

The following defines residuation of a single transition along a path, yielding a transition.

```
fun Resid1x (infix  $\langle^1 \setminus^* \rangle$  70)
where  $t^1 \setminus^* [] = R.\text{null}$ 
      |  $t^1 \setminus^* [u] = t \setminus u$ 
      |  $t^1 \setminus^* (u \# U) = (t \setminus u)^1 \setminus^* U$ 
```

Next, we have residuation of a path along a single transition, yielding a path.

```
fun Residx1 (infix  $\langle^* \setminus^1 \rangle$  70)
```

**where**  $\square^{*\setminus^1} u = \square$   
 $| [t]^{*\setminus^1} u = (\text{if } t \frown u \text{ then } [t \setminus u] \text{ else } \square)$   
 $| (t \# T)^{*\setminus^1} u =$   
 $(\text{if } t \frown u \wedge T^{*\setminus^1} (u \setminus t) \neq \square \text{ then } (t \setminus u) \# T^{*\setminus^1} (u \setminus t) \text{ else } \square)$

Finally, residuation of a path along a path, yielding a path.

**function** (*sequential*) *Resid* (**infix**  $\langle^{*\setminus^*} \rangle$  70)  
**where**  $\square^{*\setminus^*} - = \square$   
 $| -^{*\setminus^*} \square = \square$   
 $| [t]^{*\setminus^*} [u] = (\text{if } t \frown u \text{ then } [t \setminus u] \text{ else } \square)$   
 $| [t]^{*\setminus^*} (u \# U) =$   
 $(\text{if } t \frown u \wedge (t \setminus u)^{1\setminus^*} U \neq R.\text{null} \text{ then } [(t \setminus u)^{1\setminus^*} U] \text{ else } \square)$   
 $| (t \# T)^{*\setminus^*} [u] =$   
 $(\text{if } t \frown u \wedge T^{*\setminus^1} (u \setminus t) \neq \square \text{ then } (t \setminus u) \# (T^{*\setminus^1} (u \setminus t)) \text{ else } \square)$   
 $| (t \# T)^{*\setminus^*} (u \# U) =$   
 $(\text{if } t \frown u \wedge (t \setminus u)^{1\setminus^*} U \neq R.\text{null} \wedge$   
 $(T^{*\setminus^1} (u \setminus t))^{*\setminus^*} (U^{*\setminus^1} (t \setminus u)) \neq \square$   
 $\text{then } (t \setminus u)^{1\setminus^*} U \# (T^{*\setminus^1} (u \setminus t))^{*\setminus^*} (U^{*\setminus^1} (t \setminus u))$   
 $\text{else } \square)$   
**by** *pat-completeness auto*

Residuation of a path along a single transition is length non-increasing. Actually, it is length-preserving, except in case the path and the transition are not consistent. We will show that later, but for now this is what we need to establish termination for  $(\setminus)$ .

**lemma** *length-Residx1*:  
**shows**  $\text{length } (T^{*\setminus^1} u) \leq \text{length } T$   
**proof** (*induct* *T arbitrary*: *u*)  
**show**  $\bigwedge u. \text{length } (\square^{*\setminus^1} u) \leq \text{length } \square$   
**by** *simp*  
**fix** *t T u*  
**assume** *ind*:  $\bigwedge u. \text{length } (T^{*\setminus^1} u) \leq \text{length } T$   
**show**  $\text{length } ((t \# T)^{*\setminus^1} u) \leq \text{length } (t \# T)$   
**using** *ind*  
**by** (*cases* *T*, *cases*  $t \frown u$ , *cases*  $T^{*\setminus^1} (u \setminus t)$ ) *auto*  
**qed**

**termination** *Resid*  
**proof** (*relation measure*  $(\lambda(T, U). \text{length } T + \text{length } U)$ )  
**show** *wf* (*measure*  $(\lambda(T, U). \text{length } T + \text{length } U)$ )  
**by** *simp*  
**fix** *t t' T u U*  
**have**  $\text{length } ((t' \# T)^{*\setminus^1} (u \setminus t)) + \text{length } (U^{*\setminus^1} (t \setminus u))$   
 $< \text{length } (t \# t' \# T) + \text{length } (u \# U)$   
**using** *length-Residx1*  
**by** (*metis* *add-less-le-mono impossible-Cons le-neq-implies-less list.size(4) trans-le-add1*)  
**thus** *1*:  $((t' \# T)^{*\setminus^1} (u \setminus t), U^{*\setminus^1} (t \setminus u), t \# t' \# T, u \# U)$   
 $\in \text{measure } (\lambda(T, U). \text{length } T + \text{length } U)$   
**by** *simp*  
**show**  $((t' \# T)^{*\setminus^1} (u \setminus t), U^{*\setminus^1} (t \setminus u), t \# t' \# T, u \# U)$

$\in \text{measure } (\lambda(T, U). \text{length } T + \text{length } U)$   
**using** 1 *length-Residx1* **by** *blast*  
**have**  $\text{length } (T^* \setminus^1 (u \setminus t)) + \text{length } (U^* \setminus^1 (t \setminus u)) \leq \text{length } T + \text{length } U$   
**using** *length-Residx1* **by** (*simp add: add-mono*)  
**thus** 2:  $((T^* \setminus^1 (u \setminus t), U^* \setminus^1 (t \setminus u)), t \# T, u \# U)$   
 $\in \text{measure } (\lambda(T, U). \text{length } T + \text{length } U)$   
**by** *simp*  
**show**  $((T^* \setminus^1 (u \setminus t), U^* \setminus^1 (t \setminus u)), t \# T, u \# U)$   
 $\in \text{measure } (\lambda(T, U). \text{length } T + \text{length } U)$   
**using** 2 *length-Residx1* **by** *blast*  
**qed**

**lemma** *Resid1x-null*:  
**shows**  $R.\text{null } 1 \setminus^* T = R.\text{null}$   
**apply** (*induct T*)  
**apply** *auto*  
**by** (*metis R.null-is-zero(1) Resid1x.simps(2-3) list.exhaust*)

**lemma** *Resid1x-ide*:  
**shows**  $\llbracket R.\text{ide } a; a \setminus^1 T \neq R.\text{null} \rrbracket \implies R.\text{ide } (a \setminus^1 T)$   
**proof** (*induct T arbitrary: a*)  
**show**  $\bigwedge a. a \setminus^1 \square \neq R.\text{null} \implies R.\text{ide } (a \setminus^1 \square)$   
**by** *simp*  
**fix**  $a \ t \ T$   
**assume**  $a: R.\text{ide } a$   
**assume**  $\text{ind}: \bigwedge a. \llbracket R.\text{ide } a; a \setminus^1 T \neq R.\text{null} \rrbracket \implies R.\text{ide } (a \setminus^1 T)$   
**assume**  $\text{con}: a \setminus^1 (t \# T) \neq R.\text{null}$   
**have** 1:  $a \frown t$   
**using** *con*  
**by** (*metis R.con-def Resid1x.simps(2-3) Resid1x-null list.exhaust*)  
**show**  $R.\text{ide } (a \setminus^1 (t \# T))$   
**using** *a 1 con ind R.resid-ide-arr*  
**by** (*metis Resid1x.simps(2-3) list.exhaust*)  
**qed**

**abbreviation** *Con* (**infix**  $\langle^* \frown^* \rangle$  50)  
**where**  $T^* \frown^* U \equiv T^* \setminus^* U \neq \square$

**lemma** *Con-sym1*:  
**shows**  $T^* \setminus^1 u \neq \square \longleftrightarrow u \setminus^1 T \neq R.\text{null}$   
**proof** (*induct T arbitrary: u*)  
**show**  $\bigwedge u. \square^* \setminus^1 u \neq \square \longleftrightarrow u \setminus^1 \square \neq R.\text{null}$   
**by** *simp*  
**show**  $\bigwedge t \ T \ u. (\bigwedge u. T^* \setminus^1 u \neq \square \longleftrightarrow u \setminus^1 T \neq R.\text{null})$   
 $\implies (t \# T)^* \setminus^1 u \neq \square \longleftrightarrow u \setminus^1 (t \# T) \neq R.\text{null}$   
**proof** –  
**fix**  $t \ T \ u$   
**assume**  $\text{ind}: \bigwedge u. T^* \setminus^1 u \neq \square \longleftrightarrow u \setminus^1 T \neq R.\text{null}$

```

show (t # T) *1 u ≠ [] ↔ u 1 \* (t # T) ≠ R.null
proof
  show (t # T) *1 u ≠ [] ⇒ u 1 \* (t # T) ≠ R.null
    by (metis R.con-sym Resid1x.simps(2-3) Resid1x.simps(2-3)
        ind neq-Nil-conv R.conE)
  show u 1 \* (t # T) ≠ R.null ⇒ (t # T) *1 u ≠ []
    using ind R.con-sym
    apply (cases T)
    apply auto
    by (metis R.conI Resid1x-null)
qed
qed
qed

```

**lemma** *Con-sym-ind*:

**shows**  $\text{length } T + \text{length } U \leq n \implies T * \frown^* U \longleftrightarrow U * \frown^* T$

**proof** (induct n arbitrary: T U)

**show**  $\bigwedge T U. \text{length } T + \text{length } U \leq 0 \implies T * \frown^* U \longleftrightarrow U * \frown^* T$   
**by** simp

**fix** n **and** T U :: 'a list

**assume** ind:  $\bigwedge T U. \text{length } T + \text{length } U \leq n \implies T * \frown^* U \longleftrightarrow U * \frown^* T$

**assume** 1:  $\text{length } T + \text{length } U \leq \text{Suc } n$

**show**  $T * \frown^* U \longleftrightarrow U * \frown^* T$

**using** R.con-sym Con-sym1

**apply** (cases T; cases U)

**apply** auto[3]

**proof** -

**fix** t u T' U'

**assume** T:  $T = t \# T'$  **and** U:  $U = u \# U'$

**show**  $T * \frown^* U \longleftrightarrow U * \frown^* T$

**proof** (cases T' = [])

**show**  $T' = [] \implies T * \frown^* U \longleftrightarrow U * \frown^* T$

**using** T U Con-sym1 R.con-sym

**by** (cases U') auto

**show**  $T' \neq [] \implies T * \frown^* U \longleftrightarrow U * \frown^* T$

**proof** (cases U' = [])

**show**  $[[T' \neq []]; U' = []] \implies T * \frown^* U \longleftrightarrow U * \frown^* T$

**using** T U R.con-sym Con-sym1

**by** (cases T') auto

**show**  $[[T' \neq []]; U' \neq []] \implies T * \frown^* U \longleftrightarrow U * \frown^* T$

**proof** -

**assume** T':  $T' \neq []$  **and** U':  $U' \neq []$

**have** 2:  $\text{length } (U' * \frown^1 (t \setminus u)) + \text{length } (T' * \frown^1 (u \setminus t)) \leq n$

**proof** -

**have**  $\text{length } (U' * \frown^1 (t \setminus u)) + \text{length } (T' * \frown^1 (u \setminus t))$   
 $\leq \text{length } U' + \text{length } T'$

**by** (simp add: add-le-mono length-Resid1x1)

**also have** ...  $\leq \text{length } T' + \text{length } U'$

**using** T' add.commute not-less-eq-eq **by** auto

also have ...  $\leq n$   
 using 1  $T U$  by *simp*  
 finally show *?thesis* by *blast*  
 qed  
 show  $T * \frown * U \longleftrightarrow U * \frown * T$   
 proof  
 assume *Con*:  $T * \frown * U$   
 have  $\exists: t \frown u \wedge T' * \backslash^1 (u \setminus t) \neq [] \wedge (t \setminus u)^1 \backslash * U' \neq R.null \wedge$   
      $(T' * \backslash^1 (u \setminus t)) * \backslash * (U' * \backslash^1 (t \setminus u)) \neq []$   
 using *Con T U T' U' Con-sym1*  
 apply (*cases T'*; *cases U'*)  
 apply *simp-all*  
 by (*metis Resid.simps(1) Resid.simps(6) neg-Nil-conv*)  
 hence  $u \frown t \wedge U' * \backslash^1 (t \setminus u) \neq [] \wedge (u \setminus t)^1 \backslash * T' \neq R.null$   
 using *T' U' R.con-sym Con-sym1* by *simp*  
 moreover have  $(U' * \backslash^1 (t \setminus u)) * \backslash * (T' * \backslash^1 (u \setminus t)) \neq []$   
 using 2 3 *ind* by *simp*  
 ultimately show  $U * \frown * T$   
 using *T U T' U'*  
 by (*cases T'*; *cases U'*) *auto*  
 next  
 assume *Con*:  $U * \frown * T$   
 have  $\exists: u \frown t \wedge U' * \backslash^1 (t \setminus u) \neq [] \wedge (u \setminus t)^1 \backslash * T' \neq R.null \wedge$   
      $(U' * \backslash^1 (t \setminus u)) * \backslash * (T' * \backslash^1 (u \setminus t)) \neq []$   
 using *Con T U T' U' Con-sym1*  
 apply (*cases T'*; *cases U'*)  
 apply *auto*  
 apply *argo*  
 by *force*  
 hence  $t \frown u \wedge T' * \backslash^1 (u \setminus t) \neq [] \wedge (t \setminus u)^1 \backslash * U' \neq R.null$   
 using *T' U' R.con-sym Con-sym1* by *simp*  
 moreover have  $(T' * \backslash^1 (u \setminus t)) * \backslash * (U' * \backslash^1 (t \setminus u)) \neq []$   
 using 2 3 *ind* by *simp*  
 ultimately show  $T * \frown * U$   
 using *T U T' U'*  
 by (*cases T'*; *cases U'*) *auto*  
 qed  
 qed  
 qed  
 qed  
 qed  
 qed

**lemma** *Con-sym*:  
**shows**  $T * \frown * U \longleftrightarrow U * \frown * T$   
 using *Con-sym-ind* by *blast*

**lemma** *Residx1-as-Resid*:  
**shows**  $T * \backslash^1 u = T * \backslash * [u]$

**proof** (*induct T*)  
**show**  $\square \text{ }^*\backslash^1 u = \square \text{ }^*\backslash^* [u]$  **by** *simp*  
**fix**  $t T$   
**assume** *ind*:  $T \text{ }^*\backslash^1 u = T \text{ }^*\backslash^* [u]$   
**show**  $(t \# T) \text{ }^*\backslash^1 u = (t \# T) \text{ }^*\backslash^* [u]$   
**by** (*cases T*) *auto*  
**qed**

**lemma** *Resid1x-as-Resid'*:  
**shows**  $t \text{ }^1\backslash^* U = (\text{if } [t] \text{ }^*\backslash^* U \neq \square \text{ then } \text{hd } ([t] \text{ }^*\backslash^* U) \text{ else } R.\text{null})$   
**proof** (*induct U*)  
**show**  $t \text{ }^1\backslash^* \square = (\text{if } [t] \text{ }^*\backslash^* \square \neq \square \text{ then } \text{hd } ([t] \text{ }^*\backslash^* \square) \text{ else } R.\text{null})$  **by** *simp*  
**fix**  $u U$   
**assume** *ind*:  $t \text{ }^1\backslash^* U = (\text{if } [t] \text{ }^*\backslash^* U \neq \square \text{ then } \text{hd } ([t] \text{ }^*\backslash^* U) \text{ else } R.\text{null})$   
**show**  $t \text{ }^1\backslash^* (u \# U) = (\text{if } [t] \text{ }^*\backslash^* (u \# U) \neq \square \text{ then } \text{hd } ([t] \text{ }^*\backslash^* (u \# U)) \text{ else } R.\text{null})$   
**using** *Resid1x-null*  
**by** (*cases U*) *auto*  
**qed**

The following recursive expansion for consistency of paths is an intermediate result that is not yet quite in the form we really want.

**lemma** *Con-rec*:  
**shows**  $[t] \text{ }^*\frown^* [u] \longleftrightarrow t \frown u$   
**and**  $T \neq \square \implies t \# T \text{ }^*\frown^* [u] \longleftrightarrow t \frown u \wedge T \text{ }^*\frown^* [u \setminus t]$   
**and**  $U \neq \square \implies [t] \text{ }^*\frown^* (u \# U) \longleftrightarrow t \frown u \wedge [t \setminus u] \text{ }^*\frown^* U$   
**and**  $\llbracket T \neq \square; U \neq \square \rrbracket \implies$   

$$t \# T \text{ }^*\frown^* u \# U \longleftrightarrow t \frown u \wedge T \text{ }^*\frown^* [u \setminus t] \wedge [t \setminus u] \text{ }^*\frown^* U \wedge T \text{ }^*\backslash^* [u \setminus t] \text{ }^*\frown^* U \text{ }^*\backslash^* [t \setminus u]$$

**proof** –  
**show**  $[t] \text{ }^*\frown^* [u] \longleftrightarrow t \frown u$   
**by** *simp*  
**show**  $T \neq \square \implies t \# T \text{ }^*\frown^* [u] \longleftrightarrow t \frown u \wedge T \text{ }^*\frown^* [u \setminus t]$   
**using** *Resid1-as-Resid*  
**by** (*cases T*) *auto*  
**show**  $U \neq \square \implies [t] \text{ }^*\frown^* (u \# U) \longleftrightarrow t \frown u \wedge [t \setminus u] \text{ }^*\frown^* U$   
**using** *Resid1-as-Resid' Con-sym Con-sym1 Resid1x.simps(3) Resid1-as-Resid*  
**by** (*cases U*) *auto*  
**show**  $\llbracket T \neq \square; U \neq \square \rrbracket \implies$   

$$t \# T \text{ }^*\frown^* u \# U \longleftrightarrow t \frown u \wedge T \text{ }^*\frown^* [u \setminus t] \wedge [t \setminus u] \text{ }^*\frown^* U \wedge T \text{ }^*\backslash^* [u \setminus t] \text{ }^*\frown^* U \text{ }^*\backslash^* [t \setminus u]$$
  
**using** *Resid1-as-Resid Resid1-as-Resid' Con-sym1 Con-sym R.con-sym*  
**by** (*cases T; cases U*) *auto*  
**qed**

This version is a more appealing form of the previously proved fact *Resid1x-as-Resid'*.

**lemma** *Resid1x-as-Resid*:  
**assumes**  $[t] \text{ }^*\backslash^* U \neq \square$   
**shows**  $[t] \text{ }^*\backslash^* U = [t \text{ }^1\backslash^* U]$   
**using** *assms Con-rec(2,4)*

**apply** (*cases U; cases tl U*)  
**apply** *auto*  
**by** *argo+*

The following is an intermediate version of a recursive expansion for residuation, to be improved subsequently.

**lemma** *Resid-rec*:  
**shows** [*simp*]:  $[t] \text{ }^* \frown^* [u] \implies [t] \text{ }^* \setminus^* [u] = [t \setminus u]$   
**and**  $\llbracket T \neq []; t \# T \text{ }^* \frown^* [u] \rrbracket \implies (t \# T) \text{ }^* \setminus^* [u] = (t \setminus u) \# (T \text{ }^* \setminus^* [u \setminus t])$   
**and**  $\llbracket U \neq []; \text{Con } [t] (u \# U) \rrbracket \implies [t] \text{ }^* \setminus^* (u \# U) = [t \setminus u] \text{ }^* \setminus^* U$   
**and**  $\llbracket T \neq []; U \neq []; \text{Con } (t \# T) (u \# U) \rrbracket \implies$   
 $(t \# T) \text{ }^* \setminus^* (u \# U) = ([t \setminus u] \text{ }^* \setminus^* U) @ ((T \text{ }^* \setminus^* [u \setminus t]) \text{ }^* \setminus^* (U \text{ }^* \setminus^* [t \setminus u]))$   
**proof** –  
**show**  $[t] \text{ }^* \frown^* [u] \implies \text{Resid } [t] [u] = [t \setminus u]$   
**by** (*meson Resid.simps(3)*)  
**show**  $\llbracket T \neq []; t \# T \text{ }^* \frown^* [u] \rrbracket \implies (t \# T) \text{ }^* \setminus^* [u] = (t \setminus u) \# (T \text{ }^* \setminus^* [u \setminus t])$   
**using** *Resid1-as-Resid*  
**by** (*metis Resid1.simps(3) list.exhaust-sel*)  
**show** 1:  $\llbracket U \neq []; [t] \text{ }^* \frown^* u \# U \rrbracket \implies [t] \text{ }^* \setminus^* (u \# U) = [t \setminus u] \text{ }^* \setminus^* U$   
**by** (*metis Con-rec(3) Resid1x.simps(3) Resid1x-as-Resid list.exhaust*)  
**show**  $\llbracket T \neq []; U \neq []; t \# T \text{ }^* \frown^* u \# U \rrbracket \implies$   
 $(t \# T) \text{ }^* \setminus^* (u \# U) = ([t \setminus u] \text{ }^* \setminus^* U) @ ((T \text{ }^* \setminus^* [u \setminus t]) \text{ }^* \setminus^* (U \text{ }^* \setminus^* [t \setminus u]))$   
**proof** –  
**assume** *T: T ≠ [] and U: U ≠ [] and Con: Con (t # T) (u # U)*  
**have** *tu: t ∩ u*  
**using** *Con Con-rec by metis*  
**have**  $(t \# T) \text{ }^* \setminus^* (u \# U) = ((t \setminus u) \text{ }^1 \setminus^* U) \# ((T \text{ }^* \setminus^1 [u \setminus t]) \text{ }^* \setminus^* (U \text{ }^* \setminus^1 [t \setminus u]))$   
**using** *T U Con tu*  
**by** (*cases T; cases U*) *auto*  
**also have**  $\dots = ([t \setminus u] \text{ }^* \setminus^* U) @ ((T \text{ }^* \setminus^* [u \setminus t]) \text{ }^* \setminus^* (U \text{ }^* \setminus^* [t \setminus u]))$   
**using** *T U Con tu Con-rec(4) Resid1x-as-Resid Resid1-as-Resid by force*  
**finally show** *?thesis by simp*  
**qed**  
**qed**

For consistent paths, residuation is length-preserving.

**lemma** *length-Resid-ind*:  
**shows**  $\llbracket \text{length } T + \text{length } U \leq n; T \text{ }^* \frown^* U \rrbracket \implies \text{length } (T \text{ }^* \setminus^* U) = \text{length } T$   
**apply** (*induct n arbitrary: T U*)  
**apply** *simp*  
**proof** –  
**fix** *n T U*  
**assume** *ind:  $\bigwedge T U. \llbracket \text{length } T + \text{length } U \leq n; T \text{ }^* \frown^* U \rrbracket \implies \text{length } (T \text{ }^* \setminus^* U) = \text{length } T$*   
**assume** *Con: T ∩ U*  
**assume** *len: length T + length U ≤ Suc n*  
**show**  $\text{length } (T \text{ }^* \setminus^* U) = \text{length } T$   
**using** *Con len ind Resid1x-as-Resid length-Cons Con-rec(2) Resid-rec(2)*  
**apply** (*cases T; cases U*)

**apply** *auto*  
**apply** (*cases*  $tl\ T = []$ ; *cases*  $tl\ U = []$ )  
**apply** *auto*  
**apply** *metis*  
**apply** *fastforce*  
**proof** –  
**fix**  $t\ T'\ u\ U'$   
**assume**  $T: T = t \# T'$  **and**  $U: U = u \# U'$   
**assume**  $T': T' \neq []$  **and**  $U': U' \neq []$   
**show**  $length\ ((t \# T') * \setminus * (u \# U')) = Suc\ (length\ T')$   
**using** *Con Con-rec(4) Con-sym Resid-rec(4) T T' U U' ind len* **by** *auto*  
**qed**  
**qed**

**lemma** *length-Resid*:  
**assumes**  $T * \frown * U$   
**shows**  $length\ (T * \setminus * U) = length\ T$   
**using** *assms length-Resid-ind* **by** *auto*

**lemma** *Con-initial-left*:  
**shows**  $t \# T * \frown * U \implies [t] * \frown * U$   
**apply** (*induct*  $U$ )  
**apply** *simp*  
**by** (*metis Con-rec(1-4)*)

**lemma** *Con-initial-right*:  
**shows**  $T * \frown * u \# U \implies T * \frown * [u]$   
**apply** (*induct*  $T$ )  
**apply** *simp*  
**by** (*metis Con-rec(1-4)*)

**lemma** *Resid-cons-ind*:  
**shows**  $\llbracket T \neq []; U \neq []; length\ T + length\ U \leq n \rrbracket \implies$   
 $(\forall t. t \# T * \frown * U \longleftrightarrow [t] * \frown * U \wedge T * \frown * U * \setminus * [t]) \wedge$   
 $(\forall u. T * \frown * u \# U \longleftrightarrow T * \frown * [u] \wedge T * \setminus * [u] * \frown * U) \wedge$   
 $(\forall t. t \# T * \frown * U \longrightarrow (t \# T) * \setminus * U = [t] * \setminus * U @ T * \setminus * (U * \setminus * [t])) \wedge$   
 $(\forall u. T * \frown * u \# U \longrightarrow T * \setminus * (u \# U) = (T * \setminus * [u]) * \setminus * U)$   
**proof** (*induct*  $n$  *arbitrary*:  $T\ U$ )  
**show**  $\bigwedge T\ U. \llbracket T \neq []; U \neq []; length\ T + length\ U \leq 0 \rrbracket \implies$   
 $(\forall t. t \# T * \frown * U \longleftrightarrow [t] * \frown * U \wedge T * \frown * U * \setminus * [t]) \wedge$   
 $(\forall u. T * \frown * u \# U \longleftrightarrow T * \frown * [u] \wedge T * \setminus * [u] * \frown * U) \wedge$   
 $(\forall t. t \# T * \frown * U \longrightarrow (t \# T) * \setminus * U = [t] * \setminus * U @ T * \setminus * (U * \setminus * [t])) \wedge$   
 $(\forall u. T * \frown * u \# U \longrightarrow T * \setminus * (u \# U) = (T * \setminus * [u]) * \setminus * U)$   
**by** *simp*  
**fix**  $n$  **and**  $T\ U :: 'a\ list$   
**assume**  $ind: \bigwedge T\ U. \llbracket T \neq []; U \neq []; length\ T + length\ U \leq n \rrbracket \implies$   
 $(\forall t. t \# T * \frown * U \longleftrightarrow [t] * \frown * U \wedge T * \frown * U * \setminus * [t]) \wedge$   
 $(\forall u. T * \frown * u \# U \longleftrightarrow T * \frown * [u] \wedge T * \setminus * [u] * \frown * U) \wedge$   
 $(\forall t. t \# T * \frown * U \longrightarrow (t \# T) * \setminus * U = [t] * \setminus * U @ T * \setminus * (U * \setminus * [t])) \wedge$

$(\forall u. T^* \frown^* u \# U \longrightarrow T^* \backslash^* (u \# U) = (T^* \backslash^* [u])^* \backslash^* U)$   
**assume**  $T: T \neq []$  **and**  $U: U \neq []$   
**assume**  $len: length\ T + length\ U \leq Suc\ n$   
**show**  $(\forall t. t \# T^* \frown^* U \longleftrightarrow [t]^* \frown^* U \wedge T^* \frown^* U^* \backslash^* [t]) \wedge$   
 $(\forall u. T^* \frown^* u \# U \longleftrightarrow T^* \frown^* [u] \wedge T^* \backslash^* [u]^* \frown^* U) \wedge$   
 $(\forall t. t \# T^* \frown^* U \longrightarrow (t \# T)^* \backslash^* U = [t]^* \backslash^* U @ T^* \backslash^* (U^* \backslash^* [t])) \wedge$   
 $(\forall u. T^* \frown^* u \# U \longrightarrow T^* \backslash^* (u \# U) = (T^* \backslash^* [u])^* \backslash^* U)$   
**proof** (*intro allI conjI iffI impI*)  
**fix**  $t$   
**show**  $1: t \# T^* \frown^* U \implies (t \# T)^* \backslash^* U = [t]^* \backslash^* U @ T^* \backslash^* (U^* \backslash^* [t])$   
**proof** (*cases U*)  
**show**  $U = [] \implies ?thesis$   
**using**  $U$  **by** *simp*  
**fix**  $u\ U'$   
**assume**  $U: U = u \# U'$   
**assume**  $Con: t \# T^* \frown^* U$   
**show** *?thesis*  
**proof** (*cases U' = []*)  
**show**  $U' = [] \implies ?thesis$   
**using**  $T\ U\ Con\ R.con-sym\ Con-rec(2)\ Resid-rec(2)$  **by** *auto*  
**assume**  $U': U' \neq []$   
**have**  $(t \# T)^* \backslash^* U = [t \setminus u]^* \backslash^* U' @ (T^* \backslash^* [u \setminus t])^* \backslash^* (U'^* \backslash^* [t \setminus u])$   
**using**  $T\ U\ U'\ Con\ Resid-rec(4)$  **by** *fastforce*  
**also have**  $1: \dots = [t]^* \backslash^* U @ (T^* \backslash^* [u \setminus t])^* \backslash^* (U'^* \backslash^* [t \setminus u])$   
**using**  $T\ U\ U'\ Con\ Con-rec(3-4)\ Resid-rec(3)$  **by** *auto*  
**also have**  $\dots = [t]^* \backslash^* U @ T^* \backslash^* ((u \setminus t) \# (U'^* \backslash^* [t \setminus u]))$   
**proof**  $-$   
**have**  $T^* \backslash^* ((u \setminus t) \# (U'^* \backslash^* [t \setminus u])) = (T^* \backslash^* [u \setminus t])^* \backslash^* (U'^* \backslash^* [t \setminus u])$   
**using**  $T\ U\ U'\ ind\ [of\ T\ U'^* \backslash^* [t \setminus u]]\ Con\ Con-rec(4)\ Con-sym\ len\ length-Resid$   
**by** *fastforce*  
**thus** *?thesis* **by** *auto*  
**qed**  
**also have**  $\dots = [t]^* \backslash^* U @ T^* \backslash^* (U^* \backslash^* [t])$   
**using**  $T\ U\ U'\ 1\ Con\ Con-rec(4)\ Con-sym1\ Resid1-as-Resid$   
 $Resid1x-as-Resid\ Resid-rec(2)\ Con-sym\ Con-initial-left$   
**by** *auto*  
**finally show** *?thesis* **by** *simp*  
**qed**  
**qed**  
**show**  $t \# T^* \frown^* U \implies [t]^* \frown^* U$   
**by** (*simp add: Con-initial-left*)  
**show**  $t \# T^* \frown^* U \implies T^* \frown^* (U^* \backslash^* [t])$   
**by** (*metis 1 Suc-inject T append-Nil2 length-0-conv length-Cons length-Resid*)  
**show**  $[t]^* \frown^* U \wedge T^* \frown^* U^* \backslash^* [t] \implies t \# T^* \frown^* U$   
**proof** (*cases U*)  
**show**  $\llbracket [t]^* \frown^* U \wedge T^* \frown^* U^* \backslash^* [t]; U = [] \rrbracket \implies t \# T^* \frown^* U$   
**using**  $U$  **by** *simp*  
**fix**  $u\ U'$   
**assume**  $U: U = u \# U'$

```

assume Con:  $[t] \text{ }^* \frown \text{ }^* U \wedge T \text{ }^* \frown \text{ }^* U \text{ }^* \setminus \text{ }^* [t]$ 
show  $t \# T \text{ }^* \frown \text{ }^* U$ 
proof (cases  $U' = []$ )
  show  $U' = [] \implies ?thesis$ 
    using T U Con
    by (metis Con-rec(2) Resid.simps(3) R.con-sym)
  assume  $U': U' \neq []$ 
  show ?thesis
  proof –
    have  $t \frown u$ 
      using T U U' Con Con-rec(3) by blast
    moreover have  $T \text{ }^* \frown \text{ }^* [u \setminus t]$ 
      using T U U' Con Con-initial-right Con-sym1 Resid1x-as-Resid
        Resid1x-as-Resid Resid-rec(2)
      by (metis Con-sym)
    moreover have  $[t \setminus u] \text{ }^* \frown \text{ }^* U'$ 
      using T U U' Con Resid-rec(3) by force
    moreover have  $T \text{ }^* \setminus \text{ }^* [u \setminus t] \text{ }^* \frown \text{ }^* U' \text{ }^* \setminus \text{ }^* [t \setminus u]$ 
      by (metis (no-types, opaque-lifting) Con Con-sym Resid-rec(2) Suc-le-mono
        T U U' add-Suc-right calculation(3) ind len length-Cons length-Resid)
    ultimately show ?thesis
      using T U U' Con-rec(4) by simp
  qed
qed
qed
next
fix  $u$ 
show  $1: T \text{ }^* \frown \text{ }^* u \# U \implies T \text{ }^* \setminus \text{ }^* (u \# U) = (T \text{ }^* \setminus \text{ }^* [u]) \text{ }^* \setminus \text{ }^* U$ 
proof (cases  $T$ )
  show  $2: \llbracket T \text{ }^* \frown \text{ }^* u \# U; T = [] \rrbracket \implies T \text{ }^* \setminus \text{ }^* (u \# U) = (T \text{ }^* \setminus \text{ }^* [u]) \text{ }^* \setminus \text{ }^* U$ 
    using T by simp
  fix  $t T'$ 
  assume  $T: T = t \# T'$ 
  assume Con:  $T \text{ }^* \frown \text{ }^* u \# U$ 
  show ?thesis
  proof (cases  $T' = []$ )
    show  $T' = [] \implies ?thesis$ 
      using T U Con Con-rec(3) Resid1x-as-Resid Resid-rec(3) by force
    assume  $T': T' \neq []$ 
    have  $T \text{ }^* \setminus \text{ }^* (u \# U) = [t \setminus u] \text{ }^* \setminus \text{ }^* U @ (T' \text{ }^* \setminus \text{ }^* [u \setminus t]) \text{ }^* \setminus \text{ }^* (U \text{ }^* \setminus \text{ }^* [t \setminus u])$ 
      using T U T' Con Resid-rec(4) [of T' U t u] by simp
    also have  $\dots = ((t \setminus u) \# (T' \text{ }^* \setminus \text{ }^* [u \setminus t])) \text{ }^* \setminus \text{ }^* U$ 
    proof –
      have  $\text{length } (T' \text{ }^* \setminus \text{ }^* [u \setminus t]) + \text{length } U \leq n$ 
        by (metis (no-types, lifting) Con Con-rec(4) One-nat-def Suc-eq-plus1 Suc-leI
          T T' U add-Suc le-less-trans len length-Resid lessI list.size(4)
          not-le)
      thus ?thesis
        using ind [of T' \text{ }^* \setminus \text{ }^* [u \setminus t] U] Con Con-rec(4) T T' U by auto

```

**qed**  
**also have**  $\dots = (T \text{ * } \backslash \text{ * } [u]) \text{ * } \backslash \text{ * } U$   
**using**  $T \ U \ T' \ Con \ Con\text{-}rec(2,4) \ Resid\text{-}rec(2)$  **by force**  
**finally show** *?thesis* **by simp**  
**qed**  
**qed**  
**show**  $T \text{ * } \frown \text{ * } u \# U \implies T \text{ * } \frown \text{ * } [u]$   
**using** 1 **by force**  
**show**  $T \text{ * } \frown \text{ * } u \# U \implies T \text{ * } \backslash \text{ * } [u] \text{ * } \frown \text{ * } U$   
**using** 1 **by fastforce**  
**show**  $T \text{ * } \frown \text{ * } [u] \wedge T \text{ * } \backslash \text{ * } [u] \text{ * } \frown \text{ * } U \implies T \text{ * } \frown \text{ * } u \# U$   
**proof** (*cases T*)  
**show**  $\llbracket T \text{ * } \frown \text{ * } [u] \wedge T \text{ * } \backslash \text{ * } [u] \text{ * } \frown \text{ * } U; T = [] \rrbracket \implies T \text{ * } \frown \text{ * } u \# U$   
**using**  $T$  **by simp**  
**fix**  $t \ T'$   
**assume**  $T: T = t \# T'$   
**assume**  $Con: T \text{ * } \frown \text{ * } [u] \wedge T \text{ * } \backslash \text{ * } [u] \text{ * } \frown \text{ * } U$   
**show**  $Con \ T \ (u \# U)$   
**proof** (*cases T' = []*)  
**show**  $T' = [] \implies ?thesis$   
**using**  $Con \ T \ U \ Con\text{-}rec(1,3)$  **by auto**  
**assume**  $T': T' \neq []$   
**have**  $t \frown u$   
**using**  $Con \ T \ U \ T' \ Con\text{-}rec(2)$  **by blast**  
**moreover have** 2:  $T' \text{ * } \frown \text{ * } [u \setminus t]$   
**using**  $Con \ T \ U \ T' \ Con\text{-}rec(2)$  **by blast**  
**moreover have**  $[t \setminus u] \text{ * } \frown \text{ * } U$   
**using**  $Con \ T \ U \ T'$   
**by** (*metis Con-initial-left Resid-rec(2)*)  
**moreover have**  $T' \text{ * } \backslash \text{ * } [u \setminus t] \text{ * } \frown \text{ * } U \text{ * } \backslash \text{ * } [t \setminus u]$   
**proof** –  
**have** 0:  $length \ (U \text{ * } \backslash \text{ * } [t \setminus u]) = length \ U$   
**using**  $Con \ T \ U \ T' \ length\text{-}Resid \ Con\text{-}sym \ calculation(3)$  **by blast**  
**hence** 1:  $length \ T' + length \ (U \text{ * } \backslash \text{ * } [t \setminus u]) \leq n$   
**using**  $Con \ T \ U \ T' \ len \ length\text{-}Resid \ Con\text{-}sym$  **by simp**  
**have**  $length \ ((T \text{ * } \backslash \text{ * } [u]) \text{ * } \backslash \text{ * } U) =$   
 $length \ ([t \setminus u] \text{ * } \backslash \text{ * } U) + length \ ((T' \text{ * } \backslash \text{ * } [u \setminus t]) \text{ * } \backslash \text{ * } (U \text{ * } \backslash \text{ * } [t \setminus u]))$   
**proof** –  
**have**  $(T \text{ * } \backslash \text{ * } [u]) \text{ * } \backslash \text{ * } U =$   
 $[t \setminus u] \text{ * } \backslash \text{ * } U \ @ \ (T' \text{ * } \backslash \text{ * } [u \setminus t]) \text{ * } \backslash \text{ * } (U \text{ * } \backslash \text{ * } [t \setminus u])$   
**by** (*metis 0 1 2 Con Resid-rec(2) T T' U ind length-Resid*)  
**thus** *?thesis*  
**using**  $Con \ T \ U \ T' \ length\text{-}Resid$  **by simp**  
**qed**  
**moreover have**  $length \ ((T \text{ * } \backslash \text{ * } [u]) \text{ * } \backslash \text{ * } U) = length \ T$   
**using**  $Con \ T \ U \ T' \ length\text{-}Resid$  **by metis**  
**moreover have**  $length \ ([t \setminus u] \text{ * } \backslash \text{ * } U) \leq 1$   
**using**  $Con \ T \ U \ T' \ Resid1x\text{-}as\text{-}Resid$   
**by** (*metis One-nat-def length-Cons list.size(3) order-refl zero-le*)

```

ultimately show ?thesis
  using Con T U T' length-Resid by auto
qed
ultimately show T *^* u # U
  using T Con-rec(4) [of T' U t u] by fastforce
qed
qed
qed
qed

```

The following are the final versions of recursive expansion for consistency and residuation on paths. These are what I really wanted the original definitions to look like, but if this is tried, then *Con* and *Resid* end up having to be mutually recursive, expressing the definitions so that they are single-valued becomes an issue, and proving termination is more problematic.

```

lemma Con-cons:
  assumes T ≠ [] and U ≠ []
  shows t # T *^* U ↔ [t] *^* U ∧ T *^* U * \ [t]
  and T *^* u # U ↔ T *^* [u] ∧ T * \ [u] *^* U
  using assms Resid-cons-ind [of T U] by blast+

```

```

lemma Con-consI [intro, simp]:
  shows [[T ≠ []; U ≠ []; [t] *^* U; T *^* U * \ [t]] ⇒ t # T *^* U
  and [[T ≠ []; U ≠ []; T *^* [u]; T * \ [u] *^* U] ⇒ T *^* u # U
  using Con-cons by auto

```

```

lemma Resid-cons:
  assumes U ≠ []
  shows t # T *^* U ⇒ (t # T) * \ U = ([t] * \ U) @ (T * \ (U * \ [t]))
  and T *^* u # U ⇒ T * \ (u # U) = (T * \ [u]) * \ U
  using assms Resid-cons-ind [of T U] Resid.simps(1)
  by blast+

```

The following expansion of residuation with respect to the first argument is stated in terms of the more primitive cons, rather than list append, but as a result  $^1 \setminus^*$  has to be used.

```

lemma Resid-cons':
  assumes T ≠ []
  shows t # T *^* U ⇒ (t # T) * \ U = (t ^1 \ U) # (T * \ (U * \ [t]))
  using assms
  by (metis Con-sym Resid.simps(1) Resid1x-as-Resid Resid-cons(1)
  append-Cons append-Nil)

```

```

lemma Srcs-Resid-Arr-single:
  assumes T *^* [u]
  shows Srcs (T * \ [u]) = R.targets u
  proof (cases T)
    show T = [] ⇒ Srcs (T * \ [u]) = R.targets u
  
```

```

    using assms by simp
  fix  $t T'$ 
  assume  $T: T = t \# T'$ 
  show  $\text{Srcs } (T * \setminus * [u]) = R.\text{targets } u$ 
  proof (cases  $T' = []$ )
    show  $T' = [] \implies ?thesis$ 
      using assms  $T$  R.sources-resid by auto
    assume  $T': T' \neq []$ 
    have  $\text{Srcs } (T * \setminus * [u]) = \text{Srcs } ((t \# T') * \setminus * [u])$ 
      using  $T$  by simp
    also have  $\dots = \text{Srcs } ((t \setminus u) \# (T' * \setminus * ([u] * \setminus * T')))$ 
      using assms  $T$ 
      by (metis Resid-rec(2) Srcs.elims  $T'$  list.distinct(1) list.sel(1))
    also have  $\dots = R.\text{sources } (t \setminus u)$ 
      using Srcs.elims by blast
    also have  $\dots = R.\text{targets } u$ 
      using assms Con-rec(2)  $T T'$  R.sources-resid by force
    finally show  $?thesis$  by blast
  qed
qed

```

```

lemma Srcs-Resid-single-Arr:
shows  $[u] * \frown * T \implies \text{Srcs } ([u] * \setminus * T) = \text{Trgs } T$ 
proof (induct  $T$  arbitrary:  $u$ )
  show  $\bigwedge u. [u] * \frown * [] \implies \text{Srcs } ([u] * \setminus * []) = \text{Trgs } []$ 
    by simp
  fix  $t u T$ 
  assume ind:  $\bigwedge u. [u] * \frown * T \implies \text{Srcs } ([u] * \setminus * T) = \text{Trgs } T$ 
  assume Con:  $[u] * \frown * t \# T$ 
  show  $\text{Srcs } ([u] * \setminus * (t \# T)) = \text{Trgs } (t \# T)$ 
  proof (cases  $T = []$ )
    show  $T = [] \implies ?thesis$ 
      using Con Srcs-Resid-Arr-single Trgs.simps(2) by presburger
    assume  $T: T \neq []$ 
    have  $\text{Srcs } ([u] * \setminus * (t \# T)) = \text{Srcs } ([u \setminus t] * \setminus * T)$ 
      using Con Resid-rec(3)  $T$  by force
    also have  $\dots = \text{Trgs } T$ 
      using Con ind Con-rec(3)  $T$  by auto
    also have  $\dots = \text{Trgs } (t \# T)$ 
      by (metis  $T$  Trgs.elims Trgs.simps(3))
    finally show  $?thesis$  by simp
  qed
qed

```

```

lemma Trgs-Resid-sym-Arr-single:
shows  $T * \frown * [u] \implies \text{Trgs } (T * \setminus * [u]) = \text{Trgs } ([u] * \setminus * T)$ 
proof (induct  $T$  arbitrary:  $u$ )
  show  $\bigwedge u. [] * \frown * [u] \implies \text{Trgs } ([] * \setminus * [u]) = \text{Trgs } ([u] * \setminus * [])$ 
    by simp

```

```

fix t u T
assume ind:  $\bigwedge u. T * \frown * [u] \implies \text{Trgs } (T * \backslash * [u]) = \text{Trgs } ([u] * \backslash * T)$ 
assume Con:  $t \# T * \frown * [u]$ 
show  $\text{Trgs } ((t \# T) * \backslash * [u]) = \text{Trgs } ([u] * \backslash * (t \# T))$ 
proof (cases T = [])
  show  $T = [] \implies ?thesis$ 
    using R.targets-resid-sym
    by (simp add: R.con-sym)
  assume T:  $T \neq []$ 
  show ?thesis
  proof -
    have  $\text{Trgs } ((t \# T) * \backslash * [u]) = \text{Trgs } ((t \setminus u) \# (T * \backslash * [u \setminus t]))$ 
      using Con Resid-rec(2) T by auto
    also have  $\dots = \text{Trgs } (T * \backslash * [u \setminus t])$ 
      using T Con Con-rec(2) [of T t u]
      by (metis Trgs.elims Trgs.simps(3))
    also have  $\dots = \text{Trgs } ([u \setminus t] * \backslash * T)$ 
      using T Con ind Con-sym by metis
    also have  $\dots = \text{Trgs } ([u] * \backslash * (t \# T))$ 
      using T Con Con-sym Resid-rec(3) by presburger
    finally show ?thesis by blast
  qed
qed
qed

lemma Srcs-Resid [simp]:
shows  $T * \frown * U \implies \text{Srcs } (T * \backslash * U) = \text{Trgs } U$ 
proof (induct U arbitrary: T)
  show  $\bigwedge T. T * \frown * [] \implies \text{Srcs } (T * \backslash * []) = \text{Trgs } []$ 
    using Con-sym Resid.simps(1) by blast
  fix u U T
  assume ind:  $\bigwedge T. T * \frown * U \implies \text{Srcs } (T * \backslash * U) = \text{Trgs } U$ 
  assume Con:  $T * \frown * u \# U$ 
  show  $\text{Srcs } (T * \backslash * (u \# U)) = \text{Trgs } (u \# U)$ 
    by (metis Con Resid-cons(2) Srcs-Resid-Arr-single Trgs.simps(2-3) ind
      list.exhaust-sel)
qed

lemma Trgs-Resid-sym [simp]:
shows  $T * \frown * U \implies \text{Trgs } (T * \backslash * U) = \text{Trgs } (U * \backslash * T)$ 
proof (induct U arbitrary: T)
  show  $\bigwedge T. T * \frown * [] \implies \text{Trgs } (T * \backslash * []) = \text{Trgs } ([] * \backslash * T)$ 
    by (meson Con-sym Resid.simps(1))
  fix u U T
  assume ind:  $\bigwedge T. T * \frown * U \implies \text{Trgs } (T * \backslash * U) = \text{Trgs } (U * \backslash * T)$ 
  assume Con:  $T * \frown * u \# U$ 
  show  $\text{Trgs } (T * \backslash * (u \# U)) = \text{Trgs } ((u \# U) * \backslash * T)$ 
proof (cases U = [])
  show  $U = [] \implies ?thesis$ 

```

```

    using Con Trgs-Resid-sym-Arr-single by blast
  assume U: U ≠ []
  show ?thesis
  proof -
    have Trgs (T * \ $\setminus$ * (u # U)) = Trgs ((T * \ $\setminus$ * [u]) * \ $\setminus$ * U)
      using U by (metis Con Resid-cons(2))
    also have ... = Trgs (U * \ $\setminus$ * (T * \ $\setminus$ * [u]))
      using U Con by (metis Con-sym ind)
    also have ... = Trgs ((u # U) * \ $\setminus$ * T)
      by (metis (no-types, opaque-lifting) Con-cons(1) Con-sym Resid.simps(1) Resid-cons'
        Trgs.simps(3) U neq-Nil-conv)
    finally show ?thesis by simp
  qed
qed
qed

```

**lemma** *img-Resid-Srcs*:

**shows**  $Arr\ T \implies (\lambda a. [a] * \ $\setminus$ * T) \text{ ' } Srcs\ T \subseteq (\lambda b. [b]) \text{ ' } Trgs\ T$

**proof** (*induct* T)

**show**  $Arr\ [] \implies (\lambda a. [a] * \ $\setminus$ * []) \text{ ' } Srcs\ [] \subseteq (\lambda b. [b]) \text{ ' } Trgs\ []$

**by** *simp*

**fix** *t* :: 'a **and** *T* :: 'a list

**assume** *tT*:  $Arr\ (t \# T)$

**assume** *ind*:  $Arr\ T \implies (\lambda a. [a] * \ $\setminus$ * T) \text{ ' } Srcs\ T \subseteq (\lambda b. [b]) \text{ ' } Trgs\ T$

**show**  $(\lambda a. [a] * \ $\setminus$ * (t \# T)) \text{ ' } Srcs\ (t \# T) \subseteq (\lambda b. [b]) \text{ ' } Trgs\ (t \# T)$

**proof**

**fix** *B*

**assume** *B*:  $B \in (\lambda a. [a] * \ $\setminus$ * (t \# T)) \text{ ' } Srcs\ (t \# T)$

**show**  $B \in (\lambda b. [b]) \text{ ' } Trgs\ (t \# T)$

**proof** (*cases* T = [])

**assume** *T*: T = []

**obtain** *a* **where** *a*:  $a \in R.sources\ t \wedge [a \setminus t] = B$

**by** (*metis* (*no-types*, *lifting*) *B* *R.composite-of-source-arr* *R.con-prfx-composite-of(1)* *Resid-rec(1)* *Srcs.simps(2)* *T* *Arr.simps(2)* *Con-rec(1)* *imageE* *tT*)

**have**  $a \setminus t \in Trgs\ (t \# T)$

**using** *tT* *T* *a*

**by** (*simp* *add*: *R.resid-source-in-targets*)

**thus** ?thesis

**using** *B* *a* *image-iff* **by** *fastforce*

**next**

**assume** *T*: T ≠ []

**obtain** *a* **where** *a*:  $a \in R.sources\ t \wedge [a] * \ $\setminus$ * (t \# T) = B$

**using** *tT* *T* *B* *Srcs.elims* **by** *blast*

**have**  $[a \setminus t] * \ $\setminus$ * T = B$

**using** *tT* *T* *B* *a*

**by** (*metis* *Con-rec(3)* *R.arrI* *R.resid-source-in-targets* *R.targets-are-cong*

*Resid-rec(3)* *R.arr-resid-iff-con* *R.ide-implies-arr*)

**moreover** **have**  $a \setminus t \in Srcs\ T$

**using** *a* *tT*

by (metis Arr.simps(3) R.resid-source-in-targets T neq-Nil-conv subsetD)  
 ultimately show ?thesis  
 using T tT ind  
 by (metis Trgs.simps(3) Arr.simps(3) image-iff list.exhaust-sel subsetD)  
 qed  
 qed  
 qed

lemma Resid-Arr-Src:

shows  $\llbracket \text{Arr } T; a \in \text{Srcs } T \rrbracket \implies T \text{ }^* \setminus^* [a] = T$

proof (induct T arbitrary: a)

show  $\bigwedge a. \llbracket \text{Arr } []; a \in \text{Srcs } [] \rrbracket \implies [] \text{ }^* \setminus^* [a] = []$

by simp

fix a t T

assume ind:  $\bigwedge a. \llbracket \text{Arr } T; a \in \text{Srcs } T \rrbracket \implies T \text{ }^* \setminus^* [a] = T$

assume Arr:  $\text{Arr } (t \# T)$

assume a:  $a \in \text{Srcs } (t \# T)$

show  $(t \# T) \text{ }^* \setminus^* [a] = t \# T$

proof (cases  $T = []$ )

show  $T = [] \implies ?thesis$

using a R.resid-arr-ide R.sources-def by auto

assume T:  $T \neq []$

show  $(t \# T) \text{ }^* \setminus^* [a] = t \# T$

proof -

have 1:  $R.\text{arr } t \wedge \text{Arr } T \wedge R.\text{targets } t \subseteq \text{Srcs } T$

using Arr T

by (metis Arr.elims(2) list.sel(1) list.sel(3))

have 2:  $t \# T \text{ }^* \setminus^* [a]$

using T a Arr Con-rec(2)

by (metis (no-types, lifting) img-Resid-Srcs Con-sym imageE image-subset-iff list.distinct(1))

have  $(t \# T) \text{ }^* \setminus^* [a] = (t \setminus a) \# (T \text{ }^* \setminus^* [a \setminus t])$

using 2 T Resid-rec(2) by simp

moreover have  $t \setminus a = t$

using Arr a R.sources-def

by (metis 2 CollectD Con-rec(2) T Srcs-are-ide in-mono R.resid-arr-ide)

moreover have  $T \text{ }^* \setminus^* [a \setminus t] = T$

by (metis 1 2 R.in-sourcesI R.resid-source-in-targets Srcs-are-ide T a Con-rec(2) in-mono ind mem-Collect-eq)

ultimately show ?thesis by simp

qed

qed

qed

lemma Con-single-ide-ind:

shows  $R.\text{ide } a \implies [a] \text{ }^* \setminus^* T \longleftrightarrow \text{Arr } T \wedge a \in \text{Srcs } T$

proof (induct T arbitrary: a)

show  $\bigwedge a. [a] \text{ }^* \setminus^* [] \longleftrightarrow \text{Arr } [] \wedge a \in \text{Srcs } []$

by simp

```

fix a t T
assume ind:  $\bigwedge a. R.ide\ a \implies [a] * \frown^* T \longleftrightarrow Arr\ T \wedge a \in Srcs\ T$ 
assume a:  $R.ide\ a$ 
show  $[a] * \frown^* (t \# T) \longleftrightarrow Arr\ (t \# T) \wedge a \in Srcs\ (t \# T)$ 
proof (cases  $T = []$ )
  show  $T = [] \implies ?thesis$ 
    using a Con-sym
    by (metis Arr.simps(2) Resid-Arr-Src Srcs.simps(2) R.arr-iff-has-source
      Con-rec(1) empty-iff R.in-sourcesI list.distinct(1))
  assume T:  $T \neq []$ 
  have 1:  $[a] * \frown^* (t \# T) \longleftrightarrow a \frown t \wedge [a \setminus t] * \frown^* T$ 
    using a T Con-cons(2) [of [a] T t] by simp
  also have 2:  $\dots \longleftrightarrow a \frown t \wedge Arr\ T \wedge a \setminus t \in Srcs\ T$ 
    using a T ind R.resid-ide-arr by blast
  also have  $\dots \longleftrightarrow Arr\ (t \# T) \wedge a \in Srcs\ (t \# T)$ 
    using a T Con-sym R.con-sym Resid-Arr-Src R.con-implies-arr Srcs-are-ide
  apply (cases T)
  apply simp
  by (metis Arr.simps(3) R.resid-arr-ide R.targets-resid-sym Srcs.simps(3)
    Srcs-Resid-Arr-single calculation dual-order.eq-iff list.distinct(1)
    R.in-sourcesI)
  finally show ?thesis by simp
qed
qed

```

**lemma** Con-single-ide-iff:

```

assumes R.ide a
shows  $[a] * \frown^* T \longleftrightarrow Arr\ T \wedge a \in Srcs\ T$ 
  using assms Con-single-ide-ind by simp

```

**lemma** Con-single-ideI [intro]:

```

assumes R.ide a and Arr T and  $a \in Srcs\ T$ 
shows  $[a] * \frown^* T$  and  $T * \frown^* [a]$ 
  using assms Con-single-ide-iff Con-sym by auto

```

**lemma** Resid-single-ide:

```

assumes R.ide a and  $[a] * \frown^* T$ 
shows  $[a] * \setminus^* T \in (\lambda b. [b]) \text{ ' Trgs } T$  and [simp]:  $T * \setminus^* [a] = T$ 
  using assms Con-single-ide-ind img-Resid-Srcs Resid-Arr-Src Con-sym
  by blast+

```

**lemma** Resid-Arr-Ide-ind:

```

shows  $\llbracket Ide\ A; T * \frown^* A \rrbracket \implies T * \setminus^* A = T$ 
proof (induct A)
  show  $\llbracket Ide\ []; T * \frown^* [] \rrbracket \implies T * \setminus^* [] = T$ 
    by simp
  fix a A
  assume ind:  $\llbracket Ide\ A; T * \frown^* A \rrbracket \implies T * \setminus^* A = T$ 
  assume Ide:  $Ide\ (a \# A)$ 

```

```

assume Con:  $T \frown^* a \# A$ 
show  $T \backslash^* (a \# A) = T$ 
  by (metis (no-types, lifting) Con Con-initial-left Con-sym Ide Ide.elims(2)
    Resid-cons(2) Resid-single-ide(2) ind list.inject)
qed

lemma Resid-Ide-Arr-ind:
shows  $\llbracket \text{Ide } A; A \frown^* T \rrbracket \implies \text{Ide } (A \backslash^* T)$ 
proof (induct A)
  show  $\llbracket \text{Ide } []; [] \frown^* T \rrbracket \implies \text{Ide } ([] \backslash^* T)$ 
    by simp
  fix a A
  assume ind:  $\llbracket \text{Ide } A; A \frown^* T \rrbracket \implies \text{Ide } (A \backslash^* T)$ 
  assume Ide: Ide (a  $\#$  A)
  assume Con: a  $\#$   $A \frown^* T$ 
  have T: Arr T
    using Con Ide Con-single-ide-ind Con-initial-left Ide.elims(2)
    by blast
  show Ide ( $(a \# A) \backslash^* T$ )
  proof (cases  $A = []$ )
    show  $A = [] \implies ?thesis$ 
      by (metis Con Con-sym1 Ide Ide.simps(2) Resid1x-as-Resid Resid1x-ide
        Residx1-as-Resid Con-sym)
    assume A:  $A \neq []$ 
    show ?thesis
    proof –
      have Ide ( $[a] \backslash^* T$ )
        by (metis Con Con-initial-left Con-sym Con-sym1 Ide Ide.simps(3)
          Resid1x-as-Resid Residx1-as-Resid Ide.simps(2) Resid1x-ide
          list.exhaust-sel)
      moreover have Trgs ( $[a] \backslash^* T$ )  $\subseteq$  Srcs ( $A \backslash^* T$ )
        using A T Ide Con
        by (metis (no-types, lifting) Con-sym Ide.elims(2) Ide.simps(2) Resid-Arr-Ide-ind
          Srcs-Resid Trgs-Resid-sym Con-cons(2) dual-order.eq-iff list.inject)
      moreover have Ide ( $A \backslash^* (T \backslash^* [a])$ )
        by (metis A Con Con-cons(1) Con-sym Ide Ide.simps(3) Resid-Arr-Ide-ind
          Resid-single-ide(2) ind list.exhaust-sel)
      moreover have Ide ( $(a \# A) \backslash^* T$ )  $\longleftrightarrow$ 
         $\text{Ide } ([a] \backslash^* T) \wedge \text{Ide } (A \backslash^* (T \backslash^* [a])) \wedge$ 
         $\text{Trgs } ([a] \backslash^* T) \subseteq \text{Srcs } (A \backslash^* T)$ 
        using calculation(1–3)
      by (metis Arr.simps(1) Con Ide Ide.simps(3) Resid1x-as-Resid Resid-cons'
        Trgs.simps(2) Con-single-ide-iff Ide.simps(2) Ide-implies-Arr Resid-Arr-Src
        list.exhaust-sel)
      ultimately show ?thesis by blast
    qed
  qed
qed

```

**lemma** *Resid-Ide*:  
**assumes** *Ide A* **and**  $A \text{ }^* \frown^* T$   
**shows**  $T \text{ }^* \backslash^* A = T$  **and** *Ide (A \text{ }^\* \backslash^\* T)*  
**using** *assms Resid-Ide-Arr-ind Resid-Arr-Ide-ind Con-sym* **by** *auto*

**lemma** *Con-Ide-iff*:  
**shows** *Ide A*  $\implies A \text{ }^* \frown^* T \longleftrightarrow \text{Arr } T \wedge \text{Srcs } T = \text{Srcs } A$   
**proof** (*induct A*)  
**show** *Ide []*  $\implies [] \text{ }^* \frown^* T \longleftrightarrow \text{Arr } T \wedge \text{Srcs } T = \text{Srcs } []$   
**by** *simp*  
**fix** *a A*  
**assume** *ind: Ide A*  $\implies A \text{ }^* \frown^* T \longleftrightarrow \text{Arr } T \wedge \text{Srcs } T = \text{Srcs } A$   
**assume** *Ide: Ide (a # A)*  
**show**  $a \# A \text{ }^* \frown^* T \longleftrightarrow \text{Arr } T \wedge \text{Srcs } T = \text{Srcs } (a \# A)$   
**proof** (*cases A = []*)  
**show**  $A = [] \implies ?thesis$   
**using** *Con-single-ide-ind Ide*  
**by** (*metis Arr.simps(2) Con-sym Ide.simps(2) Ide-implies-Arr R.arrE Resid-Arr-Src Srcs.simps(2) Srcs-Resid R.in-sourcesI*)  
**assume** *A: A  $\neq$  []*  
**have**  $a \# A \text{ }^* \frown^* T \longleftrightarrow [a] \text{ }^* \frown^* T \wedge A \text{ }^* \frown^* T \text{ }^* \backslash^* [a]$   
**using** *A Ide Con-cons(1) [of A T a]* **by** *fastforce*  
**also have**  $1: \dots \longleftrightarrow \text{Arr } T \wedge a \in \text{Srcs } T$   
**by** (*metis A Arr-has-Src Con-single-ide-ind Ide Ide.elims(2) Resid-Arr-Src Srcs-Resid-Arr-single Con-sym Srcs-eqI ind inf.absorb-iff2 list.inject*)  
**also have**  $\dots \longleftrightarrow \text{Arr } T \wedge \text{Srcs } T = \text{Srcs } (a \# A)$   
**by** (*metis A 1 Con-sym Ide Ide.simps(3) R.ideE R.sources-resid Resid-Arr-Src Srcs.simps(3) Srcs-Resid-Arr-single list.exhaust-sel R.in-sourcesI*)  
**finally show**  $a \# A \text{ }^* \frown^* T \longleftrightarrow \text{Arr } T \wedge \text{Srcs } T = \text{Srcs } (a \# A)$   
**by** *blast*  
**qed**  
**qed**

**lemma** *Con-IdeI*:  
**assumes** *Ide A* **and** *Arr T* **and**  $\text{Srcs } T = \text{Srcs } A$   
**shows**  $A \text{ }^* \frown^* T$  **and**  $T \text{ }^* \frown^* A$   
**using** *assms Con-Ide-iff Con-sym* **by** *auto*

**lemma** *Con-Arr-self*:  
**shows** *Arr T*  $\implies T \text{ }^* \frown^* T$   
**proof** (*induct T*)  
**show** *Arr []*  $\implies [] \text{ }^* \frown^* []$   
**by** *simp*  
**fix** *t T*  
**assume** *ind: Arr T*  $\implies T \text{ }^* \frown^* T$   
**assume** *Arr: Arr (t # T)*  
**show**  $t \# T \text{ }^* \frown^* t \# T$   
**proof** (*cases T = []*)

```

show  $T = [] \implies ?thesis$ 
  using Arr R.arrE by simp
assume  $T: T \neq []$ 
have  $t \frown t \wedge T^* \frown^* [t \setminus t] \wedge [t \setminus t]^* \frown^* T \wedge T^* \setminus^* [t \setminus t]^* \frown^* T^* \setminus^* [t \setminus t]$ 
proof -
  have  $t \frown t$ 
    using Arr Arr.elims(1) by auto
  moreover have  $T^* \frown^* [t \setminus t]$ 
  proof -
    have Ide  $[t \setminus t]$ 
      by (simp add: R.arr-def R.prfx-reflexive calculation)
    moreover have  $\text{Srcs } [t \setminus t] = \text{Srcs } T$ 
      by (metis Arr Arr.simps(2) Arr-has-Trg R.arrE R.sources-resid Srcs.simps(2) Srcs-eqI T Trgs.simps(2) Arr.simps(3) inf.absorb-iff2 list.exhaust)
    ultimately show ?thesis
      by (metis Arr Con-sym T Arr.simps(3) Con-Ide-iff neq-Nil-conv)
  qed
  ultimately show ?thesis
    by (metis Con-single-ide-ind Con-sym R.prfx-reflexive Resid-single-ide(2) ind R.con-implies-arr(1))
  qed
  thus ?thesis
    using Con-rec(4) [of  $T T t t$ ] by force
  qed
qed

```

**lemma** *Resid-Arr-self*:

**shows**  $\text{Arr } T \implies \text{Ide } (T^* \setminus^* T)$

**proof** (*induct T*)

show  $\text{Arr } [] \implies \text{Ide } ([]^* \setminus^* [])$

by *simp*

**fix**  $t T$

**assume** *ind*:  $\text{Arr } T \implies \text{Ide } (T^* \setminus^* T)$

**assume** *Arr*:  $\text{Arr } (t \# T)$

**show**  $\text{Ide } ((t \# T)^* \setminus^* (t \# T))$

**proof** (*cases T = []*)

show  $T = [] \implies ?thesis$

using *Arr R.prfx-reflexive* by *auto*

assume  $T: T \neq []$

**have**  $1: (t \# T)^* \setminus^* (t \# T) = t^1 \setminus^* (t \# T) \# T^* \setminus^* ((t \# T)^* \setminus^* [t])$

using *Arr T Resid-cons'* [of  $T t t \# T$ ] *Con-Arr-self* by *presburger*

**also have**  $\dots = (t \setminus t)^1 \setminus^* T \# T^* \setminus^* (t^1 \setminus^* [t] \# T^* \setminus^* ([t]^* \setminus^* [t]))$

using *Arr T Resid-cons'* [of  $T t [t]$ ]

by (*metis Con-initial-right Resid1x.simps(3) calculation neq-Nil-conv*)

**also have**  $\dots = (t \setminus t)^1 \setminus^* T \# (T^* \setminus^* ([t]^* \setminus^* [t]))^* \setminus^* (T^* \setminus^* ([t]^* \setminus^* [t]))$

by (*metis 1 Resid1x.simps(2) Resid1x.simps(2) Resid1-as-Resid T calculation*)

*Con-cons(1) Con-rec(4) Resid-cons(2) list.distinct(1) list.inject*

**finally have**  $2: (t \# T)^* \setminus^* (t \# T) =$

$(t \setminus t)^1 \setminus^* T \# (T^* \setminus^* ([t]^* \setminus^* [t]))^* \setminus^* (T^* \setminus^* ([t]^* \setminus^* [t]))$

```

    by blast
  moreover have Ide ...
  proof -
    have R.ide ((t \ t) 1\* T)
      using Arr T
    by (metis Con-initial-right Con-rec(2) Con-sym1 R.con-implies-arr(1)
        Resid1x-ide Con-Arr-self Resid1x-as-Resid R.prfx-reflexive)
    moreover have Ide ((T *\* ([t] *\* [t])) *\* (T *\* ([t] *\* [t])))
      using Arr T
    by (metis Con-Arr-self Con-rec(4) Resid-single-ide(2) Con-single-ide-ind
        Resid.simps(3) ind R.prfx-reflexive R.con-implies-arr(2))
    moreover have R.targets ((t \ t) 1\* T)  $\subseteq$ 
      Srcs ((T *\* ([t] *\* [t])) *\* (T *\* ([t] *\* [t])))
    by (metis (no-types, lifting) 1 2 Con-cons(1) Resid1x-as-Resid T Trgs.simps(2)
        Trgs-Resid-sym Srcs-Resid dual-order.eq-iff list.discI list.inject)
    ultimately show ?thesis
      using Arr T
      by (metis Ide.simps(1,3) list.exhaust-sel)
  qed
  ultimately show ?thesis by auto
  qed
  qed

```

```

lemma Con-imp-eq-Srcs:
  assumes T *\*  $\frown$  U
  shows Srcs T = Srcs U
  proof (cases T)
    show T = []  $\implies$  ?thesis
      using assms by simp
    fix t T'
    assume T: T = t # T'
    show Srcs T = Srcs U
  proof (cases U)
    show U = []  $\implies$  ?thesis
      using assms T by simp
    fix u U'
    assume U: U = u # U'
    show Srcs T = Srcs U
      by (metis Con-initial-right Con-rec(1) Con-sym R.con-imp-common-source
          Srcs.simps(2-3) Srcs-eqI T Trgs.cases U assms)
  qed
  qed

```

```

lemma Arr-iff-Con-self:
  shows Arr T  $\longleftrightarrow$  T *\*  $\frown$  T
  proof (induct T)
    show Arr []  $\longleftrightarrow$  [] *\*  $\frown$  []
      by simp
    fix t T

```

```

assume ind:  $Arr\ T \longleftrightarrow T^* \frown^* T$ 
show  $Arr\ (t \# T) \longleftrightarrow t \# T^* \frown^* t \# T$ 
proof (cases  $T = []$ )
  show  $T = [] \implies ?thesis$ 
    by auto
  assume  $T: T \neq []$ 
  show ?thesis
proof
  show  $Arr\ (t \# T) \implies t \# T^* \frown^* t \# T$ 
    using Con-Arr-self by simp
  show  $t \# T^* \frown^* t \# T \implies Arr\ (t \# T)$ 
proof -
  assume Con:  $t \# T^* \frown^* t \# T$ 
  have R.arr  $t$ 
    using T Con Con-rec(4) [of T T t] by blast
  moreover have Arr  $T$ 
    using T Con Con-rec(4) [of T T t] ind R.arrI
    by (meson R.prfx-reflexive Con-single-ide-ind)
  moreover have R.targets  $t \subseteq Srcs\ T$ 
    using T Con
    by (metis Con-cons(2) Con-imp-eq-Srcs Trgs.simps(2)
      Srcs-Resid list.distinct(1) subsetI)
  ultimately show ?thesis
    by (cases T) auto
  qed
qed
qed
qed

```

**lemma** *Arr-Resid-single*:

```

shows  $T^* \frown^* [u] \implies Arr\ (T^* \setminus^* [u])$ 
proof (induct T arbitrary: u)
  show  $\bigwedge u. []^* \frown^* [u] \implies Arr\ ([]^* \setminus^* [u])$ 
    by simp
  fix  $t\ u\ T$ 
  assume ind:  $\bigwedge u. T^* \frown^* [u] \implies Arr\ (T^* \setminus^* [u])$ 
  assume Con:  $t \# T^* \frown^* [u]$ 
  show  $Arr\ ((t \# T)^* \setminus^* [u])$ 
proof (cases T = [])
  show  $T = [] \implies ?thesis$ 
    using Con Arr-iff-Con-self R.con-imp-arr-resid Con-rec(1) by fastforce
  assume  $T: T \neq []$ 
  have  $Arr\ ((t \# T)^* \setminus^* [u]) \longleftrightarrow Arr\ ((t \setminus u) \# (T^* \setminus^* [u \setminus t]))$ 
    using Con T Resid-rec(2) by auto
  also have  $\dots \longleftrightarrow R.arr\ (t \setminus u) \wedge Arr\ (T^* \setminus^* [u \setminus t]) \wedge$ 
     $R.targets\ (t \setminus u) \subseteq Srcs\ (T^* \setminus^* [u \setminus t])$ 
    using Con T
    by (metis Arr.simps(3) Con-rec(2) neq-Nil-conv)
  also have  $\dots \longleftrightarrow R.con\ t\ u \wedge Arr\ (T^* \setminus^* [u \setminus t])$ 

```

**using** *Con T*  
**by** (*metis Srcs-Resid-Arr-single Con-rec(2) R.arr-resid-iff-con subsetI R.targets-resid-sym*)  
**also have** ...  $\longleftrightarrow$  *True*  
**using** *Con ind T Con-rec(2)* **by** *blast*  
**finally show** *?thesis* **by** *auto*  
**qed**  
**qed**

**lemma** *Con-imp-Arr-Resid*:  
**shows**  $T * \frown * U \implies \text{Arr } (T * \backslash * U)$   
**proof** (*induct U arbitrary: T*)  
**show**  $\bigwedge T. T * \frown * [] \implies \text{Arr } (T * \backslash * [])$   
**by** (*meson Con-sym Resid.simps(1)*)  
**fix**  $u U T$   
**assume** *ind*:  $\bigwedge T. T * \frown * U \implies \text{Arr } (T * \backslash * U)$   
**assume** *Con*:  $T * \frown * u \# U$   
**show**  $\text{Arr } (T * \backslash * (u \# U))$   
**by** (*metis Arr-Resid-single Con Resid-cons(2) ind*)  
**qed**

**lemma** *Cube-ind*:  
**shows**  $\llbracket T * \frown * U; V * \frown * T; \text{length } T + \text{length } U + \text{length } V \leq n \rrbracket \implies$   
 $(V * \backslash * T * \frown * U * \backslash * T \longleftrightarrow V * \backslash * U * \frown * T * \backslash * U) \wedge$   
 $(V * \backslash * T * \frown * U * \backslash * T \longrightarrow$   
 $(V * \backslash * T) * \backslash * (U * \backslash * T) = (V * \backslash * U) * \backslash * (T * \backslash * U))$   
**proof** (*induct n arbitrary: T U V*)  
**show**  $\bigwedge T U V. \llbracket T * \frown * U; V * \frown * T; \text{length } T + \text{length } U + \text{length } V \leq 0 \rrbracket \implies$   
 $(V * \backslash * T * \frown * U * \backslash * T \longleftrightarrow V * \backslash * U * \frown * T * \backslash * U) \wedge$   
 $(V * \backslash * T * \frown * U * \backslash * T \longrightarrow$   
 $(V * \backslash * T) * \backslash * (U * \backslash * T) = (V * \backslash * U) * \backslash * (T * \backslash * U))$   
**by** *simp*  
**fix**  $n$  **and**  $T U V$  **::** 'a list  
**assume** *Con-TU*:  $T * \frown * U$  **and** *Con-VT*:  $V * \frown * T$   
**have**  $T: T \neq []$   
**using** *Con-TU* **by** *auto*  
**have**  $U: U \neq []$   
**using** *Con-TU Con-sym Resid.simps(1)* **by** *blast*  
**have**  $V: V \neq []$   
**using** *Con-VT* **by** *auto*  
**assume** *len*:  $\text{length } T + \text{length } U + \text{length } V \leq \text{Suc } n$   
**assume** *ind*:  $\bigwedge T U V. \llbracket T * \frown * U; V * \frown * T; \text{length } T + \text{length } U + \text{length } V \leq n \rrbracket \implies$   
 $(V * \backslash * T * \frown * U * \backslash * T \longleftrightarrow V * \backslash * U * \frown * T * \backslash * U) \wedge$   
 $(V * \backslash * T * \frown * U * \backslash * T \longrightarrow$   
 $(V * \backslash * T) * \backslash * (U * \backslash * T) = (V * \backslash * U) * \backslash * (T * \backslash * U))$   
**show**  $(V * \backslash * T * \frown * U * \backslash * T \longleftrightarrow V * \backslash * U * \frown * T * \backslash * U) \wedge$   
 $(V * \backslash * T * \frown * U * \backslash * T \longrightarrow (V * \backslash * T) * \backslash * (U * \backslash * T) = (V * \backslash * U) * \backslash * (T * \backslash * U))$   
**proof** (*cases V*)  
**show**  $V = [] \implies ?thesis$

using  $V$  by *simp*

fix  $v V'$   
 assume  $V: V = v \# V'$   
 show *?thesis*  
 proof (cases  $U$ )  
 show  $U = [] \implies ?thesis$   
 using  $U$  by *simp*  
 fix  $u U'$   
 assume  $U: U = u \# U'$   
 show *?thesis*  
 proof (cases  $T$ )  
 show  $T = [] \implies ?thesis$   
 using  $T$  by *simp*  
 fix  $t T'$   
 assume  $T: T = t \# T'$   
 show *?thesis*  
 proof (cases  $V' = []$ , cases  $U' = []$ , cases  $T' = []$ )  
 show  $[V' = []; U' = []; T' = []] \implies ?thesis$   
 using  $T U V R.cube Con-TU Resid.simps(2-3) R.arr-resid-iff-con$   
 $R.con-implies-arr Con-sym$   
 by *metis*  
 assume  $T': T' \neq []$  and  $V': V' = []$  and  $U': U' = []$   
 have 1:  $U \text{ * } \frown \text{ * } [t]$   
 using  $T Con-TU Con-cons(2) Con-sym Resid.simps(2)$  by *metis*  
 have 2:  $V \text{ * } \frown \text{ * } [t]$   
 using  $V Con-VT Con-initial-right T$  by *blast*  
 show *?thesis*  
 proof (intro *conjI impI*)  
 have  $\beta: \text{length } [t] + \text{length } U + \text{length } V \leq n$   
 using  $T T' le-Suc-eq len$  by *fastforce*  
 show \*:  $V \text{ * } \setminus \text{ * } T \text{ * } \frown \text{ * } U \text{ * } \setminus \text{ * } T \longleftrightarrow V \text{ * } \setminus \text{ * } U \text{ * } \frown \text{ * } T \text{ * } \setminus \text{ * } U$   
 proof –  
 have  $V \text{ * } \setminus \text{ * } T \text{ * } \frown \text{ * } U \text{ * } \setminus \text{ * } T \longleftrightarrow (V \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } T' \text{ * } \frown \text{ * } (U \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } T'$   
 using  $Con-TU Con-VT Con-sym Resid-cons(2) T T'$  by *force*  
 also have ...  $\longleftrightarrow V \text{ * } \setminus \text{ * } [t] \text{ * } \frown \text{ * } U \text{ * } \setminus \text{ * } [t] \wedge$   
 $(V \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } (U \text{ * } \setminus \text{ * } [t]) \text{ * } \frown \text{ * } T' \text{ * } \setminus \text{ * } (U \text{ * } \setminus \text{ * } [t])$   
 proof (intro *iffI conjI*)  
 show  $(V \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } T' \text{ * } \frown \text{ * } (U \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } T' \implies V \text{ * } \setminus \text{ * } [t] \text{ * } \frown \text{ * } U \text{ * } \setminus \text{ * } [t]$   
 using  $T U V T' U' V' 1 ind [of T'] len Con-TU Con-rec(2) Resid-rec(1)$   
 $Resid.simps(1) length-Cons Suc-le-mono add-Suc$   
 by (*metis (no-types)*)  
 show  $(V \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } T' \text{ * } \frown \text{ * } (U \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } T' \implies$   
 $(V \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } (U \text{ * } \setminus \text{ * } [t]) \text{ * } \frown \text{ * } T' \text{ * } \setminus \text{ * } (U \text{ * } \setminus \text{ * } [t])$   
 using  $T U V T' U' V'$   
 by (*metis Con-sym Resid.simps(1) Resid-rec(1) Suc-le-mono ind len*  
 $length-Cons list.size(3-4)$ )  
 show  $V \text{ * } \setminus \text{ * } [t] \text{ * } \frown \text{ * } U \text{ * } \setminus \text{ * } [t] \wedge$   
 $(V \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } (U \text{ * } \setminus \text{ * } [t]) \text{ * } \frown \text{ * } T' \text{ * } \setminus \text{ * } (U \text{ * } \setminus \text{ * } [t]) \implies$

$(V * [t]) * T' \frown (U * [t]) * T'$   
**using**  $T U V T' U' V' 1 \text{ ind len Con-TU Con-VT Con-rec}(1-3)$   
**by** (*metis* (*no-types*, *lifting*) *One-nat-def Resid-rec*(1) *Suc-le-mono*  
*add.commute list.size*(3) *list.size*(4) *plus-1-eq-Suc*)

qed

**also have**  $\dots \longleftrightarrow (V * U) * ([t] * U) \frown T' * (U * [t])$   
**by** (*metis* 2 3 *Con-sym ind Resid.simps*(1))

**also have**  $\dots \longleftrightarrow V * U \frown T * U$   
**using** *Con-rec*(2) [*of T' t*]  
**by** (*metis* (*no-types*, *lifting*) 1 *Con-TU Con-cons*(2) *Resid.simps*(1)  
*Resid.simps*(3) *Resid-rec*(2)  $T T' U U'$ )

**finally show** *?thesis* **by** *simp*

qed

**assume**  $\text{Con}: V * T \frown U * T$   
**show**  $(V * T) * (U * T) = (V * U) * (T * U)$   
**proof** –

**have**  $(V * T) * (U * T) = ((V * [t]) * T') * ((U * [t]) * T')$   
**using** *Con-TU Con-VT Con-sym Resid-cons*(2)  $T T'$  **by** *force*

**also have**  $\dots = ((V * [t]) * (U * [t])) * (T' * (U * [t]))$   
**using**  $T U V T' U' V' 1 \text{ Con ind [of T' Resid U [t] Resid V [t]]}$   
**by** (*metis* *One-nat-def add.commute calculation len length-0-conv length-Resid*  
*list.size*(4) *nat-add-left-cancel-le Con-sym plus-1-eq-Suc*)

**also have**  $\dots = ((V * U) * ([t] * U)) * (T' * (U * [t]))$   
**by** (*metis* 1 2 3 *Con-sym ind*)

**also have**  $\dots = (V * U) * (T * U)$   
**using**  $T U T' U' \text{ Con } *$   
**by** (*metis* *Con-sym Resid-rec*(1-2) *Resid.simps*(1) *Resid-cons*(2))

**finally show** *?thesis* **by** *simp*

qed

next

**assume**  $U': U' \neq []$  **and**  $V': V' = []$   
**show** *?thesis*  
**proof** (*intro conjI impI*)

**show**  $*$ :  $V * T \frown U * T \longleftrightarrow V * U \frown T * U$   
**proof** (*cases T' = []*)

**assume**  $T': T' = []$   
**show** *?thesis*  
**proof** –

**have**  $V * T \frown U * T \longleftrightarrow V * [t] \frown (u \setminus t) \# (U' * [t \setminus u])$   
**using** *Con-TU Con-sym Resid-rec*(2)  $T T' U U'$  **by** *auto*

**also have**  $\dots \longleftrightarrow (V * [t]) * [u \setminus t] \frown U' * [t \setminus u]$   
**by** (*metis* *Con-TU Con-cons*(2) *Con-rec*(3) *Con-sym Resid.simps*(1)  $T U U'$ )

**also have**  $\dots \longleftrightarrow (V * [u]) * [t \setminus u] \frown U' * [t \setminus u]$   
**using**  $T U V V' R.cube-ax$

**apply** *simp*  
**by** (*metis* *R.con-implies-arr*(1) *R.not-arr-null R.con-def*)

**also have**  $\dots \longleftrightarrow (V * [u]) * U' \frown [t \setminus u] * U'$   
**proof** –

```

    have length [t \ u] + length U' + length (V *\* [u]) ≤ n
      using T U V V' len by force
    thus ?thesis
      by (metis Con-sym Resid.simps(1) add commute ind)
  qed
  also have ... ↔ V *\* U *\* T *\* U
    by (metis Con-TU Resid-cons(2) Resid-rec(3) T T' U U' Con-cons(2)
        length-Resid length-0-conv)
  finally show ?thesis by simp
qed
next
assume T': T' ≠ []
show ?thesis
proof -
  have V *\* T *\* U *\* T ↔ (V *\* [t]) *\* T' *\* ((U *\* [t]) *\* T')
    using Con-TU Con-VT Con-sym Resid-cons(2) T T' by force
  also have ... ↔ (V *\* [t]) *\* (U *\* [t]) *\* T' *\* (U *\* [t])
  proof -
    have length T' + length (U *\* [t]) + length (V *\* [t]) ≤ n
      by (metis (no-types, lifting) Con-TU Con-VT Con-initial-right Con-sym
          One-nat-def Suc-eq-plus1 T ab-semigroup-add-class.add-ac(1)
          add-le-imp-le-left len length-Resid list.size(4) plus-1-eq-Suc)
    thus ?thesis
      by (metis Con-TU Con-VT Con-cons(1) Con-cons(2) T T' U V ind list.discI)
  qed
  also have ... ↔ (V *\* U) *\* ([t] *\* U) *\* T' *\* (U *\* [t])
  proof -
    have length [t] + length U + length V ≤ n
      using T T' le-Suc-eq len by fastforce
    thus ?thesis
      by (metis Con-TU Con-VT Con-initial-left Con-initial-right T ind)
  qed
  also have ... ↔ V *\* U *\* T *\* U
    by (metis Con-cons(2) Con-sym Resid.simps(1) Resid1x-as-Resid
        Resid1-as-Resid Resid-cons' T T')
  finally show ?thesis by blast
qed
qed
show V *\* T *\* U *\* T ⇒
  (V *\* T) *\* (U *\* T) = (V *\* U) *\* (T *\* U)
proof -
  assume Con: V *\* T *\* U *\* T
  show ?thesis
  proof (cases T' = [])
    assume T': T' = []
    show ?thesis
    proof -
      have 1: (V *\* T) *\* (U *\* T) =
        (V *\* T) *\* ((u \ t) # (U *\* [t \ u]))

```

**using** *Con-TU Con-sym Resid-rec(2) T T' U U'* **by force**  
**also have** ... =  $((V * \setminus [t]) * \setminus [u \setminus t]) * \setminus (U' * \setminus [t \setminus u])$   
**by** (*metis Con Con-TU Con-rec(2) Con-sym Resid-cons(2) T T' U U'*  
*calculation*)  
**also have** ... =  $((V * \setminus [u]) * \setminus [t \setminus u]) * \setminus (U' * \setminus [t \setminus u])$   
**by** (*metis \* Con Con-rec(3) R.cube Resid.simps(1,3) T T' U V V'*  
*calculation R.conI R.conE*)  
**also have** ... =  $((V * \setminus [u]) * \setminus U') * \setminus ([t \setminus u] * \setminus U')$   
**proof** –  
**have**  $\text{length } [t \setminus u] + \text{length } (U' * \setminus [t \setminus u]) + \text{length } (V * \setminus [u]) \leq n$   
**by** (*metis (no-types, lifting) Nat.le-diff-conv2 One-nat-def T U V V'*  
*add.commute add-diff-cancel-left' add-leD2 len length-Cons*  
*length-Resid list.size(3) plus-1-eq-Suc*)  
**thus** *?thesis*  
**by** (*metis Con-sym add.commute Resid.simps(1) ind length-Resid*)  
**qed**  
**also have** ... =  $(V * \setminus U) * \setminus (T * \setminus U)$   
**by** (*metis Con-TU Con-cons(2) Resid-cons(2) T T' U U'*  
*Resid-rec(3) length-0-conv length-Resid*)  
**finally show** *?thesis by blast*  
**qed**  
**next**  
**assume**  $T': T' \neq []$   
**show** *?thesis*  
**proof** –  
**have**  $(V * \setminus T) * \setminus (U * \setminus T) =$   
 $((V * \setminus T) * \setminus ([u] * \setminus T)) * \setminus (U' * \setminus (T * \setminus [u]))$   
**by** (*metis Con Con-TU Resid.simps(2) Resid1x-as-Resid U U'*  
*Con-cons(2) Con-sym Resid-cons' Resid-cons(2)*)  
**also have** ... =  $((V * \setminus [u]) * \setminus (T * \setminus [u])) * \setminus (U' * \setminus (T * \setminus [u]))$   
**proof** –  
**have**  $\text{length } T + \text{length } [u] + \text{length } V \leq n$   
**using**  $U U'$  *antisym-conv len not-less-eq-eq* **by fastforce**  
**thus** *?thesis*  
**by** (*metis Con-TU Con-VT Con-initial-right U ind*)  
**qed**  
**also have** ... =  $((V * \setminus [u]) * \setminus U') * \setminus ((T * \setminus [u]) * \setminus U')$   
**proof** –  
**have**  $\text{length } (T * \setminus [u]) + \text{length } U' + \text{length } (V * \setminus [u]) \leq n$   
**using** *Con-TU Con-initial-right U V V' len length-Resid* **by force**  
**thus** *?thesis*  
**by** (*metis Con Con-TU Con-cons(2) U U' calculation ind length-0-conv*  
*length-Resid*)  
**qed**  
**also have** ... =  $(V * \setminus U) * \setminus (T * \setminus U)$   
**by** (*metis \* Con Con-TU Resid-cons(2) U U' length-Resid length-0-conv*)  
**finally show** *?thesis by blast*  
**qed**  
**qed**

```

qed
qed
next
assume V': V' ≠ []
show ?thesis
proof (cases U' = [])
  assume U': U' = []
  show ?thesis
  proof (cases T' = [])
    assume T': T' = []
    show ?thesis
    proof (intro conjI impI)
      show *: V * \ * T * ^ * U * \ * T ↔ V * \ * U * ^ * T * \ * U
      proof -
        have V * \ * T * ^ * U * \ * T ↔ (v \ t) # (V' * \ * [t \ v]) * ^ * [u \ t]
          using Con-TU Con-VT Con-sym Resid-rec(1-2) T T' U U' V V'
          by metis
        also have ... ↔ [v \ t] * ^ * [u \ t] ∧
          V' * \ * [t \ v] * ^ * [u \ v] * \ * [t \ v]
        proof -
          have V' * ^ * [t \ v]
            using T T' V V' Con-VT Con-rec(2) by blast
          thus ?thesis
            using R.con-def R.con-sym R.cube
              Con-rec(2) [of V' * \ * [t \ v] v \ t u \ t]
            by auto
        qed
        also have ... ↔ [v \ t] * ^ * [u \ t] ∧
          V' * \ * [u \ v] * ^ * [t \ v] * \ * [u \ v]
        proof -
          have length [t \ v] + length [u \ v] + length V' ≤ n
            using T U V len by fastforce
          thus ?thesis
            by (metis Con-imp-Arr-Resid Arr-has-Src Con-VT T T' Trgs.simps(1)
              Trgs-Resid-sym V V' Con-rec(2) Srcs-Resid ind)
        qed
        also have ... ↔ [v \ t] * ^ * [u \ t] ∧
          V' * \ * [u \ v] * ^ * [t \ u] * \ * [v \ u]
          by (simp add: R.con-def R.cube)
        also have ... ↔ V * \ * U * ^ * T * \ * U
        proof
          assume 1: V * \ * U * ^ * T * \ * U
          have tu-vu: t \ u ^ v \ u
            by (metis (no-types, lifting) 1 T T' U U' V V' Con-rec(3)
              Resid-rec(1-2) Con-sym length-Resid length-0-conv)
          have vt-ut: v \ t ^ u \ t
            using 1
            by (metis R.con-def R.con-sym R.cube tu-vu)
          show [v \ t] * ^ * [u \ t] ∧ V' * \ * [u \ v] * ^ * [t \ u] * \ * [v \ u]

```

by (*metis* (*no-types, lifting*) 1 *Con-TU Con-cons*(1) *Con-rec*(1-2)  
*Resid-rec*(1) *T T' U U' V V' Resid-rec*(2) *length-Resid*  
*length-0-conv vt-ut*)

**next**

**assume** 1:  $[v \setminus t] \frown [u \setminus t] \wedge$   
 $V' \frown [u \setminus v] \frown [t \setminus u] \frown [v \setminus u]$

**have** *tu-vu*:  $t \setminus u \frown v \setminus u \wedge v \setminus t \frown u \setminus t$

by (*metis* 1 *Con-sym Resid.simps*(1) *Residx1.simps*(2)  
*Residx1-as-Resid*)

**have** *tu*:  $t \frown u$

**using** *Con-TU Con-rec*(1) *T T' U U'* **by** *blast*

**show**  $V \frown U \frown T \frown U$

by (*metis* (*no-types, opaque-lifting*) 1 *Con-rec*(2) *Con-sym*  
*R.con-implies-arr*(2) *Resid.simps*(1,3) *T T' U U' V V'*  
*Resid-rec*(2) *R.arr-resid-iff-con*)

**qed**

**finally show** *?thesis* **by** *simp*

**qed**

**show**  $V \frown T \frown U \frown T \implies$   
 $(V \frown T) \frown (U \frown T) = (V \frown U) \frown (T \frown U)$

**proof** –

**assume** *Con*:  $V \frown T \frown U \frown T$

**have**  $(V \frown T) \frown (U \frown T) = ((v \setminus t) \# (V' \frown [t \setminus v])) \frown [u \setminus t]$

**using** *Con-TU Con-VT Con-sym Resid-rec*(1-2) *T T' U U' V V'* **by** *metis*

**also have** 1:  $\dots = ((v \setminus t) \setminus (u \setminus t)) \#$   
 $(V' \frown [t \setminus v]) \frown ([u \setminus v] \frown [t \setminus v])$

**apply** *simp*

**by** (*metis* *Con Con-VT Con-rec*(2) *R.conE R.conI R.con-sym R.cube*  
*Resid-rec*(2) *T T' V V' calculation*(1))

**also have**  $\dots = ((v \setminus t) \setminus (u \setminus t)) \#$   
 $(V' \frown [u \setminus v]) \frown ([t \setminus v] \frown [u \setminus v])$

**proof** –

**have** *length*  $[t \setminus v] + \text{length } [u \setminus v] + \text{length } V' \leq n$

**using** *T U V len* **by** *fastforce*

**moreover have**  $u \setminus v \frown t \setminus v$

**by** (*metis* 1 *Con-VT Con-rec*(2) *R.con-sym-ax T T' V V' list.discI*  
*R.conE R.conI R.cube*)

**moreover have**  $t \setminus v \frown u \setminus v$

**using** *R.con-sym calculation*(2) **by** *blast*

**ultimately show** *?thesis*

**by** (*metis* *Con-VT Con-rec*(2) *T T' V V' Con-rec*(1) *ind*)

**qed**

**also have**  $\dots = ((v \setminus t) \setminus (u \setminus t)) \#$   
 $((V' \frown [u \setminus v]) \frown ([t \setminus u] \frown [v \setminus u]))$

**using** *R.cube* **by** *fastforce*

**also have**  $\dots = ((v \setminus u) \setminus (t \setminus u)) \#$   
 $((V' \frown [u \setminus v]) \frown ([t \setminus u] \frown [v \setminus u]))$

**by** (*metis* *R.cube*)

**also have**  $\dots = (V \frown U) \frown (T \frown U)$

```

proof –
  have  $(V * \setminus U) * \setminus (T * \setminus U) = ((v \setminus u) \# ((V' * \setminus [u \setminus v]))) * \setminus [t \setminus u]$ 
    using  $T T' U U' V$  Resid-cons(1) [of  $[u] v V$ ]
    by (metis * Con Con-TU Resid.simps(1) Resid-rec(1) Resid-rec(2))
  also have ... =  $((v \setminus u) \setminus (t \setminus u)) \#$ 
     $((V' * \setminus [u \setminus v]) * \setminus ([t \setminus u] * \setminus [v \setminus u]))$ 
    by (metis * Con Con-initial-left calculation Con-sym Resid.simps(1)
      Resid-rec(1-2))
  finally show ?thesis by simp
qed
finally show ?thesis by simp
qed
qed
next
assume  $T': T' \neq []$ 
show ?thesis
proof (intro conjI impI)
  show *:  $V * \setminus T * \frown U * \setminus T \longleftrightarrow V * \setminus U * \frown T * \setminus U$ 
proof –
  have  $V * \setminus T * \frown U * \setminus T \longleftrightarrow (V * \setminus [t]) * \setminus T' * \frown [u \setminus t] * \setminus T'$ 
    using Con-TU Con-VT Con-sym Resid-cons(2) Resid-rec(3)  $T T' U U'$ 
    by force
  also have ...  $\longleftrightarrow (V * \setminus [t]) * \setminus [u \setminus t] * \frown T' * \setminus [u \setminus t]$ 
proof –
  have  $\text{length } [u \setminus t] + \text{length } T' + \text{length } (V * \setminus [t]) \leq n$ 
    using Con-VT Con-initial-right  $T U$  length-Resid len by fastforce
  thus ?thesis
  by (metis Con-TU Con-VT Con-rec(2)  $T T' U V$  add.commute Con-cons(2)
    ind list.discI)
qed
also have ...  $\longleftrightarrow (V * \setminus [u]) * \setminus [t \setminus u] * \frown T' * \setminus [u \setminus t]$ 
proof –
  have  $\text{length } [t] + \text{length } [u] + \text{length } V \leq n$ 
    using  $T T' U$  le-Suc-eq len by fastforce
  hence  $(V * \setminus [t]) * \setminus ([u] * \setminus [t]) = (V * \setminus [u]) * \setminus ([t] * \setminus [u])$ 
    using ind [of  $[t] [u] V$ ]
  by (metis Con-TU Con-VT Con-initial-left Con-initial-right  $T U$ )
  thus ?thesis
  by (metis (full-types) Con-TU Con-initial-left Con-sym Resid-rec(1)  $T U$ )
qed
also have ...  $\longleftrightarrow V * \setminus U * \frown T * \setminus U$ 
  by (metis Con-TU Con-cons(2) Con-rec(2) Resid.simps(1) Resid-rec(2)
     $T T' U U'$ )
  finally show ?thesis by simp
qed
show  $V * \setminus T * \frown U * \setminus T \implies$ 
   $(V * \setminus T) * \setminus (U * \setminus T) = (V * \setminus U) * \setminus (T * \setminus U)$ 
proof –
  assume Con:  $V * \setminus T * \frown U * \setminus T$ 

```

**have**  $(V * \setminus T) * \setminus (U * \setminus T) = ((V * \setminus [t]) * \setminus T') * \setminus ([u \setminus t] * \setminus T')$   
**using** *Con-TU Con-VT Con-sym Resid-cons(2) Resid-rec(3) T T' U U'*  
**by force**  
**also have**  $\dots = ((V * \setminus [t]) * \setminus [u \setminus t]) * \setminus (T' * \setminus [u \setminus t])$   
**proof** –  
**have**  $\text{length } [u \setminus t] + \text{length } T' + \text{length } (\text{Resid } V [t]) \leq n$   
**using** *Con-VT Con-initial-right T U length-Resid len* **by fastforce**  
**thus** *?thesis*  
**by** (*metis Con-TU Con-VT Con-cons(2) Con-rec(2) T T' U V add.commute ind list.discI*)  
**qed**  
**also have**  $\dots = ((V * \setminus [u]) * \setminus [t \setminus u]) * \setminus (T' * \setminus [u \setminus t])$   
**proof** –  
**have**  $\text{length } [t] + \text{length } [u] + \text{length } V \leq n$   
**using** *T T' U le-Suc-eq len* **by fastforce**  
**thus** *?thesis*  
**using** *ind [of [t] [u] V]*  
**by** (*metis Con-TU Con-VT Con-initial-left Con-sym Resid-rec(1) T U*)  
**qed**  
**also have**  $\dots = (V * \setminus U) * \setminus (T * \setminus U)$   
**using** *\* Con Con-TU Con-rec(2) Resid-cons(2) Resid-rec(2) T T' U U'*  
**by auto**  
**finally show** *?thesis by simp*  
**qed**  
**qed**  
**qed**  
**next**  
**assume**  $U': U' \neq []$   
**show** *?thesis*  
**proof** (*cases T' = []*)  
**assume**  $T': T' = []$   
**show** *?thesis*  
**proof** (*intro conjI impI*)  
**show**  $*$ :  $V * \setminus T * \frown U * \setminus T \longleftrightarrow V * \setminus U * \frown T * \setminus U$   
**proof** –  
**have**  $V * \setminus T * \frown U * \setminus T \longleftrightarrow V * \setminus [t] * \frown (u \setminus t) \# (U' * \setminus [t \setminus u])$   
**using** *T U V T' U' V' Con-TU Con-VT Con-sym Resid-rec(2)* **by auto**  
**also have**  $\dots \longleftrightarrow V * \setminus [t] * \frown [u \setminus t] \wedge$   
 $(V * \setminus [t]) * \setminus [u \setminus t] * \frown U' * \setminus [t \setminus u]$   
**by** (*metis Con-TU Con-VT Con-cons(2) Con-initial-right Con-rec(2) Con-sym T U U'*)  
**also have**  $\dots \longleftrightarrow V * \setminus [t] * \frown [u \setminus t] \wedge$   
 $(V * \setminus [u]) * \setminus [t \setminus u] * \frown U' * \setminus [t \setminus u]$   
**proof** –  
**have**  $\text{length } [u] + \text{length } [t] + \text{length } V \leq n$   
**using** *T U V T' U' V' len not-less-eq-eq order-trans* **by fastforce**  
**thus** *?thesis*  
**using** *ind [of [t] [u] V]*  
**by** (*metis Con-TU Con-VT Con-initial-right Resid-rec(1) T U*)

*Con-sym length-Cons*)

**qed**

**also have** ...  $\longleftrightarrow V \text{ * } \setminus \text{ * } [u] \text{ * } \frown \text{ * } [t \setminus u] \wedge$   
 $(V \text{ * } \setminus \text{ * } [u]) \text{ * } \setminus \text{ * } [t \setminus u] \text{ * } \frown \text{ * } U' \text{ * } \setminus \text{ * } [t \setminus u]$

**proof** –

**have**  $\text{length } [t] + \text{length } [u] + \text{length } V \leq n$   
**using**  $T \ U \ V \ T' \ U' \ V' \ \text{len} \ \text{antisym-conv} \ \text{not-less-eq-eq}$  **by** *fastforce*  
**thus** *?thesis*  
**using** *ind [of [t]]*  
**by** (*metis (full-types) Con-TU Con-VT Con-initial-right Con-sym*  
*Resid-rec(1) T U*)

**qed**

**also have** ...  $\longleftrightarrow (V \text{ * } \setminus \text{ * } [u]) \text{ * } \setminus \text{ * } U' \text{ * } \frown \text{ * } [t \setminus u] \text{ * } \setminus \text{ * } U'$

**proof** –

**have**  $\text{length } [t \setminus u] + \text{length } U' + \text{length } (V \text{ * } \setminus \text{ * } [u]) \leq n$   
**by** (*metis T T' U add.assoc add.right-neutral add-leD1*  
*add-le-cancel-left length-Resid len length-Cons list.size(3)*  
*plus-1-eq-Suc*)  
**thus** *?thesis*  
**by** (*metis (no-types, opaque-lifting) Con-sym Resid.simps(1)*  
*add commute ind*)

**qed**

**also have** ...  $\longleftrightarrow V \text{ * } \setminus \text{ * } U \text{ * } \frown \text{ * } T \text{ * } \setminus \text{ * } U$   
**by** (*metis Con-TU Resid-cons(2) Resid-rec(3) T T' U U'*  
*Con-cons(2) length-Resid length-0-conv*)

**finally show** *?thesis* **by** *blast*

**qed**

**show**  $V \text{ * } \setminus \text{ * } T \text{ * } \frown \text{ * } U \text{ * } \setminus \text{ * } T \implies$   
 $(V \text{ * } \setminus \text{ * } T) \text{ * } \setminus \text{ * } (U \text{ * } \setminus \text{ * } T) = (V \text{ * } \setminus \text{ * } U) \text{ * } \setminus \text{ * } (T \text{ * } \setminus \text{ * } U)$

**proof** –

**assume** *Con: V \* \setminus \* T \* \frown \* U \* \setminus \* T*  
**have**  $(V \text{ * } \setminus \text{ * } T) \text{ * } \setminus \text{ * } (U \text{ * } \setminus \text{ * } T) =$   
 $(V \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } ((u \setminus t) \# (U' \text{ * } \setminus \text{ * } [t \setminus u]))$   
**using** *Con-TU Con-sym Resid-rec(2) T T' U U'* **by** *auto*

**also have** ... =  $((V \text{ * } \setminus \text{ * } [t]) \text{ * } \setminus \text{ * } [u \setminus t]) \text{ * } \setminus \text{ * } (U' \text{ * } \setminus \text{ * } [t \setminus u])$   
**by** (*metis Con Con-TU Con-rec(2) Con-sym T T' U U' calculation*  
*Resid-cons(2)*)

**also have** ... =  $((V \text{ * } \setminus \text{ * } [u]) \text{ * } \setminus \text{ * } [t \setminus u]) \text{ * } \setminus \text{ * } (U' \text{ * } \setminus \text{ * } [t \setminus u])$

**proof** –

**have**  $\text{length } [t] + \text{length } [u] + \text{length } V \leq n$   
**using**  $T \ U \ U' \ \text{le-Suc-eq} \ \text{len}$  **by** *fastforce*  
**thus** *?thesis*  
**using**  $T \ U \ \text{Con-TU} \ \text{Con-VT} \ \text{Con-sym} \ \text{ind} \ [\text{of } [t] \ [u] \ V]$   
**by** (*metis (no-types, opaque-lifting) Con-initial-right Resid.simps(3)*)

**qed**

**also have** ... =  $((V \text{ * } \setminus \text{ * } [u]) \text{ * } \setminus \text{ * } U') \text{ * } \setminus \text{ * } ([t \setminus u] \text{ * } \setminus \text{ * } U')$

**proof** –

**have**  $\text{length } [t \setminus u] + \text{length } U' + \text{length } (V \text{ * } \setminus \text{ * } [u]) \leq n$   
**by** (*metis (no-types, opaque-lifting) T T' U add.left-commute*)

*add.right-neutral add-leD2 add-le-cancel-left len length-Cons  
length-Resid list.size(3) plus-1-eq-Suc*

**thus** *?thesis*  
**by** (*metis Con Con-TU Con-rec(3) T T' U U' calculation  
ind length-0-conv length-Resid*)

**qed**  
**also have**  $\dots = (V \text{ * } \backslash \text{ * } U) \text{ * } \backslash \text{ * } (T \text{ * } \backslash \text{ * } U)$   
**by** (*metis \* Con Con-TU Resid-rec(3) T T' U U' Resid-cons(2)  
length-Resid length-0-conv*)

**finally show** *?thesis by blast*  
**qed**  
**qed**  
**next**  
**assume**  $T': T' \neq []$   
**show** *?thesis*  
**proof** (*intro conjI impI*)  
**have** 1:  $U \text{ * } \frown \text{ * } [t]$   
**using** *T Con-TU*  
**by** (*metis Con-cons(2) Con-sym Resid.simps(2)*)  
**have** 2:  $V \text{ * } \frown \text{ * } [t]$   
**using** *V Con-VT Con-initial-right T by blast*  
**have** 3:  $\text{length } T' + \text{length } (U \text{ * } \backslash \text{ * } [t]) + \text{length } (V \text{ * } \backslash \text{ * } [t]) \leq n$   
**using** 1 2 *T len length-Resid by force*  
**have** 4:  $\text{length } [t] + \text{length } U + \text{length } V \leq n$   
**using** *T T' len antisym-conv not-less-eq-eq by fastforce*  
**show** \*:  $V \text{ * } \backslash \text{ * } T \text{ * } \frown \text{ * } U \text{ * } \backslash \text{ * } T \longleftrightarrow V \text{ * } \backslash \text{ * } U \text{ * } \frown \text{ * } T \text{ * } \backslash \text{ * } U$   
**proof** –  
**have**  $V \text{ * } \backslash \text{ * } T \text{ * } \frown \text{ * } U \text{ * } \backslash \text{ * } T \longleftrightarrow (V \text{ * } \backslash \text{ * } [t]) \text{ * } \backslash \text{ * } T' \text{ * } \frown \text{ * } (U \text{ * } \backslash \text{ * } [t]) \text{ * } \backslash \text{ * } T'$   
**using** *Con-TU Con-VT Con-sym Resid-cons(2) T T' by force*  
**also have**  $\dots \longleftrightarrow (V \text{ * } \backslash \text{ * } [t]) \text{ * } \backslash \text{ * } (U \text{ * } \backslash \text{ * } [t]) \text{ * } \frown \text{ * } T' \text{ * } \backslash \text{ * } (U \text{ * } \backslash \text{ * } [t])$   
**by** (*metis 3 Con-TU Con-VT Con-cons(1) Con-cons(2) T T' U V ind  
list.discI*)  
**also have**  $\dots \longleftrightarrow (V \text{ * } \backslash \text{ * } U) \text{ * } \backslash \text{ * } ([t] \text{ * } \backslash \text{ * } U) \text{ * } \frown \text{ * } T' \text{ * } \backslash \text{ * } (U \text{ * } \backslash \text{ * } [t])$   
**by** (*metis 1 2 4 Con-sym ind*)  
**also have**  $\dots \longleftrightarrow V \text{ * } \backslash \text{ * } U \text{ * } \frown \text{ * } \text{hd } ([t] \text{ * } \backslash \text{ * } U) \# T' \text{ * } \backslash \text{ * } (U \text{ * } \backslash \text{ * } [t])$   
**by** (*metis 1 Con-TU Con-cons(1) Con-cons(2) Resid.simps(1)  
Resid1x-as-Resid T T' list.sel(1)*)  
**also have**  $\dots \longleftrightarrow V \text{ * } \backslash \text{ * } U \text{ * } \frown \text{ * } T \text{ * } \backslash \text{ * } U$   
**using** 1 *Resid-cons' [of T' t U] Con-TU T T' Resid1x-as-Resid  
Con-sym*  
**by force**  
**finally show** *?thesis by simp*  
**qed**  
**show**  $(V \text{ * } \backslash \text{ * } T) \text{ * } \backslash \text{ * } (U \text{ * } \backslash \text{ * } T) = (V \text{ * } \backslash \text{ * } U) \text{ * } \backslash \text{ * } (T \text{ * } \backslash \text{ * } U)$   
**proof** –  
**have**  $(V \text{ * } \backslash \text{ * } T) \text{ * } \backslash \text{ * } (U \text{ * } \backslash \text{ * } T) =$   
 $((V \text{ * } \backslash \text{ * } [t]) \text{ * } \backslash \text{ * } T') \text{ * } \backslash \text{ * } ((U \text{ * } \backslash \text{ * } [t]) \text{ * } \backslash \text{ * } T')$   
**using** *Con-TU Con-VT Con-sym Resid-cons(2) T T' by force*  
**also have**  $\dots = ((V \text{ * } \backslash \text{ * } [t]) \text{ * } \backslash \text{ * } (U \text{ * } \backslash \text{ * } [t])) \text{ * } \backslash \text{ * } (T' \text{ * } \backslash \text{ * } (U \text{ * } \backslash \text{ * } [t]))$

by (*metis* (*no-types*, *lifting*) 3 *Con-TU* *Con-VT*  $T$   $T'$   $U$   $V$  *Con-cons*(1)  
*Con-cons*(2) *ind* *list.simps*(3))  
 also have ... =  $((V * \setminus U) * \setminus ([t] * \setminus U)) * \setminus (T' * \setminus (U * \setminus [t]))$   
 by (*metis* 1 2 4 *Con-sym* *ind*)  
 also have ... =  $(V * \setminus U) * \setminus ((t \# T') * \setminus U)$   
 by (*metis* \* *Con-TU* *Con-cons*(1) *Resid1x-as-Resid*  
*Resid-cons'*  $T$   $T'$   $U$  *calculation* *Resid-cons*(2) *list.distinct*(1))  
 also have ... =  $(V * \setminus U) * \setminus (T * \setminus U)$   
 using  $T$  by *fastforce*  
 finally show ?*thesis* by *simp*  
 qed  
 qed  
 qed  
 qed  
 qed  
 qed  
 qed  
 qed  
 qed  
 qed

**lemma** *Cube*:

shows  $T * \setminus U * \frown V * \setminus U \longleftrightarrow T * \setminus V * \frown U * \setminus V$   
 and  $T * \setminus U * \frown V * \setminus U \implies (T * \setminus U) * \setminus (V * \setminus U) = (T * \setminus V) * \setminus (U * \setminus V)$   
 proof –  
 show  $T * \setminus U * \frown V * \setminus U \longleftrightarrow T * \setminus V * \frown U * \setminus V$   
 using *Cube-ind* by (*metis* *Con-sym* *Resid.simps*(1) *le-add2*)  
 show  $T * \setminus U * \frown V * \setminus U \implies (T * \setminus U) * \setminus (V * \setminus U) = (T * \setminus V) * \setminus (U * \setminus V)$   
 using *Cube-ind* by (*metis* *Con-sym* *Resid.simps*(1) *order-refl*)  
 qed

**lemma** *Con-implies-Arr*:

assumes  $T * \frown U$   
 shows *Arr*  $T$  and *Arr*  $U$   
 using *assms* *Con-sym*  
 by (*metis* *Con-imp-Arr-Resid* *Arr-iff-Con-self* *Cube*(1) *Resid.simps*(1))+

**sublocale** *partial-magma* *Resid*

by (*unfold-locales*, *metis* *Resid.simps*(1) *Con-sym*)

**lemma** *is-partial-magma*:

shows *partial-magma* *Resid*

..

**lemma** *null-char*:

shows *null* = []  
 by (*metis* *null-is-zero*(2) *Resid.simps*(1))

**sublocale** *residuation* *Resid*

using *null-char* *Con-sym* *Arr-iff-Con-self* *Con-imp-Arr-Resid* *Cube* *null-is-zero*(2)

by *unfold-locales auto*

**lemma** *is-residuation*:  
**shows** *residuation Resid*  
 ..

**lemma** *arr-char*:  
**shows**  $arr\ T \longleftrightarrow Arr\ T$   
 using *null-char Arr-iff-Con-self* **by** *fastforce*

**lemma** *arrI<sub>P</sub>* [*intro*]:  
**assumes** *Arr T*  
**shows** *arr T*  
 using *assms arr-char* **by** *auto*

**lemma** *ide-char*:  
**shows**  $ide\ T \longleftrightarrow Ide\ T$   
 by (*metis Con-Arr-self Ide-implies-Arr Resid-Arr-Ide-ind Resid-Arr-self arr-char ide-def arr-def*)

**lemma** *con-char*:  
**shows**  $con\ T\ U \longleftrightarrow Con\ T\ U$   
 using *null-char* **by** *auto*

**lemma** *conI<sub>P</sub>* [*intro*]:  
**assumes** *Con T U*  
**shows** *con T U*  
 using *assms con-char* **by** *auto*

**sublocale** *rts Resid*  
**proof**  
 show  $\bigwedge A\ T. \llbracket ide\ A; con\ T\ A \rrbracket \implies T\ *\ \backslash\ * \ A = T$   
 using *Resid-Arr-Ide-ind ide-char null-char* **by** *auto*  
 show  $\bigwedge T. arr\ T \implies ide\ (trg\ T)$   
 by (*metis arr-char Resid-Arr-self ide-char resid-arr-self*)  
 show  $\bigwedge A\ T. \llbracket ide\ A; con\ A\ T \rrbracket \implies ide\ (A\ *\ \backslash\ * \ T)$   
 by (*simp add: Resid-Ide-Arr-ind con-char ide-char*)  
 show  $\bigwedge T\ U. con\ T\ U \implies \exists A. ide\ A \wedge con\ A\ T \wedge con\ A\ U$   
**proof** –  
 fix *T U*  
 assume *TU: con T U*  
 have *1: Srcs T = Srcs U*  
 using *TU Con-imp-eq-Srcs con-char* **by** *force*  
 obtain *a* **where** *a: a ∈ Srcs T ∩ Srcs U*  
 using *1*  
 by (*metis Int-absorb Int-emptyI TU arr-char Arr-has-Src con-implies-arr(1)*)  
 show  $\exists A. ide\ A \wedge con\ A\ T \wedge con\ A\ U$   
 using *a 1*  
 by (*metis (full-types) Ball-Collect Con-single-ide-ind Ide.simps(2) Int-absorb TU*)

*Srcs-are-ide arr-char con-char con-implies-arr(1-2) ide-char*)

**qed**  
**show**  $\bigwedge T U V. \llbracket \text{ide } (Resid T U); \text{con } U V \rrbracket \implies \text{con } (T \text{ }^* \backslash^* U) (V \text{ }^* \backslash^* U)$   
**using** *null-char ide-char*  
**by** (*metis Con-imp-Arr-Resid Con-Ide-iff Srcs-Resid con-char con-sym arr-resid-iff-con ide-implies-arr*)

**qed**

**theorem** *is-rts*:  
**shows** *rts Resid*  
**..**

**notation** *cong* (**infix**  $\langle^* \sim^* \rangle$  50)  
**notation** *prfx* (**infix**  $\langle^* \lesssim^* \rangle$  50)

**lemma** *sources-char<sub>P</sub>*:  
**shows**  $\text{sources } T = \{A. \text{Ide } A \wedge \text{Arr } T \wedge \text{Srcs } A = \text{Srcs } T\}$   
**using** *Con-Ide-iff Con-sym con-char ide-char sources-def* **by** *fastforce*

**lemma** *sources-cons*:  
**shows**  $\text{Arr } (t \# T) \implies \text{sources } (t \# T) = \text{sources } [t]$   
**apply** (*induct T*)  
**apply** *simp*  
**using** *sources-char<sub>P</sub>* **by** *auto*

**lemma** *targets-char<sub>P</sub>*:  
**shows**  $\text{targets } T = \{B. \text{Ide } B \wedge \text{Arr } T \wedge \text{Srcs } B = \text{Trgs } T\}$   
**unfolding** *targets-def*  
**by** (*metis (no-types, lifting) trg-def Arr.simps(1) Ide-implies-Arr Resid-Arr-self arr-char Con-Ide-iff Srcs-Resid con-char ide-char con-implies-arr(1)*)

**lemma** *seq-char'*:  
**shows**  $\text{seq } T U \iff \text{Arr } T \wedge \text{Arr } U \wedge \text{Trgs } T \cap \text{Srcs } U \neq \{\}$   
**proof**  
**show**  $\text{seq } T U \implies \text{Arr } T \wedge \text{Arr } U \wedge \text{Trgs } T \cap \text{Srcs } U \neq \{\}$   
**unfolding** *seq-def*  
**using** *Arr-has-Trg arr-char Con-Arr-self sources-char<sub>P</sub> trg-def trg-in-targets*  
**by** *fastforce*  
**assume** *1: Arr T ∧ Arr U ∧ Trgs T ∩ Srcs U ≠ {}*  
**have** *targets T = sources U*  
**proof** –  
**obtain** *a* **where** *a: R.ide a ∧ a ∈ Trgs T ∧ a ∈ Srcs U*  
**using** *1 Trgs-are-ide* **by** *blast*  
**have**  $\text{Trgs } [a] = \text{Trgs } T$   
**using** *a 1*  
**by** (*metis Con-single-ide-ind Con-sym Resid-Arr-Src Srcs-Resid Trgs-eqI*)  
**moreover** **have**  $\text{Srcs } [a] = \text{Srcs } U$   
**using** *a 1 Con-single-ide-ind Con-imp-eq-Srcs* **by** *blast*  
**moreover** **have**  $\text{Trgs } [a] = \text{Srcs } [a]$

**using**  $a$   
**by** ( $metis$   $R.sources-resid$   $Srcs.simps(2)$   $Trgs.simps(2)$   $R.ideE$ )  
**ultimately show**  $?thesis$   
**using**  $1$   $sources-char_P$   $targets-char_P$  **by**  $auto$   
**qed**  
**thus**  $seq\ T\ U$   
**using**  $1$  **by**  $blast$   
**qed**

**lemma**  $seq-char$ :  
**shows**  $seq\ T\ U \longleftrightarrow Arr\ T \wedge Arr\ U \wedge Trgs\ T = Srcs\ U$   
**by** ( $metis$   $Int-absorb$   $Srcs-Resid$   $Arr-has-Src$   $Arr-iff-Con-self$   $Srcs-eqI$   $seq-char'$ )

**lemma**  $seqI_P$  [ $intro$ ]:  
**assumes**  $Arr\ T$  **and**  $Arr\ U$  **and**  $Trgs\ T \cap Srcs\ U \neq \{\}$   
**shows**  $seq\ T\ U$   
**using**  $assms\ seq-char'$  **by**  $auto$

**lemma**  $coinitial-char$ :  
**shows**  $coinitial\ T\ U \implies Arr\ T \wedge Arr\ U \wedge Srcs\ T = Srcs\ U$   
**and**  $Arr\ T \wedge Arr\ U \wedge Srcs\ T \cap Srcs\ U \neq \{\} \implies coinitial\ T\ U$   
**proof** –  
**show**  $coinitial\ T\ U \implies Arr\ T \wedge Arr\ U \wedge Srcs\ T = Srcs\ U$   
**unfolding**  $seq-def$   
**by** ( $metis$   $Con-imp-eq-Srcs$   $arr-char$   $coinitial-iff$   
 $con-char\ prfx-implies-con$   $source-is-prfx$   $src-in-sources$ )  
**assume**  $1$ :  $Arr\ T \wedge Arr\ U \wedge Srcs\ T \cap Srcs\ U \neq \{\}$   
**have**  $sources\ T = sources\ U$   
**proof** –  
**obtain**  $a$  **where**  $a$ :  $R.ide\ a \wedge a \in Srcs\ T \wedge a \in Srcs\ U$   
**using**  $1$   $Srcs-are-ide$  **by**  $blast$   
**have**  $Srcs\ [a] = Srcs\ T$   
**using**  $a\ 1$   
**by** ( $metis$   $Arr.simps(1)$   $Con-imp-eq-Srcs$   $Resid-Arr-Src$ )  
**moreover have**  $Srcs\ [a] = Srcs\ U$   
**using**  $a\ 1$   $Con-single-ide-ind$   $Con-imp-eq-Srcs$  **by**  $blast$   
**ultimately show**  $?thesis$   
**using**  $1$   $sources-char_P$   $targets-char_P$  **by**  $auto$   
**qed**  
**thus**  $coinitial\ T\ U$   
**using**  $1$  **by**  $blast$   
**qed**

**lemma**  $coinitialI_P$  [ $intro$ ]:  
**assumes**  $Arr\ T$  **and**  $Arr\ U$  **and**  $Srcs\ T \cap Srcs\ U \neq \{\}$   
**shows**  $coinitial\ T\ U$   
**using**  $assms\ coinitial-char(2)$  **by**  $auto$

**lemma**  $Ide-imp-sources-eq-targets$ :

**assumes** *Ide T*  
**shows** *sources T = targets T*  
**using** *assms*  
**by** (*metis Resid-Arr-Ide-ind arr-iff-has-source arr-iff-has-target con-char*  
*arr-def sources-resid*)

## 2.4.2 Inclusion Map

Inclusion of an RTS to the RTS of its paths.

**abbreviation** *incl*  
**where** *incl*  $\equiv \lambda t. \text{if } R.\text{arr } t \text{ then } [t] \text{ else null}$

**sublocale** *incl: simulation resid Resid incl*  
**using** *R.con-implies-arr(1-2) con-char R.arr-resid-iff-con null-char*  
**by** *unfold-locales auto*

**lemma** *incl-is-simulation:*  
**shows** *simulation resid Resid incl*  
 ..

**lemma** *incl-is-injective:*  
**shows** *inj-on incl (Collect R.arr)*  
**by** (*intro inj-onI*) *simp*

**lemma** *reflects-con:*  
**assumes** *incl t \* $\frown$ \* incl u*  
**shows** *t  $\frown$  u*  
**using** *assms*  
**by** (*metis (full-types) Arr.simps(1) Con-implies-Arr(1-2) Con-rec(1) null-char*)

end

## 2.4.3 Composites of Paths

The RTS of paths has composites, given by the append operation on lists.

**context** *paths-in-rts*  
**begin**

**lemma** *Srcs-append [simp]:*  
**assumes** *T  $\neq$  []*  
**shows** *Srcs (T @ U) = Srcs T*  
**by** (*metis Nil-is-append-conv Srcs.simps(2) Srcs.simps(3) assms hd-append list.exhaust-sel*)

**lemma** *Trgs-append [simp]:*  
**shows** *U  $\neq$  []  $\implies$  Trgs (T @ U) = Trgs U*  
**proof** (*induct T*)  
**show** *U  $\neq$  []  $\implies$  Trgs ([] @ U) = Trgs U*  
**by** *auto*  
**show**  $\bigwedge t T. [U \neq [] \implies \text{Trgs } (T @ U) = \text{Trgs } U; U \neq []]$

$\implies \text{Trgs } ((t \# T) @ U) = \text{Trgs } U$

by (metis Nil-is-append-conv Trgs.simps(3) append-Cons list.exhaust)

qed

**lemma** seq-implies-Trgs-eq-Srcs:  
**shows**  $[[\text{Arr } T; \text{Arr } U; \text{Trgs } T \subseteq \text{Srcs } U]] \implies \text{Trgs } T = \text{Srcs } U$   
 by (metis inf.orderE Arr-has-Trg seqI<sub>P</sub> seq-char)

**lemma** Arr-append-iff<sub>P</sub>:  
**shows**  $[[T \neq []; U \neq []]] \implies \text{Arr } (T @ U) \longleftrightarrow \text{Arr } T \wedge \text{Arr } U \wedge \text{Trgs } T \subseteq \text{Srcs } U$   
**proof** (induct T arbitrary: U)  
**show**  $\bigwedge U. [[] \neq []; U \neq []] \implies \text{Arr } ([] @ U) = (\text{Arr } [] \wedge \text{Arr } U \wedge \text{Trgs } [] \subseteq \text{Srcs } U)$   
 by simp  
**fix** t T **and** U :: 'a list  
**assume** ind:  $\bigwedge U. [[T \neq []; U \neq []]]$   
 $\implies \text{Arr } (T @ U) = (\text{Arr } T \wedge \text{Arr } U \wedge \text{Trgs } T \subseteq \text{Srcs } U)$   
**assume** U: U  $\neq []$   
**show**  $\text{Arr } ((t \# T) @ U) \longleftrightarrow \text{Arr } (t \# T) \wedge \text{Arr } U \wedge \text{Trgs } (t \# T) \subseteq \text{Srcs } U$   
**proof** (cases T = [])  
**show** T = []  $\implies$  ?thesis  
 using Arr.elims(1) U by auto  
**assume** T: T  $\neq []$   
**have**  $\text{Arr } ((t \# T) @ U) \longleftrightarrow \text{Arr } (t \# (T @ U))$   
 by simp  
**also have** ...  $\longleftrightarrow R.\text{arr } t \wedge \text{Arr } (T @ U) \wedge R.\text{targets } t \subseteq \text{Srcs } (T @ U)$   
 using T U  
 by (metis Arr.simps(3) Nil-is-append-conv neq-Nil-conv)  
**also have** ...  $\longleftrightarrow R.\text{arr } t \wedge \text{Arr } T \wedge \text{Arr } U \wedge \text{Trgs } T \subseteq \text{Srcs } U \wedge R.\text{targets } t \subseteq \text{Srcs } T$   
 using T U ind by auto  
**also have** ...  $\longleftrightarrow \text{Arr } (t \# T) \wedge \text{Arr } U \wedge \text{Trgs } (t \# T) \subseteq \text{Srcs } U$   
 using T U  
 by (metis Arr.simps(3) Trgs.simps(3) neq-Nil-conv)  
**finally show** ?thesis by auto

qed

qed

**lemma** Arr-consI<sub>P</sub> [intro, simp]:  
**assumes** R.arr t **and** Arr U **and** R.targets t  $\subseteq$  Srcs U  
**shows** Arr (t # U)  
 using assms Arr.elims(3) by blast

**lemma** Arr-appendI<sub>P</sub> [intro, simp]:  
**assumes** Arr T **and** Arr U **and** Trgs T  $\subseteq$  Srcs U  
**shows** Arr (T @ U)  
 using assms  
 by (metis Arr.simps(1) Arr-append-iff<sub>P</sub>)

**lemma** Arr-appendE<sub>P</sub> [elim]:  
**assumes** Arr (T @ U) **and** T  $\neq []$  **and** U  $\neq []$

and  $\llbracket \text{Arr } T; \text{Arr } U; \text{Trgs } T = \text{Srcs } U \rrbracket \implies \text{thesis}$   
shows *thesis*  
using *assms Arr-append-iff<sub>P</sub> seq-implies-Trgs-eq-Srcs* **by force**

**lemma** *Ide-append-iff<sub>P</sub>*:  
shows  $\llbracket T \neq []; U \neq [] \rrbracket \implies \text{Ide } (T @ U) \longleftrightarrow \text{Ide } T \wedge \text{Ide } U \wedge \text{Trgs } T \subseteq \text{Srcs } U$   
using *Ide-char* **by auto**

**lemma** *Ide-appendI<sub>P</sub>* [*intro, simp*]:  
assumes *Ide T* and *Ide U* and  $\text{Trgs } T \subseteq \text{Srcs } U$   
shows *Ide (T @ U)*  
using *assms*  
**by** (*metis Ide.simps(1) Ide-append-iff<sub>P</sub>*)

**lemma** *Resid-append-ind*:  
shows  $\llbracket T \neq []; U \neq []; V \neq [] \rrbracket \implies$   
 $(V @ T^* \frown^* U \longleftrightarrow V^* \frown^* U \wedge T^* \frown^* U^* \setminus^* V) \wedge$   
 $(T^* \frown^* V @ U \longleftrightarrow T^* \frown^* V \wedge T^* \setminus^* V^* \frown^* U) \wedge$   
 $(V @ T^* \frown^* U \longrightarrow (V @ T)^* \setminus^* U = V^* \setminus^* U @ T^* \setminus^* (U^* \setminus^* V)) \wedge$   
 $(T^* \frown^* V @ U \longrightarrow T^* \setminus^* (V @ U) = (T^* \setminus^* V)^* \setminus^* U)$

**proof** (*induct V arbitrary: T U*)  
show  $\wedge T U. \llbracket T \neq []; U \neq []; [] \neq [] \rrbracket \implies$   
 $([] @ T^* \frown^* U \longleftrightarrow []^* \frown^* U \wedge T^* \frown^* U^* \setminus^* []) \wedge$   
 $(T^* \frown^* [] @ U \longleftrightarrow T^* \frown^* [] \wedge T^* \setminus^* []^* \frown^* U) \wedge$   
 $([] @ T^* \frown^* U \longrightarrow ([] @ T)^* \setminus^* U = []^* \setminus^* U @ T^* \setminus^* (U^* \setminus^* [])) \wedge$   
 $(T^* \frown^* [] @ U \longrightarrow T^* \setminus^* ([] @ U) = (T^* \setminus^* [])^* \setminus^* U)$

**by simp**  
**fix**  $v :: 'a$  and  $T U V :: 'a \text{ list}$

**assume** *ind*:  $\wedge T U. \llbracket T \neq []; U \neq []; V \neq [] \rrbracket \implies$   
 $(V @ T^* \frown^* U \longleftrightarrow V^* \frown^* U \wedge T^* \frown^* U^* \setminus^* V) \wedge$   
 $(T^* \frown^* V @ U \longleftrightarrow T^* \frown^* V \wedge T^* \setminus^* V^* \frown^* U) \wedge$   
 $(V @ T^* \frown^* U \longrightarrow (V @ T)^* \setminus^* U = V^* \setminus^* U @ T^* \setminus^* (U^* \setminus^* V)) \wedge$   
 $(T^* \frown^* V @ U \longrightarrow T^* \setminus^* (V @ U) = (T^* \setminus^* V)^* \setminus^* U)$

**assume**  $T: T \neq []$  and  $U: U \neq []$   
**show**  $((v \# V) @ T^* \frown^* U \longleftrightarrow (v \# V)^* \frown^* U \wedge T^* \frown^* U^* \setminus^* (v \# V)) \wedge$   
 $(T^* \frown^* (v \# V) @ U \longleftrightarrow T^* \frown^* (v \# V) \wedge T^* \setminus^* (v \# V)^* \frown^* U) \wedge$   
 $((v \# V) @ T^* \frown^* U \longrightarrow$   
 $((v \# V) @ T)^* \setminus^* U = (v \# V)^* \setminus^* U @ T^* \setminus^* (U^* \setminus^* (v \# V))) \wedge$   
 $(T^* \frown^* (v \# V) @ U \longrightarrow T^* \setminus^* ((v \# V) @ U) = (T^* \setminus^* (v \# V))^* \setminus^* U)$

**proof** (*intro conjI iffI impI*)  
**show** 1:  $(v \# V) @ T^* \frown^* U \implies$   
 $((v \# V) @ T)^* \setminus^* U = (v \# V)^* \setminus^* U @ T^* \setminus^* (U^* \setminus^* (v \# V))$

**proof** (*cases V = []*)  
**show**  $V = [] \implies (v \# V) @ T^* \frown^* U \implies ?thesis$   
using *T U Resid-cons(1) U* **by auto**

**assume**  $V: V \neq []$   
**assume** *Con*:  $(v \# V) @ T^* \frown^* U$   
**have**  $((v \# V) @ T)^* \setminus^* U = (v \# (V @ T))^* \setminus^* U$   
**by simp**

**also have** ... =  $[v] * \setminus^* U @ (V @ T) * \setminus^* (U * \setminus^* [v])$   
**using**  $T U Con Resid-cons$  **by** *simp*  
**also have** ... =  $[v] * \setminus^* U @ V * \setminus^* (U * \setminus^* [v]) @ T * \setminus^* ((U * \setminus^* [v]) * \setminus^* V)$   
**using**  $T U V Con ind Resid-cons$   
**by** (*metis Con-sym Cons-eq-appendI append-is-Nil-conv Con-cons(1)*)  
**also have** ... =  $(v \# V) * \setminus^* U @ T * \setminus^* (U * \setminus^* (v \# V))$   
**using** *ind[of T]*  
**by** (*metis Con Con-cons(2) Cons-eq-appendI Resid-cons(1) Resid-cons(2) T U V append.assoc append-is-Nil-conv Con-sym*)  
**finally show** *?thesis* **by** *simp*  
**qed**  
**show** 2:  $T * \frown^* (v \# V) @ U \implies T * \setminus^* ((v \# V) @ U) = (T * \setminus^* (v \# V)) * \setminus^* U$   
**proof** (*cases V = []*)  
**show**  $V = [] \implies T * \frown^* (v \# V) @ U \implies ?thesis$   
**using**  $Resid-cons(2) T U$  **by** *auto*  
**assume**  $V: V \neq []$   
**assume**  $Con: T * \frown^* (v \# V) @ U$   
**have**  $T * \setminus^* ((v \# V) @ U) = T * \setminus^* (v \# (V @ U))$   
**by** *simp*  
**also have** 1: ... =  $(T * \setminus^* [v]) * \setminus^* (V @ U)$   
**using**  $V Con Resid-cons(2) T$  **by** *force*  
**also have** ... =  $((T * \setminus^* [v]) * \setminus^* V) * \setminus^* U$   
**using**  $T U V 1 Con ind$   
**by** (*metis Con-initial-right Cons-eq-appendI*)  
**also have** ... =  $(T * \setminus^* (v \# V)) * \setminus^* U$   
**using**  $T V Con$   
**by** (*metis Con-cons(2) Con-initial-right Cons-eq-appendI Resid-cons(2)*)  
**finally show** *?thesis* **by** *blast*  
**qed**  
**show**  $(v \# V) @ T * \frown^* U \implies v \# V * \frown^* U$   
**by** (*metis 1 Con-sym Resid.simps(1) append-Nil*)  
**show**  $(v \# V) @ T * \frown^* U \implies T * \frown^* U * \setminus^* (v \# V)$   
**using**  $T U Con-sym$   
**by** (*metis 1 Con-initial-right Resid-cons(1-2) append.simps(2) ind self-append-conv*)  
**show**  $T * \frown^* (v \# V) @ U \implies T * \frown^* v \# V$   
**using** 2 **by** *fastforce*  
**show**  $T * \frown^* (v \# V) @ U \implies T * \setminus^* (v \# V) * \frown^* U$   
**using** 2 **by** *fastforce*  
**show**  $T * \frown^* v \# V \wedge T * \setminus^* (v \# V) * \frown^* U \implies T * \frown^* (v \# V) @ U$   
**proof** –  
**assume**  $Con: T * \frown^* v \# V \wedge T * \setminus^* (v \# V) * \frown^* U$   
**have**  $T * \frown^* (v \# V) @ U \longleftrightarrow T * \frown^* v \# (V @ U)$   
**by** *simp*  
**also have** ...  $\longleftrightarrow T * \frown^* [v] \wedge T * \setminus^* [v] * \frown^* V @ U$   
**using**  $T U Con-cons(2)$  **by** *simp*  
**also have** ...  $\longleftrightarrow T * \setminus^* [v] * \frown^* V @ U$   
**by** *fastforce*  
**also have** ...  $\longleftrightarrow True$   
**using**  $Con ind$

by (*metis Con-cons(2) Resid-cons(2) T U self-append-conv2*)  
 finally show ?thesis by blast  
 qed  
 show  $v \# V * \frown * U \wedge T * \frown * U * \backslash * (v \# V) \implies (v \# V) @ T * \frown * U$   
 proof -  
 assume *Con*:  $v \# V * \frown * U \wedge T * \frown * U * \backslash * (v \# V)$   
 have  $(v \# V) @ T * \frown * U \longleftrightarrow v \# (V @ T) * \frown * U$   
 by *simp*  
 also have  $\dots \longleftrightarrow [v] * \frown * U \wedge V @ T * \frown * U * \backslash * [v]$   
 using *T U Con-cons(1) by simp*  
 also have  $\dots \longleftrightarrow V @ T * \frown * U * \backslash * [v]$   
 by (*metis Con Con-cons(1) U*)  
 also have  $\dots \longleftrightarrow \text{True}$   
 using *Con ind*  
 by (*metis Con-cons(1) Con-sym Resid-cons(2) T U append-self-conv2*)  
 finally show ?thesis by blast  
 qed  
 qed  
 qed

lemma *Con-append*:

assumes  $T \neq []$  and  $U \neq []$  and  $V \neq []$   
 shows  $T @ U * \frown * V \longleftrightarrow T * \frown * V \wedge U * \frown * V * \backslash * T$   
 and  $T * \frown * U @ V \longleftrightarrow T * \frown * U \wedge T * \backslash * U * \frown * V$   
 using *assms Resid-append-ind by blast+*

lemma *Con-appendI* [*intro*]:

shows  $\llbracket T * \frown * V; U * \frown * V * \backslash * T \rrbracket \implies T @ U * \frown * V$   
 and  $\llbracket T * \frown * U; T * \backslash * U * \frown * V \rrbracket \implies T * \frown * U @ V$   
 by (*metis Con-append(1) Con-sym Resid.simps(1)*)<sup>+</sup>

lemma *Resid-append* [*intro, simp*]:

shows  $\llbracket T \neq []; T @ U * \frown * V \rrbracket \implies (T @ U) * \backslash * V = (T * \backslash * V) @ (U * \backslash * (V * \backslash * T))$   
 and  $\llbracket U \neq []; V \neq []; T * \frown * U @ V \rrbracket \implies T * \backslash * (U @ V) = (T * \backslash * U) * \backslash * V$   
 using *Resid-append-ind*  
 apply (*metis Con-sym Resid.simps(1) append-self-conv*)  
 using *Resid-append-ind*  
 by (*metis Resid.simps(1)*)

lemma *Resid-append2* [*simp*]:

assumes  $T \neq []$  and  $U \neq []$  and  $V \neq []$  and  $W \neq []$   
 and  $T @ U * \frown * V @ W$   
 shows  $(T @ U) * \backslash * (V @ W) =$   
 $(T * \backslash * V) * \backslash * W @ (U * \backslash * (V * \backslash * T)) * \backslash * (W * \backslash * (T * \backslash * V))$   
 using *assms Resid-append*  
 by (*metis Con-append(1-2) append-is-Nil-conv*)

lemma *append-is-composite-of*:

assumes *seq T U*

**shows** *composite-of T U (T @ U)*  
**unfolding** *composite-of-def*  
**using** *assms*  
**apply** (*intro conjI*)  
**apply** (*metis Arr.simps(1) Resid-Arr-self Resid-Ide-Arr-ind Arr-appendI<sub>P</sub>*  
*Resid-append-ind ide-char order-refl seq-char*)  
**apply** (*metis Arr.simps(1) Arr-appendI<sub>P</sub> Con-Arr-self Resid-Arr-self Resid-append-ind*  
*ide-char seq-char order-refl*)  
**by** (*metis Arr.simps(1) Con-Arr-self Con-append(1) Resid-Arr-self Arr-appendI<sub>P</sub>*  
*Ide-append-iff<sub>P</sub> Resid-append(1) ide-char seq-char order-refl*)

**sublocale** *rts-with-composites Resid*  
**using** *append-is-composite-of composable-def* **by** *unfold-locales blast*

**theorem** *is-rts-with-composites:*  
**shows** *rts-with-composites Resid*  
**..**

**lemma** *arr-append [intro, simp]:*  
**assumes** *seq T U*  
**shows** *arr (T @ U)*  
**using** *assms arrI<sub>P</sub> seq-char* **by** *simp*

**lemma** *arr-append-imp-seq:*  
**assumes** *T ≠ [] and U ≠ [] and arr (T @ U)*  
**shows** *seq T U*  
**using** *assms arr-char seq-char Arr-append-iff<sub>P</sub> seq-implies-Trgs-eq-Srcs* **by** *simp*

**lemma** *sources-append [simp]:*  
**assumes** *seq T U*  
**shows** *sources (T @ U) = sources T*  
**using** *assms*  
**by** (*meson append-is-composite-of sources-composite-of*)

**lemma** *targets-append [simp]:*  
**assumes** *seq T U*  
**shows** *targets (T @ U) = targets U*  
**using** *assms*  
**by** (*meson append-is-composite-of targets-composite-of*)

**lemma** *cong-respects-seq<sub>P</sub>:*  
**assumes** *seq T U and T \*~\* T' and U \*~\* U'*  
**shows** *seq T' U'*  
**by** (*meson assms cong-respects-seq*)

**lemma** *cong-append [intro]:*  
**assumes** *seq T U and T \*~\* T' and U \*~\* U'*  
**shows** *T @ U \*~\* T' @ U'*

**proof**

**have 1:**  $\bigwedge T U T' U'. \llbracket \text{seq } T U; T \sim^* T'; U \sim^* U' \rrbracket \implies \text{seq } T' U'$   
**using** *assms cong-respects-seq<sub>P</sub> by simp*  
**have 2:**  $\bigwedge T U T' U'. \llbracket \text{seq } T U; T \sim^* T'; U \sim^* U' \rrbracket \implies T @ U \sim^* T' @ U'$

**proof** –

**fix**  $T U T' U'$

**assume**  $TU: \text{seq } T U$  **and**  $TT': T \sim^* T'$  **and**  $UU': U \sim^* U'$

**have**  $T'U': \text{seq } T' U'$

**using**  $TU TT' UU'$  *cong-respects-seq<sub>P</sub> by simp*

**have 3:**  $\text{Ide } (T \sim^* T') \wedge \text{Ide } (T' \sim^* T) \wedge \text{Ide } (U \sim^* U') \wedge \text{Ide } (U' \sim^* U)$

**using**  $TU TT' UU'$  *ide-char by blast*

**have**  $(T @ U) \sim^* (T' @ U') =$

$((T \sim^* T') \sim^* U') @ U \sim^* ((T' \sim^* T) @ U' \sim^* (T \sim^* T'))$

**proof** –

**have 4:**  $T \neq [] \wedge U \neq [] \wedge T' \neq [] \wedge U' \neq []$

**using**  $TU TT' UU'$  *Arr.simps(1) seq-char ide-char by auto*

**moreover have**  $(T @ U) \sim^* (T' @ U') \neq []$

**proof** (*intro Con-appendI*)

**show**  $T \sim^* T' \neq []$

**using 3 by force**

**show**  $(T \sim^* T') \sim^* U' \neq []$

**using 3**  $T'U' \langle T \sim^* T' \neq [] \rangle$  *Con-Ide-iff seq-char by fastforce*

**show**  $U \sim^* ((T' @ U') \sim^* T) \neq []$

**proof** –

**have**  $U \sim^* ((T' @ U') \sim^* T) = U \sim^* ((T' \sim^* T) @ U' \sim^* (T \sim^* T'))$

**by** (*metis Con-appendI(1) Resid-append(1)  $\langle T \sim^* T' \rangle \sim^* U' \neq []$* )  
 $\langle T \sim^* T' \neq [] \rangle$  *calculation Con-sym*)

**also have**  $\dots = (U \sim^* (T' \sim^* T)) \sim^* (U' \sim^* (T \sim^* T'))$

**by** (*metis Arr.simps(1) Con-append(2) Resid-append(2)  $\langle T \sim^* T' \rangle \sim^* U' \neq []$* )  
 $\langle T \sim^* T' \rangle \sim^* U' \neq []$  *Con-implies-Arr(1) Con-sym*)

**also have**  $\dots = U \sim^* U'$

**by** (*metis (mono-tags, lifting) 3 Ide.simps(1) Resid-Ide(1) Srcs-Resid TU*)  
 $\langle T \sim^* T' \rangle \sim^* U' \neq []$  *Con-Ide-iff seq-char*)

**finally show** *?thesis*

**using 3**  $UU'$  *by force*

**qed**

**qed**

**ultimately show** *?thesis*

**using** *Resid-append2 [of T U T' U'] seq-char*

**by** (*metis Con-append(2) Con-sym Resid-append(2) Resid.simps(1)*)

**qed**

**moreover have** *Ide ...*

**proof**

**have 3:**  $\text{Ide } (T \sim^* T') \wedge \text{Ide } (T' \sim^* T) \wedge \text{Ide } (U \sim^* U') \wedge \text{Ide } (U' \sim^* U)$

**using**  $TU TT' UU'$  *ide-char by blast*

**show 4:**  $\text{Ide } ((T \sim^* T') \sim^* U')$

**using**  $TU T'U' TT' UU'$  *1 3*

**by** (*metis (full-types) Srcs-Resid Con-Ide-iff Resid-Ide-Arr-ind seq-char*)

**show 5:**  $\text{Ide } (U \sim^* ((T' \sim^* T) @ U' \sim^* (T \sim^* T')))$

**proof** –  
**have**  $U * \setminus * (T' * \setminus * T) = U$   
**by** (*metis* (*full-types*) 3 *TT'* *TU* *Con-Ide-iff* *Resid-Ide(1)* *Srcs-Resid*  
*con-char* *seq-char* *prfx-implies-con*)  
**moreover have**  $U' * \setminus * (T * \setminus * T') = U'$   
**by** (*metis* 3 4 *Ide.simps(1)* *Resid-Ide(1)*)  
**ultimately show** *?thesis*  
**by** (*metis* 3 4 *Arr.simps(1)* *Con-append(2)* *Ide.simps(1)* *Resid-append(2)*  
*TU* *Con-sym* *seq-char*)  
**qed**  
**show**  $\text{Trgs } ((T * \setminus * T') * \setminus * U') \subseteq \text{Srcs } (U * \setminus * (T' * \setminus * T @ U' * \setminus * (T * \setminus * T')))$   
**by** (*metis* 4 5 *Arr-append-iff<sub>P</sub>* *Ide.simps(1)* *Nil-is-append-conv*  
*calculation* *Con-imp-Arr-Resid*)  
**qed**  
**ultimately show**  $T @ U * \lesssim^* T' @ U'$   
**using** *ide-char* **by** *presburger*  
**qed**  
**show**  $T @ U * \lesssim^* T' @ U'$   
**using** *assms* 2 **by** *simp*  
**show**  $T' @ U' * \lesssim^* T @ U$   
**using** *assms* 1 2 *cong-symmetric* **by** *blast*  
**qed**

**lemma** *cong-cons* [*intro*]:  
**assumes** *seq* [t] *U* **and**  $t \sim t'$  **and**  $U * \sim * U'$   
**shows**  $t \# U * \sim * t' \# U'$   
**using** *assms* *cong-append* [*of* [t] *U* [t'] *U'*]  
**by** (*simp* *add*: *R.prfx-implies-con* *ide-char*)

**lemma** *cong-append-ideI* [*intro*]:  
**assumes** *seq* *T* *U*  
**shows**  $\text{ide } T \implies T @ U * \sim * U$  **and**  $\text{ide } U \implies T @ U * \sim * T$   
**and**  $\text{ide } T \implies U * \sim * T @ U$  **and**  $\text{ide } U \implies T * \sim * T @ U$   
**proof** –

**show** 1:  $\text{ide } T \implies T @ U * \sim * U$   
**using** *assms*  
**by** (*metis* *append-is-composite-of* *composite-ofE* *resid-arr-ide* *prfx-implies-con*  
*con-sym*)  
**show** 2:  $\text{ide } U \implies T @ U * \sim * T$   
**by** (*meson* *assms* *append-is-composite-of* *composite-ofE* *ide-backward-stable*)  
**show**  $\text{ide } T \implies U * \sim * T @ U$   
**using** 1 *cong-symmetric* **by** *auto*  
**show**  $\text{ide } U \implies T * \sim * T @ U$   
**using** 2 *cong-symmetric* **by** *auto*  
**qed**

**lemma** *cong-cons-ideI* [*intro*]:  
**assumes** *seq* [t] *U* **and** *R.ide* *t*  
**shows**  $t \# U * \sim * U$  **and**  $U * \sim * t \# U$

**using** *assms cong-append-ideI* [of [t] U]  
**by** (*auto simp add: ide-char*)

**lemma** *prfx-decomp*:

**assumes**  $[t] \ast \lesssim^* [u]$

**shows**  $[t] @ [u \setminus t] \ast \sim^* [u]$

**proof**

**show**  $1: [u] \ast \lesssim^* [t] @ [u \setminus t]$

**using** *assms*

**by** (*metis Con-imp-Arr-Resid Con-rec(3) Resid.simps(3) Resid-rec(3) R.con-sym  
append.left-neutral append-Cons arr-char cong-reflexive list.distinct(1)*)

**show**  $[t] @ [u \setminus t] \ast \lesssim^* [u]$

**proof** –

**have**  $([t] @ [u \setminus t]) \ast \setminus^* [u] = ([t] \ast \setminus^* [u]) @ ([u \setminus t] \ast \setminus^* [u \setminus t])$

**using** *assms*

**by** (*metis Arr-Resid-single Con-Arr-self Con-appendI(1) Con-sym Resid-append(1)  
Resid-rec(1) con-char list.discI prfx-implies-con*)

**moreover have** *Ide ...*

**using** *assms*

**by** (*metis 1 Con-sym append-Nil2 arr-append-imp-seq calculation cong-append-ideI(4)  
ide-backward-stable Con-implies-Arr(2) Resid-Arr-self con-char ide-char  
prfx-implies-con arr-resid-iff-con*)

**ultimately show** *?thesis*

**using** *ide-char* **by** *presburger*

**qed**

**qed**

**lemma** *composite-of-single-single*:

**assumes** *R.composite-of t u v*

**shows** *composite-of*  $[t] [u] ([t] @ [u])$

**proof**

**show**  $[t] \ast \lesssim^* [t] @ [u]$

**proof** –

**have**  $[t] \ast \setminus^* ([t] @ [u]) = ([t] \ast \setminus^* [t]) \ast \setminus^* [u]$

**using** *assms* **by** *auto*

**moreover have** *Ide ...*

**by** (*metis (no-types, lifting) Con-implies-Arr(2) R.bounded-imp-con  
R.con-composite-of-iff R.con-prfx-composite-of(1) assms resid-ide-arr  
Con-rec(1) Resid.simps(3) Resid-Arr-self con-char ide-char*)

**ultimately show** *?thesis*

**using** *ide-char* **by** *presburger*

**qed**

**show**  $([t] @ [u]) \ast \setminus^* [t] \ast \sim^* [u]$

**using** *assms*

**by** (*metis <prfx [t] ([t] @ [u])> append-is-composite-of arr-append-imp-seq  
composite-ofE con-def not-Cons-self2 Con-implies-Arr(2) arr-char null-char  
prfx-implies-con*)

**qed**

end

#### 2.4.4 Paths in a Weakly Extensional RTS

```
locale paths-in-weakly-extensional-rts =  
  R: weakly-extensional-rts +  
  paths-in-rts
```

begin

```
lemma ex-un-Src:  
  assumes Arr T  
  shows  $\exists! a. a \in \text{Srcs } T$   
  using assms  
  by (simp add: Srcs-simpP R.arr-has-un-source)
```

```
fun Src  
where Src T = R.src (hd T)
```

```
lemma Srcs-simpPWE:  
  assumes Arr T  
  shows Srcs T = {Src T}  
  proof -  
    have [R.src (hd T)]  $\in$  sources T  
    by (metis Arr-imp-arr-hd Con-single-ide-ind Ide.simps(2) Srcs-simpP assms  
        con-char ide-char in-sourcesI con-sym R.ide-src R.src-in-sources)  
    hence R.src (hd T)  $\in$  Srcs T  
    using assms  
    by (metis Srcs.elims Arr-has-Src list.sel(1) R.arr-iff-has-source R.src-in-sources)  
    thus ?thesis  
    using assms ex-un-Src by auto  
  qed
```

```
lemma ex-un-Trg:  
  assumes Arr T  
  shows  $\exists! b. b \in \text{Trgs } T$   
  using assms  
  apply (induct T)  
  apply auto[1]  
  by (metis Con-Arr-self Ide-implies-Arr Resid-Arr-self Srcs-Resid ex-un-Src)
```

```
fun Trg  
where Trg [] = R.null  
  | Trg [t] = R.trg t  
  | Trg (t # T) = Trg T
```

```
lemma Trg-simp [simp]:  
  shows  $T \neq [] \implies \text{Trg } T = \text{R.trg } (\text{last } T)$   
  apply (induct T)
```

**apply** *auto*  
**by** (*metis* *Trg.simps(3)* *list.exhaust-sel*)

**lemma** *Trgs-simp<sub>PWE</sub>* [*simp*]:  
**assumes** *Arr T*  
**shows**  $\text{Trgs } T = \{\text{Trg } T\}$   
**using** *assms*  
**by** (*metis* *Arr-imp-arr-last* *Con-Arr-self* *Con-imp-Arr-Resid* *R.trg-in-targets*  
*Srcs.simps(1)* *Srcs-Resid* *Srcs-simp<sub>PWE</sub>* *Trg-simp* *insertE* *insert-absorb* *insert-not-empty*  
*Trgs-simp<sub>P</sub>*)

**lemma** *Src-resid* [*simp*]:  
**assumes**  $T \text{ }^* \frown^* U$   
**shows**  $\text{Src } (T \text{ }^* \backslash^* U) = \text{Trg } U$   
**using** *assms* *Con-imp-Arr-Resid* *Con-implies-Arr(2)* *Srcs-Resid* *Srcs-simp<sub>PWE</sub>* **by force**

**lemma** *Trg-resid-sym*:  
**assumes**  $T \text{ }^* \frown^* U$   
**shows**  $\text{Trg } (T \text{ }^* \backslash^* U) = \text{Trg } (U \text{ }^* \backslash^* T)$   
**using** *assms* *Con-imp-Arr-Resid* *Con-sym* *Trgs-Resid-sym* **by auto**

**lemma** *Src-append* [*simp*]:  
**assumes** *seq T U*  
**shows**  $\text{Src } (T @ U) = \text{Src } T$   
**using** *assms*  
**by** (*metis* *Arr.simps(1)* *Src.simps* *hd-append* *seq-char*)

**lemma** *Trg-append* [*simp*]:  
**assumes** *seq T U*  
**shows**  $\text{Trg } (T @ U) = \text{Trg } U$   
**using** *assms*  
**by** (*metis* *Ide.simps(1)* *Resid.simps(1)* *Trg-simp* *append-is-Nil-conv* *ide-char* *ide-trg*  
*last-appendR* *seqE* *trg-def*)

**lemma** *Arr-append-iff<sub>PWE</sub>*:  
**assumes**  $T \neq []$  **and**  $U \neq []$   
**shows**  $\text{Arr } (T @ U) \longleftrightarrow \text{Arr } T \wedge \text{Arr } U \wedge \text{Trg } T = \text{Src } U$   
**using** *assms* *Arr-appendE<sub>P</sub>* *Srcs-simp<sub>PWE</sub>* **by auto**

**lemma** *Arr-consI<sub>PWE</sub>* [*intro*, *simp*]:  
**assumes** *R.arr t* **and** *Arr U* **and**  $R.\text{trg } t = \text{Src } U$   
**shows**  $\text{Arr } (t \# U)$   
**using** *assms*  
**by** (*metis* *Arr.simps(2)* *Srcs-simp<sub>PWE</sub>* *Trg.simps(2)* *Trgs.simps(2)* *Trgs-simp<sub>PWE</sub>*  
*dual-order.eq-iff* *Arr-consI<sub>P</sub>*)

**lemma** *Arr-consE* [*elim*]:  
**assumes**  $\text{Arr } (t \# U)$   
**and**  $\llbracket R.\text{arr } t; U \neq [] \implies \text{Arr } U; U \neq [] \implies R.\text{trg } t = \text{Src } U \rrbracket \implies \textit{thesis}$

**shows** *thesis*  
**using** *assms*  
**by** (*metis* *Arr-append-iff<sub>PWE</sub>* *Trg.simps(2)* *append-Cons* *append-Nil* *list.distinct(1)* *Arr.simps(2)*)

**lemma** *Arr-appendI<sub>PWE</sub>* [*intro*, *simp*]:  
**assumes** *Arr T* **and** *Arr U* **and** *Trg T = Src U*  
**shows** *Arr (T @ U)*  
**using** *assms*  
**by** (*metis* *Arr.simps(1)* *Arr-append-iff<sub>PWE</sub>*)

**lemma** *Arr-appendE<sub>PWE</sub>* [*elim*]:  
**assumes** *Arr (T @ U)* **and**  $T \neq []$  **and**  $U \neq []$   
**and**  $\llbracket \text{Arr } T; \text{Arr } U; \text{Trg } T = \text{Src } U \rrbracket \implies \textit{thesis}$   
**shows** *thesis*  
**using** *assms* *Arr-append-iff<sub>PWE</sub>* *seq-implies-Trgs-eq-Srcs* **by** *force*

**lemma** *Ide-append-iff<sub>PWE</sub>*:  
**assumes**  $T \neq []$  **and**  $U \neq []$   
**shows**  $\text{Ide } (T @ U) \longleftrightarrow \text{Ide } T \wedge \text{Ide } U \wedge \text{Trg } T = \text{Src } U$   
**using** *assms* *Ide-char*  
**apply** (*intro iffI*)  
**by** *force auto*

**lemma** *Ide-appendI<sub>PWE</sub>* [*intro*, *simp*]:  
**assumes** *Ide T* **and** *Ide U* **and** *Trg T = Src U*  
**shows** *Ide (T @ U)*  
**using** *assms*  
**by** (*metis* *Ide.simps(1)* *Ide-append-iff<sub>PWE</sub>*)

**lemma** *Ide-appendE* [*elim*]:  
**assumes** *Ide (T @ U)* **and**  $T \neq []$  **and**  $U \neq []$   
**and**  $\llbracket \text{Ide } T; \text{Ide } U; \text{Trg } T = \text{Src } U \rrbracket \implies \textit{thesis}$   
**shows** *thesis*  
**using** *assms* *Ide-append-iff<sub>PWE</sub>* **by** *metis*

**lemma** *Ide-consI* [*intro*, *simp*]:  
**assumes** *R.ide t* **and** *Ide U* **and** *R.trg t = Src U*  
**shows** *Ide (t # U)*  
**using** *assms*  
**by** (*simp add: Ide-char*)

**lemma** *Ide-consE* [*elim*]:  
**assumes** *Ide (t # U)*  
**and**  $\llbracket \text{R.ide } t; U \neq [] \rrbracket \implies \text{Ide } U; U \neq [] \implies \text{R.trg } t = \text{Src } U \rrbracket \implies \textit{thesis}$   
**shows** *thesis*  
**using** *assms*  
**by** (*metis* *Con-rec(4)* *Ide.simps(2)* *Ide-imp-Ide-hd* *Ide-imp-Ide-tl* *R.trg-def* *R.trg-ide* *Resid-Arr-Ide-ind* *Trg.simps(2)* *ide-char* *list.sel(1)* *list.sel(3)* *list.simps(3)*)

*Src-resid ide-def*)

**lemma** *Ide-imp-Src-eq-Trg*:  
**assumes** *Ide T*  
**shows**  $\text{Src } T = \text{Trg } T$   
**using** *assms*  
**by** (*metis Ide.simps(1) Src-resid ide-char ide-def*)

**end**

## 2.4.5 Paths in a Confluent RTS

Here we show that confluence of an RTS extends to confluence of the RTS of its paths.

**locale** *paths-in-confluent-rts* =  
*paths-in-rts* +  
*R: confluent-rts*  
**begin**

**lemma** *confluence-single*:  
**assumes**  $\bigwedge t u. R.\text{coinitial } t u \implies t \frown u$   
**shows**  $\llbracket R.\text{arr } t; \text{Arr } U; R.\text{sources } t = \text{Srcs } U \rrbracket \implies [t] \text{ }^* \frown^* U$   
**proof** (*induct U arbitrary: t*)  
**show**  $\bigwedge t. \llbracket R.\text{arr } t; \text{Arr } []; R.\text{sources } t = \text{Srcs } [] \rrbracket \implies [t] \text{ }^* \frown^* []$   
**by** *simp*  
**fix**  $t u U$   
**assume** *ind*:  $\bigwedge t. \llbracket R.\text{arr } t; \text{Arr } U; R.\text{sources } t = \text{Srcs } U \rrbracket \implies [t] \text{ }^* \frown^* U$   
**assume**  $t: R.\text{arr } t$   
**assume**  $uU: \text{Arr } (u \# U)$   
**assume** *coinitial*:  $R.\text{sources } t = \text{Srcs } (u \# U)$   
**hence**  $1: R.\text{coinitial } t u$   
**using**  $t uU$   
**by** (*metis Arr.simps(2) Con-implies-Arr(1) Con-imp-eq-Srcs Con-initial-left Srcs.simps(2) Con-Arr-self R.coinitial-iff*)  
**show**  $[t] \text{ }^* \frown^* u \# U$   
**proof** (*cases U = []*)  
**show**  $U = [] \implies ?thesis$   
**using** *assms t uU coinitial R.coinitial-iff* **by** *fastforce*  
**assume**  $U: U \neq []$   
**show** *?thesis*  
**proof** –  
**have**  $2: \text{Arr } [t \setminus u] \wedge \text{Arr } U \wedge \text{Srcs } [t \setminus u] = \text{Srcs } U$   
**using** *assms 1 t uU U R.arr-resid-iff-con*  
**apply** (*intro conjI*)  
**apply** *simp*  
**apply** (*metis Con-Arr-self Con-implies-Arr(2) Resid-cons(2)*)  
**by** (*metis (full-types) Con-cons(2) Srcs.simps(2) Srcs-Resid Trgs.simps(2) Con-Arr-self Con-imp-eq-Srcs list.simps(3) R.sources-resid*)  
**have**  $[t] \text{ }^* \frown^* u \# U \longleftrightarrow t \frown u \wedge [t \setminus u] \text{ }^* \frown^* U$   
**using**  $U$  *Con-rec(3) [of U t u]* **by** *simp*

```

    also have ...  $\longleftrightarrow$  True
      using assms t uU U 1 2 ind by force
    finally show ?thesis by blast
  qed
qed
qed

lemma confluence-ind:
shows  $\llbracket \text{Arr } T; \text{Arr } U; \text{Srcs } T = \text{Srcs } U \rrbracket \implies T^* \frown^* U$ 
proof (induct T arbitrary: U)
  show  $\bigwedge U. \llbracket \text{Arr } []; \text{Arr } U; \text{Srcs } [] = \text{Srcs } U \rrbracket \implies []^* \frown^* U$ 
    by simp
  fix t T U
  assume ind:  $\bigwedge U. \llbracket \text{Arr } T; \text{Arr } U; \text{Srcs } T = \text{Srcs } U \rrbracket \implies T^* \frown^* U$ 
  assume tT:  $\text{Arr } (t \# T)$ 
  assume U:  $\text{Arr } U$ 
  assume coinitial:  $\text{Srcs } (t \# T) = \text{Srcs } U$ 
  show  $t \# T^* \frown^* U$ 
  proof (cases  $T = []$ )
    show  $T = [] \implies ?thesis$ 
      using U tT coinitial confluence-single [of t U] R.confluence by simp
    assume  $T \neq []$ 
    show ?thesis
    proof -
      have 1:  $[t]^* \frown^* U$ 
        using tT U coinitial R.confluence
        by (metis R.arr-def Srcs.simps(2) T Con-Arr-self Con-imp-eq-Srcs
          Con-initial-right Con-rec(4) confluence-single)
      moreover have  $T^* \frown^* U^* \setminus^* [t]$ 
        using 1 tT U T coinitial ind [of U^* \setminus^* [t]]
        by (metis (full-types) Con-imp-Arr-Resid Arr-iff-Con-self Con-implies-Arr(2)
          Con-imp-eq-Srcs Con-sym R.sources-resid Srcs.simps(2) Srcs-Resid
          Trgs.simps(2) Con-rec(4))
      ultimately show ?thesis
        using Con-cons(1) [of T U t] by fastforce
    qed
  qed
qed
qed

```

```

lemma confluenceP:
assumes coinitial T U
shows con T U
  using assms confluence-ind sources-charP coinitial-def con-char by auto

```

```

sublocale confluent-rts Resid
  apply (unfold-locales)
  using confluenceP by simp

```

```

lemma is-confluent-rts:

```

**shows** *confluent-rts Resid*

..

**end**

## 2.4.6 Simulations Lift to Paths

In this section we show that a simulation from RTS  $A$  to RTS  $B$  determines a simulation from the RTS of paths in  $A$  to the RTS of paths in  $B$ . In other words, the path-RTS construction is functorial with respect to simulation.

**context** *simulation*

**begin**

**interpretation**  $P_A$ : *paths-in-rts A*

..

**interpretation**  $P_B$ : *paths-in-rts B*

..

**lemma** *map-Resid-single*:

**shows**  $P_A.con\ T\ [u] \implies map\ F\ (P_A.Resid\ T\ [u]) = P_B.Resid\ (map\ F\ T)\ [F\ u]$

**apply** (*induct T arbitrary: u*)

**apply** *simp*

**proof** –

**fix**  $t\ u\ T$

**assume** *ind*:  $\bigwedge u. P_A.con\ T\ [u] \implies map\ F\ (P_A.Resid\ T\ [u]) = P_B.Resid\ (map\ F\ T)\ [F\ u]$

**assume** *1*:  $P_A.con\ (t\ \# \ T)\ [u]$

**show**  $map\ F\ (P_A.Resid\ (t\ \# \ T)\ [u]) = P_B.Resid\ (map\ F\ (t\ \# \ T))\ [F\ u]$

**proof** (*cases T = []*)

**show**  $T = [] \implies ?thesis$

**using** *1 P\_A.null-char by fastforce*

**assume**  $T$ :  $T \neq []$

**show** *?thesis*

**using**  $T\ 1\ ind\ P_A.con-def\ P_A.null-char\ P_A.Con-rec(2)\ P_A.Resid-rec(2)\ P_B.Con-rec(2)\ P_B.Resid-rec(2)$

**apply** *simp*

**by** (*metis A.con-sym Nil-is-map-conv preserves-con preserves-resid*)

**qed**

**qed**

**lemma** *map-Resid*:

**shows**  $P_A.con\ T\ U \implies map\ F\ (P_A.Resid\ T\ U) = P_B.Resid\ (map\ F\ T)\ (map\ F\ U)$

**apply** (*induct U arbitrary: T*)

**using**  $P_A.Resid.simps(1)\ P_A.con-char\ P_A.con-sym$

**apply** *blast*

**proof** –

**fix**  $u\ U\ T$

**assume** *ind*:  $\bigwedge T. P_A.con\ T\ U \implies$

$map\ F\ (P_A.Resid\ T\ U) = P_B.Resid\ (map\ F\ T)\ (map\ F\ U)$

**assume** *1*:  $P_A.con\ T\ (u\ \# \ U)$

```

show map F (PA.Resid T (u # U)) = PB.Resid (map F T) (map F (u # U))
proof (cases U = [])
  show U = []  $\implies$  ?thesis
    using 1 map-Resid-single by force
  assume U: U  $\neq$  []
  have PB.Resid (map F T) (map F (u # U)) =
    PB.Resid (PB.Resid (map F T) [F u]) (map F U)
    using U 1 PB.Resid-cons(2)
  apply simp
  by (metis PB.Arr.simps(1) PB.Con-consI(2) PB.Con-implies-Arr(1) list.map-disc-iff)
  also have ... = map F (PA.Resid (PA.Resid T [u]) U)
    using U 1 ind
    by (metis PA.Con-initial-right PA.Resid-cons(2) PA.con-char map-Resid-single)
  also have ... = map F (PA.Resid T (u # U))
    using 1 PA.Resid-cons(2) PA.con-char U by auto
  finally show ?thesis by simp
qed
qed

```

**lemma** *preserves-paths*:

```

shows PA.Arr T  $\implies$  PB.Arr (map F T)
  by (metis PA.Con-Arr-self PA.conIP PB.Arr-iff-Con-self map-Resid map-is-Nil-conv)

```

**interpretation** *Fx*: simulation P<sub>A</sub>.Resid P<sub>B</sub>.Resid  $\langle \lambda T. \text{if } P_A.Arr T \text{ then map F T else } [] \rangle$   
**proof**

```

let ?Fx =  $\lambda T. \text{if } P_A.Arr T \text{ then map F T else } []$ 
show  $\bigwedge T. \neg P_A.arr T \implies ?Fx T = P_B.null$ 
  by (simp add: PA.arr-char PB.null-char)
show  $\bigwedge T U. P_A.con T U \implies P_B.con (?Fx T) (?Fx U)$ 
  using PA.Con-implies-Arr(1) PA.Con-implies-Arr(2) PA.con-char map-Resid by fastforce
show  $\bigwedge T U. P_A.con T U \implies ?Fx (P_A.Resid T U) = P_B.Resid (?Fx T) (?Fx U)$ 
  by (simp add: PA.Con-imp-Arr-Resid PA.Con-implies-Arr(1) PA.Con-implies-Arr(2)
    PA.con-char map-Resid)

```

**qed**

**lemma** *lifts-to-paths*:

```

shows simulation PA.Resid PB.Resid  $(\lambda T. \text{if } P_A.Arr T \text{ then map F T else } [])$ 

```

..

**end**

## 2.4.7 Normal Sub-RTS's Lift to Paths

Here we show that a normal sub-RTS  $N$  of an RTS  $R$  lifts to a normal sub-RTS of the RTS of paths in  $N$ , and that it is coherent if  $N$  is.

```

locale paths-in-rtswith-normal =
  R: rts +
  N: normal-sub-rtswith-normal +
  paths-in-rtswith-normal

```

**begin**

We define a “normal path” to be a path that consists entirely of normal transitions. We show that the collection of all normal paths is a normal sub-RTS of the RTS of paths.

**definition** *NPath*

**where**  $NPath\ T \equiv (Arr\ T \wedge set\ T \subseteq \mathfrak{N})$

**lemma** *Ide-implies-NPath*:

**assumes** *Ide*  $T$

**shows**  $NPath\ T$

**using** *assms*

**by** (*metis Ball-Collect NPath-def Ide-implies-Arr N.ide-closed set-Ide-subset-ide subsetI*)

**lemma** *NPath-implies-Arr*:

**assumes**  $NPath\ T$

**shows**  $Arr\ T$

**using** *assms NPath-def* **by** *simp*

**lemma** *NPath-append*:

**assumes**  $T \neq []$  **and**  $U \neq []$

**shows**  $NPath\ (T @ U) \longleftrightarrow NPath\ T \wedge NPath\ U \wedge Trgs\ T \subseteq Srcs\ U$

**using** *assms NPath-def* **by** *auto*

**lemma** *NPath-appendI* [*intro, simp*]:

**assumes**  $NPath\ T$  **and**  $NPath\ U$  **and**  $Trgs\ T \subseteq Srcs\ U$

**shows**  $NPath\ (T @ U)$

**using** *assms NPath-def* **by** *simp*

**lemma** *NPath-Resid-single-Arr*:

**shows**  $\llbracket t \in \mathfrak{N}; Arr\ U; R.sources\ t = Srcs\ U \rrbracket \Longrightarrow NPath\ (Resid\ [t]\ U)$

**proof** (*induct*  $U$  *arbitrary*:  $t$ )

**show**  $\bigwedge t. \llbracket t \in \mathfrak{N}; Arr\ []; R.sources\ t = Srcs\ [] \rrbracket \Longrightarrow NPath\ (Resid\ [t]\ [])$

**by** *simp*

**fix**  $t\ u\ U$

**assume** *ind*:  $\bigwedge t. \llbracket t \in \mathfrak{N}; Arr\ U; R.sources\ t = Srcs\ U \rrbracket \Longrightarrow NPath\ (Resid\ [t]\ U)$

**assume**  $t: t \in \mathfrak{N}$

**assume**  $uU: Arr\ (u \# U)$

**assume** *src*:  $R.sources\ t = Srcs\ (u \# U)$

**show**  $NPath\ (Resid\ [t]\ (u \# U))$

**proof** (*cases*  $U = []$ )

**show**  $U = [] \Longrightarrow ?thesis$

**using** *NPath-def*  $t\ src$

**apply** *simp*

**by** (*metis Arr.simps(2) R.arr-resid-iff-con R.coinitialI N.forward-stable N.elements-are-arr*  $uU$ )

**assume**  $U: U \neq []$

**show** *?thesis*

**proof** –

```

have  $NPath (Resid [t] (u \# U)) \longleftrightarrow NPath (Resid [t \setminus u] U)$ 
  using  $t U uU src$ 
by (metis  $Arr.simps(2)$   $Con-implies-Arr(1)$   $Resid-rec(3)$   $Con-rec(3)$   $R.arr-resid-iff-con$ )
also have  $\dots \longleftrightarrow True$ 
proof –
  have  $t \setminus u \in \mathfrak{N}$ 
    using  $t U uU src N.forward-stable [of t u]$ 
    by (metis  $Con-Arr-self$   $Con-imp-eq-Srcs$   $Con-initial-left$ 
       $Srcs.simps(2)$   $inf.idem$   $Arr-has-Src$   $R.coinitial-def$ )
  moreover have  $Arr U$ 
    using  $U uU$ 
    by (metis  $Arr.simps(3)$   $neq-Nil-conv$ )
  moreover have  $R.sources (t \setminus u) = Srcs U$ 
    using  $t uU src$ 
    by (metis  $Con-Arr-self$   $Srcs.simps(2)$   $U$   $calculation(1)$   $Con-imp-eq-Srcs$ 
       $Con-rec(4)$   $N.elements-are-arr$   $R.sources-resid$   $R.arr-resid-iff-con$ )
  ultimately show ?thesis
    using  $ind [of t \setminus u]$  by simp
  qed
finally show ?thesis by blast
qed
qed
qed

```

**lemma** *NPath-Resid-Arr-single*:

```

shows  $\llbracket NPath T; R.arr u; Srcs T = R.sources u \rrbracket \Longrightarrow NPath (Resid T [u])$ 
proof (induct  $T$  arbitrary:  $u$ )
  show  $\bigwedge u. \llbracket NPath []; R.arr u; Srcs [] = R.sources u \rrbracket \Longrightarrow NPath (Resid [] [u])$ 
    by simp
  fix  $t u T$ 
  assume  $ind: \bigwedge u. \llbracket NPath T; R.arr u; Srcs T = R.sources u \rrbracket \Longrightarrow NPath (Resid T [u])$ 
  assume  $tT: NPath (t \# T)$ 
  assume  $u: R.arr u$ 
  assume  $src: Srcs (t \# T) = R.sources u$ 
  show  $NPath (Resid (t \# T) [u])$ 
proof (cases  $T = []$ )
  show  $T = [] \Longrightarrow ?thesis$ 
    using  $tT u src NPath-def$ 
    by (metis  $Arr.simps(2)$   $NPath-Resid-single-Arr$   $Srcs.simps(2)$   $list.set-intros(1)$   $subsetD$ )
  assume  $T: T \neq []$ 
  have  $R.coinitial u t$ 
    by (metis  $R.coinitialI$   $Srcs.simps(3)$   $T$   $list.exhaust-sel src u$ )
  hence  $con: t \frown u$ 
    using  $tT T u src R.con-sym NPath-def$ 
    by (metis  $N.forward-stable$   $N.elements-are-arr$   $R.not-arr-null$ 
       $list.set-intros(1)$   $R.conI$   $subsetD$ )
  have  $1: NPath (Resid (t \# T) [u]) \longleftrightarrow NPath ((t \setminus u) \# Resid T [u \setminus t])$ 
proof –
  have  $t \# T \frown^* [u]$ 

```

```

proof –
  have  $\mathcal{Q}$ :  $[t] \text{ }^* \frown^* [u]$ 
    by (simp add: Con-rec(1) con)
  moreover have  $T \text{ }^* \frown^* \text{Resid } [u] [t]$ 
  proof –
    have  $NPath\ T$ 
      using  $tT\ T\ NPath\ def$ 
      by (metis NPath-append append-Cons append-Nil)
    moreover have  $\mathcal{R}$ :  $R.arr\ (u \setminus t)$ 
      using con by (meson R.arr-resid-iff-con R.con-sym)
    moreover have  $Srcs\ T = R.sources\ (u \setminus t)$ 
      using  $tT\ T\ u\ src\ con$ 
      by (metis \mathcal{R} Arr-iff-Con-self Con-cons(2) Con-imp-eq-Srcs
        R.sources-resid Srcs-Resid Trgs.simps(2) NPath-implies-Arr list.discI
        R.arr-resid-iff-con)
    ultimately show ?thesis
      using  $\mathcal{Q}\ ind\ [of\ u \setminus t]\ NPath\ def$  by auto
  qed
  ultimately show ?thesis
    using  $tT\ T\ u\ src\ Con-cons(1)\ [of\ T\ [u]\ t]$  by simp
  qed
  thus ?thesis
    using  $tT\ T\ u\ src\ Resid-cons(1)\ [of\ T\ t\ [u]]\ Resid-rec(2)$  by presburger
  qed
also have  $\dots \longleftrightarrow True$ 
proof –
  have  $\mathcal{Q}$ :  $t \setminus u \in \mathfrak{N} \wedge R.arr\ (u \setminus t)$ 
    using  $tT\ u\ src\ con\ NPath\ def$ 
    by (meson R.arr-resid-iff-con R.con-sym N.forward-stable \langle R.coinitial\ u\ \rangle
      list.set-intros(1) subsetD)
  moreover have  $\mathcal{R}$ :  $NPath\ (T \text{ }^* \setminus^* [u \setminus t])$ 
    using  $tT\ ind\ [of\ u \setminus t]\ NPath\ def$ 
    by (metis Con-Arr-self Con-imp-eq-Srcs Con-rec(4) R.arr-resid-iff-con
      R.sources-resid Srcs.simps(2) T\ calculation\ insert-subset\ list.exhaust
      list.simps(15) Arr.simps(3))
  moreover have  $R.targets\ (t \setminus u) \subseteq Srcs\ (Resid\ T\ [u \setminus t])$ 
    using  $tT\ T\ u\ src\ NPath\ def$ 
    by (metis \mathcal{R} Arr.simps(1) R.targets-resid-sym Srcs-Resid-Arr-single\ con\ subset-refl)
  ultimately show ?thesis
    using  $NPath\ def$ 
    by (metis Arr-consI_P N.elements-are-arr\ insert-subset\ list.simps(15))
  qed
  finally show ?thesis by blast
  qed
qed

```

**lemma**  $NPath\ Resid\ [simp]$ :

**shows**  $\llbracket NPath\ T; Arr\ U; Srcs\ T = Srcs\ U \rrbracket \implies NPath\ (T \text{ }^* \setminus^* U)$

**proof** (*induct T arbitrary: U*)

```

show  $\bigwedge U. \llbracket NPath \ []; Arr U; Srcs \ [] = Srcs U \rrbracket \implies NPath \ ([\ ] \ * \ * \ U)$ 
  by simp
fix  $t T U$ 
assume  $ind: \bigwedge U. \llbracket NPath T; Arr U; Srcs T = Srcs U \rrbracket \implies NPath (T \ * \ * \ U)$ 
assume  $tT: NPath (t \ \# \ T)$ 
assume  $U: Arr U$ 
assume  $Coinitial: Srcs (t \ \# \ T) = Srcs U$ 
show  $NPath ((t \ \# \ T) \ * \ * \ U)$ 
proof (cases  $T = []$ )
  show  $T = [] \implies ?thesis$ 
    using  $tT U Coinitial NPath-Resid-single-Arr$  [of  $t U$ ] NPath-def by force
  assume  $T: T \neq []$ 
  have  $0: NPath ((t \ \# \ T) \ * \ * \ U) \longleftrightarrow NPath ([t] \ * \ * \ U @ T \ * \ * \ (U \ * \ * \ [t]))$ 
  proof –
    have  $U \neq []$ 
      using  $U$  by auto
    moreover have  $(t \ \# \ T) \ * \ \frown \ * \ U$ 
    proof –
      have  $t \in \mathfrak{N}$ 
        using  $tT NPath-def$  by auto
      moreover have  $R.sources\ t = Srcs\ U$ 
        using Coinitial
        by (metis  $Srcs.elims\ U\ list.sel(1)\ Arr-has-Src$ )
      ultimately have  $1: [t] \ * \ \frown \ * \ U$ 
        using  $U NPath-Resid-single-Arr$  [of  $t U$ ] NPath-def by auto
      moreover have  $T \ * \ \frown \ * \ (U \ * \ * \ [t])$ 
      proof –
        have  $Srcs\ T = Srcs\ (U \ * \ * \ [t])$ 
          using  $tT U Coinitial\ 1$ 
        by (metis  $Con-Arr-self\ Con-cons(2)\ Con-imp-eq-Srcs\ Con-sym\ Srcs-Resid-Arr-single$ 
           $T\ list.discI\ NPath-implies-Arr$ )
        hence  $NPath (T \ * \ * \ (U \ * \ * \ [t]))$ 
          using  $tT U Coinitial\ 1\ Con-sym\ ind$  [of  $Resid\ U\ [t]$ ] NPath-def
          by (metis  $Con-imp-Arr-Resid\ Srcs.elims\ T\ insert-subset\ list.simps(15)$ 
             $Arr.simps(3)$ )
        thus ?thesis
          using NPath-def by auto
      qed
    ultimately show ?thesis
      using  $Con-cons(1)$  [of  $T\ U\ t$ ] by fastforce
    qed
  ultimately show ?thesis
    using  $tT U T Coinitial Resid-cons(1)$  by auto
  qed
also have  $\dots \longleftrightarrow True$ 
proof (intro iffI, simp-all)
  have  $1: NPath ([t] \ * \ * \ U)$ 
    by (metis  $Coinitial NPath-Resid-single-Arr Srcs-simp_P U insert-subset$ 
       $list.sel(1)\ list.simps(15)\ NPath-def\ tT$ )

```

**moreover have**  $2: NPath (T \setminus^* (U \setminus^* [t]))$   
**by** (*metis* 0 *Arr.simps*(1) *Con-cons*(1) *Con-imp-eq-Srcs* *Con-implies-Arr*(1–2)  
*NPath-def* *T append-Nil2 calculation ind insert-subset list.simps*(15) *tT*)  
**moreover have**  $Trgs ([t] \setminus^* U) \subseteq Srcs (T \setminus^* (U \setminus^* [t]))$   
**by** (*metis* *Arr.simps*(1) *NPath-def Srcs-Resid* *Trgs-Resid-sym calculation*(2)  
*dual-order.refl*)  
**ultimately show**  $NPath ([t] \setminus^* U @ T \setminus^* (U \setminus^* [t]))$   
**using** *NPath-append* [of  $T \setminus^* (U \setminus^* [t]) [t] \setminus^* U$ ] **by** *fastforce*  
**qed**  
**finally show** *?thesis* **by** *blast*  
**qed**  
**qed**

**lemma** *Backward-stable-single*:  
**shows**  $\llbracket NPath U; NPath ([t] \setminus^* U) \rrbracket \Longrightarrow NPath [t]$   
**proof** (*induct* *U arbitrary*: *t*)  
**show**  $\bigwedge t. \llbracket NPath []; NPath ([t] \setminus^* []) \rrbracket \Longrightarrow NPath [t]$   
**using** *NPath-def* **by** *simp*  
**fix** *t u U*  
**assume** *ind*:  $\bigwedge t. \llbracket NPath U; NPath ([t] \setminus^* U) \rrbracket \Longrightarrow NPath [t]$   
**assume** *uU*:  $NPath (u \# U)$   
**assume** *resid*:  $NPath ([t] \setminus^* (u \# U))$   
**show**  $NPath [t]$   
**using** *uU ind NPath-def*  
**by** (*metis* *Arr.simps*(1) *Arr.simps*(2) *Con-implies-Arr*(2) *N.backward-stable*  
*N.elements-are-arr* *Resid-rec*(1) *Resid-rec*(3) *insert-subset list.simps*(15) *resid*)  
**qed**

**lemma** *Backward-stable*:  
**shows**  $\llbracket NPath U; NPath (T \setminus^* U) \rrbracket \Longrightarrow NPath T$   
**proof** (*induct* *T arbitrary*: *U*)  
**show**  $\bigwedge U. \llbracket NPath U; NPath ([] \setminus^* U) \rrbracket \Longrightarrow NPath []$   
**by** *simp*  
**fix** *t T U*  
**assume** *ind*:  $\bigwedge U. \llbracket NPath U; NPath (T \setminus^* U) \rrbracket \Longrightarrow NPath T$   
**assume** *U*:  $NPath U$   
**assume** *resid*:  $NPath ((t \# T) \setminus^* U)$   
**show**  $NPath (t \# T)$   
**proof** (*cases*  $T = []$ )  
**show**  $T = [] \Longrightarrow ?thesis$   
**using** *U resid Backward-stable-single* **by** *blast*  
**assume** *T*:  $T \neq []$   
**have**  $1: NPath ([t] \setminus^* U) \wedge NPath (T \setminus^* (U \setminus^* [t]))$   
**using** *T U NPath-append resid NPath-def*  
**by** (*metis* *Arr.simps*(1) *Con-cons*(1) *Resid-cons*(1))  
**have**  $2: t \in \mathfrak{N}$   
**using**  $1 U$  *Backward-stable-single NPath-def* **by** *simp*  
**moreover have**  $NPath T$   
**using**  $1 U$  *resid ind*

by (*metis 2 Arr.simps(2) Con-imp-eq-Srcs NPath-Resid N.elements-are-arr*)  
**moreover have**  $R.targets \subseteq Srcs\ T$   
 using *resid 1 Con-imp-eq-Srcs Con-sym Srcs-Resid-Arr-single NPath-def*  
 by (*metis Arr.simps(1) dual-order.eq-iff*)  
**ultimately show** *?thesis*  
 using *NPath-def*  
 by (*simp add: N.elements-are-arr*)  
**qed**  
**qed**

**sublocale** *normal-sub-rts Resid*  $\langle Collect\ NPath \rangle$   
 using *Ide-implies-NPath NPath-implies-Arr arr-char ide-char coinital-def*  
*sources-char<sub>P</sub> append-is-composite-of*  
**apply** *unfold-locales*  
**apply** *auto*  
 using *Backward-stable*  
 by *metis+*

**theorem** *normal-extends-to-paths:*  
**shows** *normal-sub-rts Resid*  $(Collect\ NPath)$   
 ..

**lemma** *Resid-NPath-preserves-reflects-Con:*  
**assumes** *NPath U and Srcs T = Srcs U*  
**shows**  $T \backslash^* U \frown^* T' \backslash^* U \longleftrightarrow T \frown^* T'$   
 using *assms NPath-def NPath-Resid con-char con-imp-coinital resid-along-elem-preserves-con*  
*Con-implies-Arr(2) Con-sym Cube(1)*  
 by (*metis Arr.simps(1) mem-Collect-eq*)

**notation** *Cong<sub>0</sub>* (**infix**  $\langle \approx^*_0 \rangle$  50)  
**notation** *Cong* (**infix**  $\langle \approx^* \rangle$  50)

**lemma** *Cong<sub>0</sub>-cancel-left<sub>CS</sub>:*  
**assumes**  $T @ U \approx^*_0 T @ U'$  **and**  $T \neq []$  **and**  $U \neq []$  **and**  $U' \neq []$   
**shows**  $U \approx^*_0 U'$   
 using *assms Cong<sub>0</sub>-cancel-left [of T U T @ U U' T @ U] Cong<sub>0</sub>-reflexive*  
*append-is-composite-of*  
 by (*metis Cong<sub>0</sub>-implies-Cong Cong-imp-arr(1) arr-append-imp-seq*)

**lemma** *Srcs-respects-Cong:*  
**assumes**  $T \approx^* T'$  **and**  $a \in Srcs\ T$  **and**  $a' \in Srcs\ T'$   
**shows**  $[a] \approx^* [a']$   
**proof** –  
**obtain**  $U\ U'$  **where**  $UU'$ :  $NPath\ U \wedge NPath\ U' \wedge T \backslash^* U \approx^*_0 T' \backslash^* U'$   
**using** *assms(1) by blast*  
**show** *?thesis*  
**proof**  
**show**  $U \in Collect\ NPath$

```

    using UU' by simp
  show U' ∈ Collect NPath
    using UU' by simp
  show [a] * \ * U ≈*0 [a'] * \ * U'
  proof -
    have NPath ([a] * \ * U) ∧ NPath ([a'] * \ * U')
      by (metis Arr.simps(1) Con-imp-eq-Srcs Con-implies-Arr(1) Con-single-ide-ind
          NPath-implies-Arr N.ide-closed R.in-sourcesE Srcs.simps(2) Srcs-simpp
          UU' assms(2-3) elements-are-arr not-arr-null null-char NPath-Resid-single-Arr)
    thus ?thesis
      using UU'
      by (metis Con-imp-eq-Srcs Cong0-imp-con NPath-Resid Srcs-Resid
          con-char NPath-implies-Arr mem-Collect-eq arr-resid-iff-con con-implies-arr(2))
  qed
qed
qed

```

**lemma** *Trgs-respects-Cong*:

**assumes**  $T \approx^* T'$  **and**  $b \in \text{Trgs } T$  **and**  $b' \in \text{Trgs } T'$

**shows**  $[b] \approx^* [b']$

**proof** -

**have**  $[b] \in \text{targets } T \wedge [b'] \in \text{targets } T'$

**proof** -

**have**  $1: \text{Ide } [b] \wedge \text{Ide } [b']$

**using** *assms*

**by** (metis Ball-Collect Trgs-are-ide Ide.simps(2))

**moreover have**  $\text{Srcs } [b] = \text{Trgs } T$

**using** *assms*

**by** (metis 1 Con-imp-Arr-Resid Con-imp-eq-Srcs Cong-imp-arr(1) Ide.simps(2)
 Srcs-Resid Con-single-ide-ind con-char arrE)

**moreover have**  $\text{Srcs } [b'] = \text{Trgs } T'$

**using** *assms*

**by** (metis Con-imp-Arr-Resid Con-imp-eq-Srcs Cong-imp-arr(2) Ide.simps(2)
 Srcs-Resid 1 Con-single-ide-ind con-char arrE)

**ultimately show** ?thesis

**unfolding** *targets-char<sub>P</sub>*

**using** *assms Cong-imp-arr(2) arr-char* **by** *blast*

**qed**

**thus** ?thesis

**using** *assms targets-char in-targets-respects-Cong* [of  $T T' [b] [b']$ ] **by** *simp*

**qed**

**lemma** *Cong<sub>0</sub>-append-resid-NPath*:

**assumes**  $\text{NPath } (T * \ * U)$

**shows**  $\text{Cong}_0 (T @ (U * \ * T)) U$

**proof** (*intro conjI*)

**show**  $0: (T @ U * \ * T) * \ * U \in \text{Collect NPath}$

**proof** -

**have**  $1: (T @ U * \ * T) * \ * U = T * \ * U @ (U * \ * T) * \ * (U * \ * T)$

```

    by (metis Arr.simps(1) NPath-implies-Arr assms Con-append(1) Con-implies-Arr(2)
        Con-sym Resid-append(1) con-imp-arr-resid null-char)
  moreover have NPath ...
    using assms
    by (metis 1 Arr-append-iffP NPath-append NPath-implies-Arr Ide-implies-NPath
        Nil-is-append-conv Resid-Arr-self arr-char con-char arr-resid-iff-con
        self-append-conv)
  ultimately show ?thesis by simp
qed
show U * \ $\backslash$  * (T @ U * \ $\backslash$  * T) ∈ Collect NPath
  using assms 0
  by (metis Arr.simps(1) Con-implies-Arr(2) Cong0-reflexive Resid-append(2)
      append.right-neutral arr-char Con-sym)
qed

```

end

```

locale paths-in-rts-with-coherent-normal =
  R: rts +
  N: coherent-normal-sub-rts +
  paths-in-rts
begin

```

```

  sublocale paths-in-rts-with-normal resid  $\mathfrak{N}$  ..

```

```

  notation Cong0 (infix <math>\approx^*_0</math> 50)
  notation Cong (infix <math>\approx^*</math> 50)

```

Since composites of normal transitions are assumed to exist, normal paths can be “folded” by composition down to single transitions.

**lemma** *NPath-folding*:

```

shows NPath U  $\implies$   $\exists u. u \in \mathfrak{N} \wedge R.sources\ u = Srcs\ U \wedge R.targets\ u = Trgs\ U \wedge$ 
  ( $\forall t. con\ [t]\ U \longrightarrow [t]\ *\ \backslash\ * U \approx^*_0 [t\ \backslash\ u]$ )

```

**proof** (*induct* U)

```

  show NPath []  $\implies$   $\exists u. u \in \mathfrak{N} \wedge R.sources\ u = Srcs\ [] \wedge R.targets\ u = Trgs\ [] \wedge$ 
  ( $\forall t. con\ [t]\ [] \longrightarrow [t]\ *\ \backslash\ * [] \approx^*_0 [t\ \backslash\ u]$ )

```

```

  using NPath-def by auto

```

```

  fix v U

```

```

  assume ind: NPath U  $\implies$   $\exists u. u \in \mathfrak{N} \wedge R.sources\ u = Srcs\ U \wedge R.targets\ u = Trgs\ U \wedge$ 
  ( $\forall t. con\ [t]\ U \longrightarrow [t]\ *\ \backslash\ * U \approx^*_0 [t\ \backslash\ u]$ )

```

```

  assume vU: NPath (v # U)

```

```

  show  $\exists vU. vU \in \mathfrak{N} \wedge R.sources\ vU = Srcs\ (v\ \# U) \wedge R.targets\ vU = Trgs\ (v\ \# U) \wedge$ 
  ( $\forall t. con\ [t]\ (v\ \# U) \longrightarrow [t]\ *\ \backslash\ * (v\ \# U) \approx^*_0 [t\ \backslash\ vU]$ )

```

**proof** (*cases* U = [])

```

  show U = []  $\implies$   $\exists vU. vU \in \mathfrak{N} \wedge R.sources\ vU = Srcs\ (v\ \# U) \wedge$ 
  R.targets vU = Trgs (v # U)  $\wedge$ 
  ( $\forall t. con\ [t]\ (v\ \# U) \longrightarrow [t]\ *\ \backslash\ * (v\ \# U) \approx^*_0 [t\ \backslash\ vU]$ )

```

```

  using vU Resid-rec(1) con-char

```

```

  by (metis Cong0-reflexive NPath-def Srcs.simps(2) Trgs.simps(2) arr-resid-iff-con)

```

```

    insert-subset list.simps(15))
assume  $U \neq []$ 
hence  $U: NPath\ U$ 
    using  $vU$  by (metis NPath-append append-Cons append-Nil)
obtain  $u$  where  $u: u \in \mathfrak{N} \wedge R.sources\ u = Srcs\ U \wedge R.targets\ u = Trgs\ U \wedge$ 
     $(\forall t. con\ [t]\ U \longrightarrow [t]^*\backslash^*\ U \approx^*_0 [t \setminus u])$ 
    using  $U\ ind$  by blast
have  $seq: R.seq\ v\ u$ 
proof
  show  $R.arr\ u$ 
    by (simp add: N.elements-are-arr u)
  show  $R.trg\ v \sim R.src\ u$ 
  proof –
    have  $R.targets\ v = R.sources\ u$ 
    by (metis (full-types) NPath-implies-Arr R.sources-resid Srcs.simps(2)
     $\langle U \neq [] \rangle$  Con-Arr-self Con-imp-eq-Srcs Con-initial-right Con-rec(2)
     $u\ vU$ )
    thus ?thesis
    using  $R.seqI(2)\ \langle R.arr\ u \rangle$  by blast
  qed
qed
obtain  $vu$  where  $vu: R.composite-of\ v\ u\ vu$ 
    using  $N.composite-closed-right\ seq\ u$  by presburger
have  $vu \in \mathfrak{N} \wedge R.sources\ vu = Srcs\ (v \# U) \wedge R.targets\ vu = Trgs\ (v \# U) \wedge$ 
     $(\forall t. con\ [t]\ (v \# U) \longrightarrow [t]^*\backslash^*\ (v \# U) \approx^*_0 [t \setminus vu])$ 
proof (intro conjI allI)
  show  $vu \in \mathfrak{N}$ 
    by (meson NPath-def N.composite-closed list.set-intros(1) subsetD u vU vu)
  show  $R.sources\ vu = Srcs\ (v \# U)$ 
    by (metis Con-imp-eq-Srcs Con-initial-right NPath-implies-Arr
     $R.sources-composite-of\ Srcs.simps(2)\ Arr-iff-Con-self\ vU\ vu$ )
  show  $R.targets\ vu = Trgs\ (v \# U)$ 
    by (metis  $R.targets-composite-of\ Trgs.simps(3)\ \langle U \neq [] \rangle$  list.exhaust-sel u vu)
  fix  $t$ 
  show  $con\ [t]\ (v \# U) \longrightarrow [t]^*\backslash^*\ (v \# U) \approx^*_0 [t \setminus vu]$ 
  proof (intro impI)
    assume  $t: con\ [t]\ (v \# U)$ 
    have  $1: [t]^*\backslash^*\ (v \# U) = [t \setminus v]^*\backslash^*\ U$ 
    using  $t\ Resid-rec(3)\ \langle U \neq [] \rangle$  con-char by force
    also have  $\dots \approx^*_0 [(t \setminus v) \setminus u]$ 
    using  $1\ t\ u$  by force
    also have  $[(t \setminus v) \setminus u] \approx^*_0 [t \setminus vu]$ 
    proof –
      have  $(t \setminus v) \setminus u \sim t \setminus vu$ 
      using  $vu\ R.resid-composite-of$ 
    by (metis (no-types, lifting)  $N.Cong_0-composite-of-arr-normal\ N.Cong_0-subst-right(1)$ 
     $\langle U \neq [] \rangle$  Con-rec(3) con-char R.con-sym t u)
    thus ?thesis
    using  $Ide.simps(2)\ R.prfx-implies-con\ Resid.simps(3)\ ide-char\ ide-closed$ 

```

by *presburger*  
 qed  
 finally show  $[t]^{**} (v \# U) \approx^*_0 [t \setminus vu]$  by *blast*  
 qed  
 qed  
 thus *?thesis* by *blast*  
 qed  
 qed

Coherence for single transitions extends inductively to paths.

**lemma** *Coherent-single*:

**assumes**  $R.arr\ t$  and  $NPath\ U$  and  $NPath\ U'$   
**and**  $R.sources\ t = Srcs\ U$  and  $Srcs\ U = Srcs\ U'$  and  $Trgs\ U = Trgs\ U'$   
**shows**  $[t]^{**} U \approx^*_0 [t]^{**} U'$

**proof** –

have 1:  $con\ [t]\ U \wedge con\ [t]\ U'$

using *assms*

by (*metis*  $Arr.simps(1-2)$   $Arr-iff-Con-self\ Resid-NPath-preserves-reflects-Con$   $Srcs.simps(2)$  *con-char*)

obtain  $u$  where  $u: u \in \mathfrak{N} \wedge R.sources\ u = Srcs\ U \wedge R.targets\ u = Trgs\ U \wedge$   
 $(\forall t. con\ [t]\ U \longrightarrow [t]^{**} U \approx^*_0 [t \setminus u])$

using *assms*  $NPath-folding$  by *metis*

obtain  $u'$  where  $u': u' \in \mathfrak{N} \wedge R.sources\ u' = Srcs\ U' \wedge R.targets\ u' = Trgs\ U' \wedge$   
 $(\forall t. con\ [t]\ U' \longrightarrow [t]^{**} U' \approx^*_0 [t \setminus u'])$

using *assms*  $NPath-folding$  by *metis*

have  $[t]^{**} U \approx^*_0 [t \setminus u]$

using  $u\ 1$  by *blast*

also have  $[t \setminus u] \approx^*_0 [t \setminus u']$

using *assms*( $1,4-6$ )  $N.Cong0-imp-con\ N.coherent\ u\ u'\ NPath-def$  by *simp*

also have  $[t \setminus u'] \approx^*_0 [t]^{**} U'$

using  $u'\ 1$  by *simp*

finally show *?thesis* by *simp*

qed

**lemma** *Coherent*:

**shows**  $\llbracket Arr\ T; NPath\ U; NPath\ U'; Srcs\ T = Srcs\ U;$

$Srcs\ U = Srcs\ U'; Trgs\ U = Trgs\ U' \rrbracket$

$\implies T^{**} U \approx^*_0 T^{**} U'$

**proof** (*induct*  $T$  *arbitrary*:  $U\ U'$ )

**show**  $\bigwedge U\ U'. \llbracket Arr\ []; NPath\ U; NPath\ U'; Srcs\ [] = Srcs\ U;$

$Srcs\ U = Srcs\ U'; Trgs\ U = Trgs\ U' \rrbracket$

$\implies []^{**} U \approx^*_0 []^{**} U'$

by (*simp* *add*: *arr-char*)

**fix**  $t\ T\ U\ U'$

**assume**  $tT: Arr\ (t \# T)$  and  $U: NPath\ U$  and  $U': NPath\ U'$

**and**  $Srcs1: Srcs\ (t \# T) = Srcs\ U$  and  $Srcs2: Srcs\ U = Srcs\ U'$

**and**  $Trgs: Trgs\ U = Trgs\ U'$

**and** *ind*:  $\bigwedge U\ U'. \llbracket Arr\ T; NPath\ U; NPath\ U'; Srcs\ T = Srcs\ U;$

$Srcs\ U = Srcs\ U'; Trgs\ U = Trgs\ U' \rrbracket$

$$\implies T * \setminus * U \approx^*_0 T * \setminus * U'$$

**have**  $t: R.arr\ t$   
**using**  $tT$  **by** (*metis Arr.simps(2) Con-Arr-self Con-rec(4) R.arrI*)  
**show**  $(t \# T) * \setminus * U \approx^*_0 (t \# T) * \setminus * U'$   
**proof** (*cases T = []*)  
**show**  $T = [] \implies ?thesis$   
**by** (*metis Srcs.simps(2) Srcs1 Srcs2 Trgs U U' Coherent-single Arr.simps(2) tT*)  
**assume**  $T: T \neq []$   
**let**  $?t = [t] * \setminus * U$  **and**  $?t' = [t] * \setminus * U'$   
**let**  $?T = T * \setminus * (U * \setminus * [t])$   
**let**  $?T' = T * \setminus * (U' * \setminus * [t])$   
**have**  $0: (t \# T) * \setminus * U = ?t @ ?T \wedge (t \# T) * \setminus * U' = ?t' @ ?T'$   
**using**  $tT\ U\ U'\ Srcs1\ Srcs2$   
**by** (*metis Arr-has-Src Arr-iff-Con-self Resid-cons(1) Srcs.simps(1) Resid-NPath-preserves-reflects-Con*)  
**have**  $1: ?t \approx^*_0 ?t'$   
**by** (*metis Srcs1 Srcs2 Srcs-simpP Trgs U U' list.sel(1) Coherent-single t tT*)  
**have**  $A: ?T * \setminus * (?t' * \setminus * ?t) = T * \setminus * ((U * \setminus * [t]) @ (?t' * \setminus * ?t))$   
**using**  $1\ Arr.simps(1)\ Con-append(2)\ Con-sym\ Resid-append(2)\ Con-implies-Arr(1)\ NPath-def$   
**by** (*metis arr-char elements-are-arr*)  
**have**  $B: ?T' * \setminus * (?t * \setminus * ?t') = T * \setminus * ((U' * \setminus * [t]) @ (?t * \setminus * ?t'))$   
**by** (*metis 1 Con-appendI(2) Con-sym Resid.simps(1) Resid-append(2) elements-are-arr not-arr-null null-char*)  
**have**  $E: ?T * \setminus * (?t' * \setminus * ?t) \approx^*_0 ?T' * \setminus * (?t * \setminus * ?t')$   
**proof** –  
**have**  $Arr\ T$   
**using**  $Arr.elims(1)\ T\ tT$  **by** *blast*  
**moreover** **have**  $NPath\ (U * \setminus * [t] @ ([t] * \setminus * U') * \setminus * ([t] * \setminus * U))$   
**using**  $1\ U\ t\ tT\ Srcs1\ Srcs-simpP$   
**apply** (*intro NPath-appendI*)  
**apply** *auto*  
**by** (*metis Arr.simps(1) NPath-def Srcs-Resid Trgs-Resid-sym*)  
**moreover** **have**  $NPath\ (U' * \setminus * [t] @ ([t] * \setminus * U) * \setminus * ([t] * \setminus * U'))$   
**using**  $t\ U'\ 1\ Con-imp-eq-Srcs\ Trgs-Resid-sym$   
**apply** (*intro NPath-appendI*)  
**apply** *auto*  
**apply** (*metis Arr.simps(2) NPath-Resid Resid.simps(1)*)  
**by** (*metis Arr.simps(1) NPath-def Srcs-Resid*)  
**moreover** **have**  $Srcs\ T = Srcs\ (U * \setminus * [t] @ ([t] * \setminus * U') * \setminus * ([t] * \setminus * U))$   
**using**  $A\ B$   
**by** (*metis (full-types) 0 1 Arr-has-Src Con-cons(1) Con-implies-Arr(1) Srcs.simps(1) Srcs-append T elements-are-arr not-arr-null null-char Con-imp-eq-Srcs*)  
**moreover** **have**  $Srcs\ (U * \setminus * [t] @ ([t] * \setminus * U') * \setminus * ([t] * \setminus * U)) =$   
 $Srcs\ (U' * \setminus * [t] @ ([t] * \setminus * U) * \setminus * ([t] * \setminus * U'))$   
**by** (*metis 1 Con-implies-Arr(2) Con-sym Cong0-imp-con Srcs-Resid Srcs-append arr-char con-char arr-resid-iff-con*)  
**moreover** **have**  $Trgs\ (U * \setminus * [t] @ ([t] * \setminus * U') * \setminus * ([t] * \setminus * U)) =$

$Trgs (U' * \setminus [t] @ ([t] * \setminus U) * \setminus ([t] * \setminus U'))$

using 1 *Cong<sub>0</sub>-imp-con con-char* by force

ultimately show *?thesis*

using *A B ind* [of  $(U * \setminus [t]) @ (?t' * \setminus ?t) (U' * \setminus [t]) @ (?t * \setminus ?t')$ ]

by *simp*

qed

have *C*: *NPath*  $((?T * \setminus (?t' * \setminus ?t)) * \setminus (?T' * \setminus (?t * \setminus ?t')))$

using *E* by *blast*

have *D*: *NPath*  $((?T' * \setminus (?t * \setminus ?t')) * \setminus (?T * \setminus (?t' * \setminus ?t)))$

using *E* by *blast*

show *?thesis*

proof –

have 2:  $((t \# T) * \setminus U) * \setminus ((t \# T) * \setminus U') =$   
 $((?t * \setminus ?t') * \setminus ?T') @ (((?T * \setminus (?t' * \setminus ?t)) * \setminus (?T' * \setminus (?t * \setminus ?t')))$

proof –

have  $((t \# T) * \setminus U) * \setminus ((t \# T) * \setminus U') = (?t @ ?T) * \setminus (?t' @ ?T')$

using 0 by *fastforce*

also have ... =  $((?t @ ?T) * \setminus ?t') * \setminus ?T'$

using *tT T U U' Srcs1 Srcs2 0*

by (*metis Con-appendI(2) Con-cons(1) Con-sym Resid.simps(1) Resid-append(2)*)

also have ... =  $((?t * \setminus ?t') @ (?T * \setminus (?t' * \setminus ?t))) * \setminus ?T'$

by (*metis (no-types, lifting) Arr.simps(1) Con-appendI(1) Con-implies-Arr(1)*  
*D NPath-def Resid-append(1) null-is-zero(2)*)

also have ... =  $((?t * \setminus ?t') * \setminus ?T') @$   
 $((?T * \setminus (?t' * \setminus ?t)) * \setminus (?T' * \setminus (?t * \setminus ?t')))$

proof –

have  $?t * \setminus ?t' @ ?T * \setminus (?t' * \setminus ?t) * \setminus ?T'$

using *C D E Con-sym*

by (*metis Con-append(2) Cong<sub>0</sub>-imp-con con-char arr-resid-iff-con*  
*con-implies-arr(2)*)

thus *?thesis*

using *Resid-append(1)*

by (*metis Con-sym append.right-neutral Resid.simps(1)*)

qed

finally show *?thesis* by *simp*

qed

moreover have 3: *NPath* ...

proof –

have *NPath*  $((?t * \setminus ?t') * \setminus ?T')$

using 0 1 *E*

by (*metis Con-imp-Arr-Resid Con-imp-eq-Srcs NPath-Resid Resid.simps(1)*  
*ex-un-null mem-Collect-eq*)

moreover have *Trgs*  $((?t * \setminus ?t') * \setminus ?T') =$   
 $Srcs ((?T * \setminus (?t' * \setminus ?t)) * \setminus (?T' * \setminus (?t * \setminus ?t')))$

using *C*

by (*metis NPath-implies-Arr Srcs.simps(1) Srcs-Resid*  
*Trgs-Resid-sym Arr-has-Src*)

ultimately show *?thesis*

using *C* by *blast*

**qed**  
**ultimately show**  $((t \# T) * \setminus U) * \setminus ((t \# T) * \setminus U') \in \text{Collect NPath}$   
**by** *simp*  
  
**have**  $4: ((t \# T) * \setminus U') * \setminus ((t \# T) * \setminus U) =$   
 $((?t' * \setminus ?t) * \setminus ?T) @ ((?T' * \setminus (?t * \setminus ?t')) * \setminus (?T * \setminus (?t' * \setminus ?t)))$   
**by** (*metis 0 2 3 Arr.simps(1) Con-implies-Arr(1) Con-sym D NPath-def Resid-append2*)  
**moreover have** *NPath ...*  
**proof** –  
**have** *NPath*  $((?t' * \setminus ?t) * \setminus ?T)$   
**by** (*metis 1 CollectD Cong<sub>0</sub>-imp-con E con-imp-coinitial forward-stable*  
*arr-resid-iff-con con-implies-arr(2)*)  
**moreover have** *NPath*  $((?T' * \setminus (?t * \setminus ?t')) * \setminus (?T * \setminus (?t' * \setminus ?t)))$   
**using** *U U' 1 D ind Coherent-single [of t U' U] by blast*  
**moreover have** *Trgs*  $((?t' * \setminus ?t) * \setminus ?T) =$   
 $\text{Srcs } ((?T' * \setminus (?t * \setminus ?t')) * \setminus (?T * \setminus (?t' * \setminus ?t)))$   
**by** (*metis Arr.simps(1) NPath-def Srcs-Resid Trgs-Resid-sym calculation(2)*)  
**ultimately show** *?thesis* **by** *blast*  
**qed**  
**ultimately show**  $((t \# T) * \setminus U') * \setminus ((t \# T) * \setminus U) \in \text{Collect NPath}$   
**by** *simp*  
**qed**  
**qed**  
**qed**

**sublocale** *rts-with-composites Resid*  
**using** *is-rts-with-composites* **by** *simp*

**sublocale** *coherent-normal-sub-rts Resid*  $\langle \text{Collect NPath} \rangle$

**proof**  
**fix** *T U U'*  
**assume** *T: arr T and U: U ∈ Collect NPath and U': U' ∈ Collect NPath*  
**assume** *sources-UU': sources U = sources U' and targets-UU': targets U = targets U'*  
**and** *TU: sources T = sources U*  
**have** *Srcs T = Srcs U*  
**using** *TU sources-char<sub>P</sub> T arr-iff-has-source* **by** *auto*  
**moreover have** *Srcs U = Srcs U'*  
**by** (*metis Con-imp-eq-Srcs T TU con-char con-imp-coinitial-ax con-sym in-sourcesE*  
*in-sourcesI arr-def sources-UU')*  
**moreover have** *Trgs U = Trgs U'*  
**using** *U U' targets-UU' targets-char*  
**by** (*metis (full-types) arr-iff-has-target composable-def composable-iff-seq*  
*composite-of-arr-target elements-are-arr equalsOI seq-char*)  
**ultimately show**  $T * \setminus U \approx^*_0 T * \setminus U'$   
**using** *T U U' Coherent [of T U U'] arr-char* **by** *blast*  
**qed**

**theorem** *coherent-normal-extends-to-paths:*  
**shows** *coherent-normal-sub-rts Resid*  $(\text{Collect NPath})$

..

**lemma** *Cong<sub>0</sub>-append-Arr-NPath*:  
**assumes**  $T \neq []$  **and** *Arr*  $(T @ U)$  **and** *NPath*  $U$   
**shows** *Cong<sub>0</sub>*  $(T @ U)$   $T$   
**using** *assms*  
**by** (*metis* *Arr.simps(1)* *Arr-appendE<sub>P</sub>* *NPath-implies-Arr* *append-is-composite-of arrI<sub>P</sub>*  
*arr-append-imp-seq* *composite-of-arr-normal* *mem-Collect-eq*)

**lemma** *Cong-append-NPath-Arr*:  
**assumes**  $T \neq []$  **and** *Arr*  $(U @ T)$  **and** *NPath*  $U$   
**shows**  $U @ T \approx^* T$   
**using** *assms*  
**by** (*metis* (*full-types*) *Arr.simps(1)* *Con-Arr-self* *Con-append(2)* *Con-implies-Arr(2)*  
*Con-imp-eq-Srcs* *composite-of-normal-arr* *Srcs-Resid* *append-is-composite-of arr-char*  
*NPath-implies-Arr* *mem-Collect-eq* *seq-char*)

## Permutation Congruence

Here we show that  $\approx^*$  coincides with “permutation congruence”: the least congruence respecting composition that relates  $[t, u \setminus t]$  and  $[u, t \setminus u]$  whenever  $t \frown u$  and that relates  $T @ [b]$  and  $T$  whenever  $b$  is an identity such that  $\text{seq } T [b]$ .

**inductive** *PCong*  
**where** *Arr*  $T \implies PCong\ T\ T$   
| *PCong*  $T\ U \implies PCong\ U\ T$   
|  $\llbracket PCong\ T\ U; PCong\ U\ V \rrbracket \implies PCong\ T\ V$   
|  $\llbracket \text{seq } T\ U; PCong\ T\ T'; PCong\ U\ U' \rrbracket \implies PCong\ (T @ U)\ (T' @ U')$   
|  $\llbracket \text{seq } T\ [b]; R.\text{ide } b \rrbracket \implies PCong\ (T @ [b])\ T$   
|  $t \frown u \implies PCong\ [t, u \setminus t]\ [u, t \setminus u]$

**lemmas** *PCong.intros(3)* [*trans*]

**lemma** *PCong-append-Ide*:  
**shows**  $\llbracket \text{seq } T\ B; Ide\ B \rrbracket \implies PCong\ (T @ B)\ T$   
**proof** (*induct*  $B$ )  
**show**  $\llbracket \text{seq } T\ []; Ide\ [] \rrbracket \implies PCong\ (T @ [])\ T$   
**by** *auto*  
**fix**  $b\ B\ T$   
**assume** *ind*:  $\llbracket \text{seq } T\ B; Ide\ B \rrbracket \implies PCong\ (T @ B)\ T$   
**assume** *seq*:  $\text{seq } T\ (b \# B)$   
**assume** *Ide*: *Ide*  $(b \# B)$   
**have**  $T @ (b \# B) = (T @ [b]) @ B$   
**by** *simp*  
**also have** *PCong* ...  $(T @ B)$   
**apply** (*cases*  $B = []$ )  
**using** *Ide* *PCong.intros(5)* *seq* **apply** *force*  
**using** *seq* *Ide* *PCong.intros(4)* [*of*  $T @ [b]$   $B\ T\ B$ ]  
**by** (*metis* *Arr.simps(1)* *Ide-imp-Ide-hd* *PCong.intros(1)* *PCong.intros(5)*  
*append-is-Nil-conv* *arr-append* *arr-append-imp-seq* *arr-char* *calculation*)

```

    list.distinct(1) list.sel(1) seq-char)
also have PCong (T @ B) T
proof (cases B = [])
  show B = []  $\implies$  ?thesis
    using PCong.intros(1) seq seq-char by force
  assume B: B  $\neq$  []
  have seq T B
    using B seq Ide
  by (metis Con-imp-eq-Srcs Ide-imp-Ide-hd Trgs-append <T @ b # B = (T @ [b]) @ B>
      append-is-Nil-conv arr-append arr-append-imp-seq arr-char cong-cons-ideI(2)
      list.distinct(1) list.sel(1) not-arr-null null-char seq-char ide-implies-arr)
  thus ?thesis
    using seq Ide ind
  by (metis Arr.simps(1) Ide.elims(3) Ide.simps(3) seq-char)
qed
finally show PCong (T @ (b # B)) T by blast
qed

```

```

lemma PCong-imp-Cong:
shows PCong T U  $\implies$  T  $\sim^*$  U
proof (induct rule: PCong.induct)
  show  $\bigwedge T. \text{Arr } T \implies T \sim^* T$ 
    using cong-reflexive by blast
  show  $\bigwedge T U. \llbracket \text{PCong } T U; T \sim^* U \rrbracket \implies U \sim^* T$ 
    by blast
  show  $\bigwedge T U V. \llbracket \text{PCong } T U; T \sim^* U; \text{PCong } U V; U \sim^* V \rrbracket \implies T \sim^* V$ 
    using cong-transitive by blast
  show  $\bigwedge T U U' T'. \llbracket \text{seq } T U; \text{PCong } T T'; T \sim^* T'; \text{PCong } U U'; U \sim^* U' \rrbracket$ 
     $\implies T @ U \sim^* T' @ U'$ 
    using cong-append by simp
  show  $\bigwedge T b. \llbracket \text{seq } T [b]; R.\text{ide } b \rrbracket \implies T @ [b] \sim^* T$ 
    using cong-append-ideI(4) ide-char by force
  show  $\bigwedge t u. t \frown u \implies [t, u \setminus t] \sim^* [u, t \setminus u]$ 
proof -
  have  $\bigwedge t u. t \frown u \implies [t, u \setminus t] \lesssim^* [u, t \setminus u]$ 
proof -
  fix t u
  assume con: t  $\frown$  u
  have  $([t] @ [u \setminus t]) \sim^* ([u] @ [t \setminus u]) =$ 
     $[(t \setminus u) \setminus (t \setminus u), ((u \setminus t) \setminus (u \setminus t)) \setminus ((t \setminus u) \setminus (t \setminus u))]$ 
    using con Resid-append2 [of [t] [u \setminus t] [u] [t \setminus u]]
  apply simp
  by (metis R.arr-resid-iff-con R.con-target R.conE R.con-sym
      R.prfx-implies-con R.prfx-reflexive R.cube)
moreover have Ide ...
  using con
  by (metis Arr.simps(2) Arr.simps(3) Ide.simps(2) Ide.simps(3) R.arr-resid-iff-con
      R.con-sym R.resid-ide-arr R.prfx-reflexive calculation Con-imp-Arr-Resid)
ultimately show  $[t, u \setminus t] \lesssim^* [u, t \setminus u]$ 

```

```

    using ide-char by auto
  qed
  thus  $\bigwedge t u. t \frown u \implies [t, u \setminus t] \sim^* [u, t \setminus u]$ 
    using R.con-sym by blast
  qed
qed

lemma PCong-permute-single:
shows  $[t] \frown^* U \implies PCong ([t] @ (U \setminus^* [t])) (U @ ([t] \setminus^* U))$ 
proof (induct U arbitrary: t)
  show  $\bigwedge t. [t] \frown^* [] \implies PCong ([t] @ [] \setminus^* [t]) ([] @ [t] \setminus^* [])$ 
    by auto
  fix t u U
  assume ind:  $\bigwedge t. [t] \setminus^* U \neq [] \implies PCong ([t] @ (U \setminus^* [t])) (U @ ([t] \setminus^* U))$ 
  assume con:  $[t] \frown^* u \neq U$ 
  show  $PCong ([t] @ (u \# U) \setminus^* [t]) ((u \# U) @ [t] \setminus^* (u \# U))$ 
  proof (cases  $U = []$ )
    show  $U = [] \implies ?thesis$ 
      by (metis PCong.intros(6) Resid.simps(3) append-Cons append-eq-append-conv2
        append-self-conv con-char con-def con con-sym-ax)
    assume  $U: U \neq []$ 
    show  $PCong ([t] @ ((u \# U) \setminus^* [t])) ((u \# U) @ ([t] \setminus^* (u \# U)))$ 
  proof -
    have  $[t] @ ((u \# U) \setminus^* [t]) = [t] @ ([u \setminus t] @ (U \setminus^* [t \setminus u]))$ 
      using Con-sym Resid-rec(2) U con by auto
    also have  $\dots = ([t] @ [u \setminus t]) @ (U \setminus^* [t \setminus u])$ 
      by auto
    also have  $PCong \dots (([u] @ [t \setminus u]) @ (U \setminus^* [t \setminus u]))$ 
  proof -
    have  $PCong ([t] @ [u \setminus t]) ([u] @ [t \setminus u])$ 
      using con
    by (simp add: Con-rec(3) PCong.intros(6) U)
    thus  $?thesis$ 
      by (metis Arr-Resid-single Con-implies-Arr(1) Con-rec(2) Con-sym
        PCong.intros(1,4) Srcs-Resid U append-is-Nil-conv append-is-composite-of
        arr-append-imp-seq arr-char calculation composite-of-unq-upto-cong
        con not-arr-null null-char ide-implies-arr seq-char)
  qed
  also have  $([u] @ [t \setminus u]) @ (U \setminus^* [t \setminus u]) = [u] @ ([t \setminus u] @ (U \setminus^* [t \setminus u]))$ 
    by simp
  also have  $PCong \dots ([u] @ (U @ ([t \setminus u] \setminus^* U)))$ 
  proof -
    have  $PCong ([t \setminus u] @ (U \setminus^* [t \setminus u])) (U @ ([t \setminus u] \setminus^* U))$ 
      using ind
    by (metis Resid-rec(3) U con)
    moreover have  $seq [u] ([t \setminus u] @ U \setminus^* [t \setminus u])$ 
  proof
    show  $Arr [u]$ 
      using Con-implies-Arr(2) Con-initial-right con by blast

```

**show**  $Arr ([t \setminus u] @ U * \setminus * [t \setminus u])$   
**using** *Con-implies-Arr(1) U con Con-imp-Arr-Resid Con-rec(3) Con-sym*  
**by** *fastforce*  
**show**  $Trgs [u] \cap Srcs ([t \setminus u] @ U * \setminus * [t \setminus u]) \neq \{\}$   
**by** (*metis Arr.simps(1) Arr.simps(2) Arr-has-Trg Con-implies-Arr(1)*  
*Int-absorb R.arr-resid-iff-con R.sources-resid Resid-rec(3)*  
*Srcs.simps(2) Srcs-append Trgs.simps(2) U <Arr [u]> con*)  
**qed**  
**moreover have**  $PCong [u] [u]$   
**using** *PCong.intros(1) calculation(2) seq-char* **by** *force*  
**ultimately show** *?thesis*  
**using** *U arr-append arr-char con seq-char*  
 $PCong.intros(4)$  [*of [u] [t \setminus u] @ (U \* \setminus \* [t \setminus u])*]  
 $[u] U @ ([t \setminus u] * \setminus * U)$   
**by** *blast*  
**qed**  
**also have**  $([u] @ (U @ ([t \setminus u] * \setminus * U))) = ((u \# U) @ [t] * \setminus * (u \# U))$   
**by** (*metis Resid-rec(3) U append-Cons append-Nil con*)  
**finally show** *?thesis* **by** *blast*  
**qed**  
**qed**  
**qed**

**lemma** *PCong-permute:*

**shows**  $T * \frown * U \implies PCong (T @ (U * \setminus * T)) (U @ (T * \setminus * U))$

**proof** (*induct T arbitrary: U*)

**show**  $\bigwedge U. [] * \setminus * U \neq [] \implies PCong ([] @ U * \setminus * []) (U @ [] * \setminus * U)$

**by** *simp*

**fix**  $t T U$

**assume** *ind:  $\bigwedge U. T * \frown * U \implies PCong (T @ (U * \setminus * T)) (U @ (T * \setminus * U))$*

**assume** *con:  $t \# T * \frown * U$*

**show**  $PCong ((t \# T) @ (U * \setminus * (t \# T))) (U @ ((t \# T) * \setminus * U))$

**proof** (*cases T = []*)

**assume**  $T: T = []$

**have**  $(t \# T) @ (U * \setminus * (t \# T)) = [t] @ (U * \setminus * [t])$

**using** *con T* **by** *simp*

**also have**  $PCong \dots (U @ ([t] * \setminus * U))$

**using** *PCong-permute-single T con* **by** *blast*

**finally show** *?thesis*

**using**  $T$  **by** *fastforce*

**next**

**assume**  $T: T \neq []$

**have**  $(t \# T) @ (U * \setminus * (t \# T)) = [t] @ (T @ (U * \setminus * (t \# T)))$

**by** *simp*

**also have**  $PCong \dots ([t] @ (U * \setminus * [t]) @ (T * \setminus * (U * \setminus * [t])))$

**using** *ind [of U \* \setminus \* [t]]*

**by** (*metis Arr.simps(1) Con-imp-Arr-Resid Con-implies-Arr(2) Con-sym*

*PCong.intros(1,4) Resid-cons(2) Srcs-Resid T arr-append arr-append-imp-seq*  
*calculation con not-arr-null null-char seq-char*)

```

also have  $[t] @ (U^* \setminus^* [t]) @ (T^* \setminus^* (U^* \setminus^* [t])) =$ 
            $([t] @ (U^* \setminus^* [t])) @ (T^* \setminus^* (U^* \setminus^* [t]))$ 
by simp
also have  $PCong (([t] @ (U^* \setminus^* [t])) @ (T^* \setminus^* (U^* \setminus^* [t])))$ 
            $((U @ ([t]^* \setminus^* U)) @ (T^* \setminus^* (U^* \setminus^* [t])))$ 
by (metis Arr.simps(1) Con-cons(1) Con-imp-Arr-Resid Con-implies-Arr(2)
      PCong.intros(1,4) PCong-permute-single Srcs-Resid T Trgs-append arr-append
      arr-char con seq-char)
also have  $(U @ ([t]^* \setminus^* U)) @ (T^* \setminus^* (U^* \setminus^* [t])) = U @ ((t \# T)^* \setminus^* U)$ 
by (metis Resid.simps(2) Resid-cons(1) append.assoc con)
finally show ?thesis by blast
qed
qed

```

```

lemma Cong-imp-PCong:
assumes  $T^* \sim^* U$ 
shows  $PCong T U$ 
proof –
  have  $PCong T (T @ (U^* \setminus^* T))$ 
    using assms PCong.intros(2) PCong-append-Ide
    by (metis Con-implies-Arr(1) Ide.simps(1) Srcs-Resid ide-char Con-imp-Arr-Resid
      seq-char)
  also have  $PCong (T @ (U^* \setminus^* T)) (U @ (T^* \setminus^* U))$ 
    using PCong-permute assms con-char prfx-implies-con by presburger
  also have  $PCong (U @ (T^* \setminus^* U)) U$ 
    using assms PCong-append-Ide
    by (metis Con-imp-Arr-Resid Con-implies-Arr(1) Srcs-Resid arr-resid-iff-con
      ide-implies-arr con-char ide-char seq-char)
  finally show ?thesis by blast
qed

```

```

lemma Cong-iff-PCong:
shows  $T^* \sim^* U \longleftrightarrow PCong T U$ 
using PCong-imp-Cong Cong-imp-PCong by blast

```

end

## 2.5 Composite Completion

The RTS of paths in an RTS factors via the coherent normal sub-RTS of identity paths into an extensional RTS with composites, which can be regarded as a “composite completion” of the original RTS.

```

locale composite-completion =
  R: rts R
for  $R :: 'a resid$ 
begin

  type-synonym  $'b arr = 'b list set$ 

```

**sublocale**  $N$ : *coherent-normal-sub-rts*  $R$   $\langle$ Collect  $R$ .*ide* $\rangle$   
**using**  $R$ .*rts-axioms*  $R$ .*identities-form-coherent-normal-sub-rts* **by** *auto*  
**sublocale**  $P$ : *paths-in-rts-with-coherent-normal*  $R$   $\langle$ Collect  $R$ .*ide* $\rangle$  ..  
**sublocale**  $Q$ : *quotient-by-coherent-normal*  $P$ .*Resid*  $\langle$ Collect  $P$ .*NPath* $\rangle$  ..

**notation**  $P$ .*Cong-class*  $\langle$ {-} $\rangle$

**definition** *resid* (**infix**  $\langle$ {\*\}\* $\rangle$  70)  
**where**  $resid \equiv Q$ .*Resid*

**sublocale** *extensional-rts resid*  
**unfolding** *resid-def*  
**using**  $Q$ .*extensional-rts-axioms* **by** *simp*

**notation** *con* (**infix**  $\langle$ {\*}\{\*\} $\rangle$  50)

**notation** *prfx* (**infix**  $\langle$ {\*}\lesssim{\*} $\rangle$  50)

**notation**  $P$ .*Resid* (**infix**  $\langle$ \*\{\*} $\rangle$  70)

**notation**  $P$ .*Con* (**infix**  $\langle$ \*\{\*\} $\rangle$  50)

**notation**  $P$ .*Cong* (**infix**  $\langle$ \*\approx{\*} $\rangle$  50)

**notation**  $P$ .*Cong<sub>0</sub>* (**infix**  $\langle$ \*\approx<sub>0</sub>\* $\rangle$  50)

**notation**  $P$ .*Cong-class*  $\langle$ {-} $\rangle$

**lemma** *P-ide-iff-NPath*:  
**shows**  $P$ .*ide*  $T \longleftrightarrow P$ .*NPath*  $T$   
**using**  $P$ .*NPath-def*  $P$ .*ide-char*  $P$ .*ide-closed* **by** *auto*

**lemma** *Cong-eq-Cong<sub>0</sub>*:  
**shows**  $T$  \*\approx\*  $T' \longleftrightarrow T$  \*\approx<sub>0</sub>\*  $T'$   
**by** (*metis*  $Collect$ -*cong*  $P$ .*Cong-iff-cong*  $P$ -*ide-iff-NPath* *mem-Collect-eq*)

**lemma** *Srcs-respects-Cong*:  
**assumes**  $T$  \*\approx\*  $T'$   
**shows**  $P$ .*Srcs*  $T = P$ .*Srcs*  $T'$   
**using** *assms*  
**by** (*meson*  $P$ .*Con-imp-eq-Srcs*  $P$ .*Cong<sub>0</sub>-imp-con*  $P$ .*con-char*  $Cong$ -*eq-Cong<sub>0</sub>*)

**lemma** *sources-respects-Cong*:  
**assumes**  $T$  \*\approx\*  $T'$   
**shows**  $P$ .*sources*  $T = P$ .*sources*  $T'$   
**using** *assms*  
**by** (*meson*  $P$ .*Cong<sub>0</sub>-imp-coinitial*  $Cong$ -*eq-Cong<sub>0</sub>*)

**lemma** *Trgs-respects-Cong*:  
**assumes**  $T$  \*\approx\*  $T'$   
**shows**  $P$ .*Trgs*  $T = P$ .*Trgs*  $T'$   
**by** (*metis*  $Cong$ -*eq-Cong<sub>0</sub>*  $P$ .*Srcs-Resid*  $P$ .*con-char*  $P$ .*cube*  $P$ .*ide-def*  
 $P$ -*ide-iff-NPath* *assms* *mem-Collect-eq*)

**lemma** *targets-respects-Cong*:

**assumes**  $T \approx^* T'$

**shows**  $P.\text{targets } T = P.\text{targets } T'$

**using** *assms*  $P.\text{Cong-imp-arr}(1)$   $P.\text{Cong-imp-arr}(2)$   $P.\text{arr-iff-has-target}$   
 $P.\text{targets-char}_P$  *Trgs-respects-Cong*

**by force**

**lemma** *ide-char<sub>CC</sub>*:

**shows**  $\text{ide } \mathcal{T} \longleftrightarrow \text{arr } \mathcal{T} \wedge (\forall T. T \in \mathcal{T} \longrightarrow P.\text{Ide } T)$

**by** (*metis*  $\text{Ball-Collect } P.\text{IdeI } P.\text{Ide-implies-NPath } Q.\text{ide-char}' P.\text{NPath-def}$   
 $\text{resid-def}$ )

**lemma** *con-char<sub>CC</sub>*:

**shows**  $\mathcal{T} \{\!\! \{^* \frown^* \}\!\!\} \mathcal{U} \longleftrightarrow \text{arr } \mathcal{T} \wedge \text{arr } \mathcal{U} \wedge P.\text{Cong-class-rep } \mathcal{T} \frown^* P.\text{Cong-class-rep } \mathcal{U}$

**proof**

**show**  $\text{arr } \mathcal{T} \wedge \text{arr } \mathcal{U} \wedge P.\text{Cong-class-rep } \mathcal{T} \frown^* P.\text{Cong-class-rep } \mathcal{U} \Longrightarrow \mathcal{T} \{\!\! \{^* \frown^* \}\!\!\} \mathcal{U}$   
**by** (*metis*  $P.\text{Cong-class-rep } P.\text{conI}_P Q.\text{arr-char } Q.\text{quot.preserves-con resid-def}$ )

**show**  $\mathcal{T} \{\!\! \{^* \frown^* \}\!\!\} \mathcal{U} \Longrightarrow \text{arr } \mathcal{T} \wedge \text{arr } \mathcal{U} \wedge P.\text{Cong-class-rep } \mathcal{T} \frown^* P.\text{Cong-class-rep } \mathcal{U}$

**proof** –

**assume**  $\text{con}: \mathcal{T} \{\!\! \{^* \frown^* \}\!\!\} \mathcal{U}$

**have**  $1: \text{arr } \mathcal{T} \wedge \text{arr } \mathcal{U}$

**using** *con coinitial-iff con-imp-coinitial* **by** *blast*

**moreover have**  $P.\text{Cong-class-rep } \mathcal{T} \frown^* P.\text{Cong-class-rep } \mathcal{U}$

**proof** –

**obtain**  $T U$  **where**  $TU: T \in \mathcal{T} \wedge U \in \mathcal{U} \wedge P.\text{Con } T U$

**using**  $P.\text{con-char } Q.\text{con-char}_{QCN}$  *con resid-def* **by** *auto*

**have**  $T \approx^* P.\text{Cong-class-rep } \mathcal{T} \wedge U \approx^* P.\text{Cong-class-rep } \mathcal{U}$

**using**  $TU 1 P.\text{Cong-class-memb-Cong-rep } Q.\text{arr-char resid-def}$  **by** *simp*

**thus** *?thesis*

**using**  $TU P.\text{Cong-subst}(1)$  [*of*  $T P.\text{Cong-class-rep } \mathcal{T} U P.\text{Cong-class-rep } \mathcal{U}$ ]

**by** (*metis*  $P.\text{coinitial-iff } P.\text{con-char } P.\text{con-imp-coinitial sources-respects-Cong}$ )

**qed**

**ultimately show** *?thesis* **by** *simp*

**qed**

**qed**

**lemma** *con-char<sub>CC</sub>'*:

**shows**  $\mathcal{T} \{\!\! \{^* \frown^* \}\!\!\} \mathcal{U} \longleftrightarrow \text{arr } \mathcal{T} \wedge \text{arr } \mathcal{U} \wedge (\forall T U. T \in \mathcal{T} \wedge U \in \mathcal{U} \longrightarrow T \frown^* U)$

**proof**

**show**  $\text{arr } \mathcal{T} \wedge \text{arr } \mathcal{U} \wedge (\forall T U. T \in \mathcal{T} \wedge U \in \mathcal{U} \longrightarrow T \frown^* U) \Longrightarrow \mathcal{T} \{\!\! \{^* \frown^* \}\!\!\} \mathcal{U}$

**using**  $\text{con-char}_{CC} P.\text{rep-in-Cong-class } Q.\text{arr-char resid-def}$  **by** *simp*

**show**  $\mathcal{T} \{\!\! \{^* \frown^* \}\!\!\} \mathcal{U} \Longrightarrow \text{arr } \mathcal{T} \wedge \text{arr } \mathcal{U} \wedge (\forall T U. T \in \mathcal{T} \wedge U \in \mathcal{U} \longrightarrow T \frown^* U)$

**proof** (*intro conjI allI impI*)

**assume**  $1: \mathcal{T} \{\!\! \{^* \frown^* \}\!\!\} \mathcal{U}$

**show**  $\text{arr } \mathcal{T}$

**using**  $1$  *con-implies-arr* **by** *simp*

**show**  $\text{arr } \mathcal{U}$

**using**  $1$  *con-implies-arr* **by** *simp*

```

fix T U
assume  $\mathcal{Q}$ :  $T \in \mathcal{T} \wedge U \in \mathcal{U}$ 
show  $T \text{ * } \frown \text{ * } U$ 
proof –
  have  $P.\text{Cong } T (P.\text{Cong-class-rep } \mathcal{T})$ 
    using  $\langle \text{arr } \mathcal{T} \rangle$ 
    by (simp add:  $\mathcal{Q} P.\text{Cong-class-memb-Cong-rep } Q.\text{arr-char resid-def}$ )
  moreover have  $P.\text{con } (P.\text{Cong-class-rep } \mathcal{T}) (P.\text{Cong-class-rep } \mathcal{U})$ 
    using 1 con-charCC by blast
  moreover have  $P.\text{Cong } (P.\text{Cong-class-rep } \mathcal{U}) U$ 
    using  $\langle \text{arr } \mathcal{U} \rangle$ 
    by (simp add:  $\mathcal{Q} P.\text{Cong-class-membs-are-Cong } P.\text{rep-in-Cong-class } Q.\text{arr-char resid-def}$ )
  ultimately show ?thesis
    by (meson Cong-eq-Cong0 P.Cong0-subst-Con P.con-char)
  qed
qed
qed

lemma resid-char:
shows  $\mathcal{T} \{ \text{*} \setminus \text{*} \} \mathcal{U} =$ 
  (if  $\mathcal{T} \{ \text{*} \frown \text{*} \} \mathcal{U}$  then  $\{ P.\text{Cong-class-rep } \mathcal{T} \text{ * } \setminus \text{ * } P.\text{Cong-class-rep } \mathcal{U} \}$  else  $\{ \}$ )
by (metis P.con-char P.rep-in-Cong-class Q.Resid-by-members Q.arr-char arr-resid-iff-con con-charCC Q.is-Cong-class-Resid resid-def)

lemma resid-simp:
assumes  $\mathcal{T} \{ \text{*} \frown \text{*} \} \mathcal{U}$  and  $T \in \mathcal{T}$  and  $U \in \mathcal{U}$ 
shows  $\mathcal{T} \{ \text{*} \setminus \text{*} \} \mathcal{U} = \{ P.\text{Resid } T U \}$ 
using assms resid-char
by (metis (no-types, lifting) P.con-char con-charCC' Q.Resid-by-members Q.con-charQCN resid-def)

lemma src-char':
shows  $\text{src } \mathcal{T} = \{ A. \text{arr } \mathcal{T} \wedge P.\text{Ide } A \wedge P.\text{Srcs } (P.\text{Cong-class-rep } \mathcal{T}) = P.\text{Srcs } A \}$ 
proof (cases arr  $\mathcal{T}$ )
  show  $\neg \text{arr } \mathcal{T} \implies \text{?thesis}$ 
    by (simp add: Q.null-char Q.src-def resid-def)
  assume  $\mathcal{T}$ :  $\text{arr } \mathcal{T}$ 
  have 1:  $\exists A. P.\text{Ide } A \wedge P.\text{Srcs } (P.\text{Cong-class-rep } \mathcal{T}) = P.\text{Srcs } A$ 
    by (metis P.Con-imp-eq-Srcs P.con-char P.con-imp-coinitial-ax P.ide-char  $\mathcal{T}$  con-charCC arrE)
  let  $?A = \text{SOME } A. P.\text{Ide } A \wedge P.\text{Srcs } (P.\text{Cong-class-rep } \mathcal{T}) = P.\text{Srcs } A$ 
  have  $A: P.\text{Ide } ?A \wedge P.\text{Srcs } (P.\text{Cong-class-rep } \mathcal{T}) = P.\text{Srcs } ?A$ 
    using 1 someI-ex [of  $\lambda A. P.\text{Ide } A \wedge P.\text{Srcs } (P.\text{Cong-class-rep } \mathcal{T}) = P.\text{Srcs } A$ ] by simp
  have  $a: \text{arr } \{ ?A \}$ 
    using  $A P.\text{ide-char } P.\text{is-Cong-classI } Q.\text{arr-char resid-def}$  by auto
  have ide-a:  $\text{ide } \{ ?A \}$ 
  using  $a P.\text{Cong-class-def } P.\text{normal-is-Cong-closed } P.\text{ide-iff-NPath } P.\text{ide-char ide-char}_{CC}$ 
by auto

```

**have**  $\text{sources } \mathcal{T} = \{\{?A\}\}$   
**proof** –  
**have**  $\mathcal{T} \{\{^* \frown^*\} \{?A\}\}$   
**by** (*metis* (*no-types*, *lifting*)  $A$   $P$ .*Cong-class-rep*  $P$ .*conI<sub>P</sub>*  $Q$ .*quot.preserves-con*  
 $\mathcal{T}$  *con-arr-self* *con-char<sub>CC</sub>*  $P$ .*Arr-iff-Con-self*  $P$ .*Con-IdeI*(2)  $Q$ .*arr-char resid-def*)  
**hence**  $\{?A\} \in \text{sources } \mathcal{T}$   
**using** *ide-a in-sourcesI* **by** *simp*  
**thus** *?thesis*  
**using** *sources-char<sub>WE</sub>* **by** *auto*  
**qed**  
**moreover** **have**  $\{?A\} = \{A. P.Ide A \wedge P.Srcs (P.Cong-class-rep \mathcal{T}) = P.Srcs A\}$   
**proof**  
**show**  $\{A. P.Ide A \wedge P.Srcs (P.Cong-class-rep \mathcal{T}) = P.Srcs A\} \subseteq \{?A\}$   
**using**  $A$   $P$ .*Cong-class-def*  $P$ .*Cong-closure-props*(3)  $P$ .*Ide-implies-Arr*  
 $P$ .*ide-closed*  $P$ .*ide-char*  
**by** *fastforce*  
**show**  $\{?A\} \subseteq \{A. P.Ide A \wedge P.Srcs (P.Cong-class-rep \mathcal{T}) = P.Srcs A\}$   
**using**  $a$   $P$ .*Cong-class-def* *Srcs-respects-Cong* *ide-a ide-char<sub>CC</sub>* **by** *blast*  
**qed**  
**ultimately show** *?thesis*  
**using**  $\mathcal{T}$  *src-in-sources* *sources-char<sub>WE</sub>* **by** *auto*  
**qed**

**lemma** *src-char*:  
**shows**  $\text{src } \mathcal{T} = \{A. \text{arr } \mathcal{T} \wedge P.Ide A \wedge (\forall T. T \in \mathcal{T} \longrightarrow P.Srcs T = P.Srcs A)\}$   
**proof** (*cases arr*  $\mathcal{T}$ )  
**show**  $\neg \text{arr } \mathcal{T} \Longrightarrow ?thesis$   
**using** *src-char'* **by** *simp*  
**assume**  $\mathcal{T}: \text{arr } \mathcal{T}$   
**have**  $\bigwedge T. T \in \mathcal{T} \Longrightarrow P.Srcs T = P.Srcs (P.Cong-class-rep \mathcal{T})$   
**using**  $\mathcal{T}$   $P$ .*Cong-class-memb-Cong-rep* *Srcs-respects-Cong*  $Q$ .*arr-char resid-def* **by** *auto*  
**thus** *?thesis*  
**using**  $\mathcal{T}$  *src-char'*  $P$ .*is-Cong-class-def*  $Q$ .*arr-char resid-def* **by** *force*  
**qed**

**lemma** *trg-char'*:  
**shows**  $\text{trg } \mathcal{T} = \{B. \text{arr } \mathcal{T} \wedge P.Ide B \wedge P.Trgs (P.Cong-class-rep \mathcal{T}) = P.Srcs B\}$   
**proof** (*cases arr*  $\mathcal{T}$ )  
**show**  $\neg \text{arr } \mathcal{T} \Longrightarrow ?thesis$   
**using** *resid-char resid-def*  $Q$ .*trg-char<sub>QN</sub>* **by** *auto*  
**assume**  $\mathcal{T}: \text{arr } \mathcal{T}$   
**have**  $1: \exists B. P.Ide B \wedge P.Trgs (P.Cong-class-rep \mathcal{T}) = P.Srcs B$   
**by** (*metis*  $P$ .*Con-implies-Arr*(2)  $P$ .*Resid-Arr-self*  $P$ .*Srcs-Resid*  $\mathcal{T}$  *con-char<sub>CC</sub>* *arrE*)  
**define**  $B$  **where**  $B = (\text{SOME } B. P.Ide B \wedge P.Trgs (P.Cong-class-rep \mathcal{T}) = P.Srcs B)$   
**have**  $B: P.Ide B \wedge P.Trgs (P.Cong-class-rep \mathcal{T}) = P.Srcs B$   
**unfolding**  $B$ -*def*  
**using**  $1$  *someI-ex* [*of*  $\lambda B. P.Ide B \wedge P.Trgs (P.Cong-class-rep \mathcal{T}) = P.Srcs B$ ] **by** *simp*  
**hence**  $2: P.Ide B \wedge P.Con (P.Resid (P.Cong-class-rep \mathcal{T}) (P.Cong-class-rep \mathcal{T})) B$   
**using**  $\mathcal{T}$

**by** (*metis* (*no-types*, *lifting*) *P.Con-Ide-iff* *P.Ide-implies-Arr* *P.Resid-Arr-self*  
*P.Srcs-Resid* *arrE* *P.Con-implies-Arr*(2) *con-char<sub>CC</sub>*)

**have** *b*: *arr*  $\{\!|B|\!\}$

**by** (*simp* *add*: 2 *P.ide-char* *P.is-Cong-classI* *Q.arr-char* *resid-def*)

**have** *ide-b*: *ide*  $\{\!|B|\!\}$

**by** (*simp* *add*: 2 *P.ide-char* *resid-def*)

**have** *targets*  $\mathcal{T} = \{\!|B|\!\}$

**proof** –

**have** *cong* ( $\mathcal{T}$   $\{\!|\ast\!\}$   $\mathcal{T}$ )  $\{\!|B|\!\}$

**by** (*metis* 2 *P.con-char* *Q.ide-imp-con-iff-cong* *Q.quot.preserves-con*  
*T* *composite-completion.resid-char* *composite-completion-axioms*  
*cong-reflexive* *ide-b* *resid-def*)

**thus** *?thesis*

**using**  $\mathcal{T}$  *Q.targets-char<sub>QCN</sub>* [*of*  $\mathcal{T}$ ] *cong-char* *resid-def* **by** *auto*

**qed**

**moreover** **have**  $\{\!|B|\!\} = \{B. P.Ide B \wedge P.Trgs (P.Cong-class-rep \mathcal{T}) = P.Srcs B\}$

**proof**

**show**  $\{B. P.Ide B \wedge P.Trgs (P.Cong-class-rep \mathcal{T}) = P.Srcs B\} \subseteq \{\!|B|\!\}$

**using** *B* *P.Cong-class-def* *P.Cong-closure-props*(3) *P.Ide-implies-Arr*  
*P.ide-closed* *P.ide-char*

**by** *force*

**show**  $\{\!|B|\!\} \subseteq \{B. P.Ide B \wedge P.Trgs (P.Cong-class-rep \mathcal{T}) = P.Srcs B\}$

**proof** –

**have**  $\bigwedge B'. P.Cong B' B \implies P.Ide B' \wedge P.Trgs (P.Cong-class-rep \mathcal{T}) = P.Srcs B'$

**using** *B* *P.ide-iff-NPath* *P.normal-is-Cong-closed* *Srcs-respects-Cong*

**by** (*metis* *P.Cong-closure-props*(1) *P.ide-char* *mem-Collect-eq*)

**thus** *?thesis*

**using** *P.Cong-class-def* **by** *blast*

**qed**

**qed**

**ultimately** **show** *?thesis*

**using**  $\mathcal{T}$  *trg-in-targets* *targets-char* **by** *auto*

**qed**

**lemma** *trg-char*:

**shows** *trg*  $\mathcal{T} = \{B. arr \mathcal{T} \wedge P.Ide B \wedge (\forall T. T \in \mathcal{T} \implies P.Trgs T = P.Srcs B)\}$

**proof** (*cases* *arr*  $\mathcal{T}$ )

**show**  $\neg arr \mathcal{T} \implies ?thesis$

**using** *trg-char'* **by** *presburger*

**assume**  $\mathcal{T}$ : *arr*  $\mathcal{T}$

**have**  $\bigwedge T. T \in \mathcal{T} \implies P.Trgs T = P.Trgs (P.Cong-class-rep \mathcal{T})$

**using**  $\mathcal{T}$

**by** (*metis* *P.Cong-class-memb-Cong-rep* *Trgs-respects-Cong* *Q.arr-char* *resid-def*)

**thus** *?thesis*

**using**  $\mathcal{T}$  *trg-char'* *P.is-Cong-class-def* *Q.arr-char* *resid-def* **by** *force*

**qed**

**lemma** *prfx-char*:

**shows**  $\mathcal{T} \{\!|\lesssim\!\} \mathcal{U} \iff arr \mathcal{T} \wedge arr \mathcal{U} \wedge (\forall T U. T \in \mathcal{T} \wedge U \in \mathcal{U} \implies P.prfx T U)$

**proof**

**show**  $\mathcal{T} \{\!\{^* \lesssim^* \}\!\} \mathcal{U} \implies \text{arr } \mathcal{T} \wedge \text{arr } \mathcal{U} \wedge (\forall T U. T \in \mathcal{T} \wedge U \in \mathcal{U} \longrightarrow P.\text{prfx } T U)$   
**by** (*metis* (*mono-tags, lifting*) *Ball-Collect* *P.arr-in-Cong-class*  
*P.arr-resid-iff-con* *P.conI<sub>P</sub>* *P.ide-iff-NPath* *Q.ide-char'*  
*con-char<sub>CC'</sub>* *prfx-implies-con* *resid-def* *resid-simp*)  
**show**  $\text{arr } \mathcal{T} \wedge \text{arr } \mathcal{U} \wedge (\forall T U. T \in \mathcal{T} \wedge U \in \mathcal{U} \longrightarrow P.\text{prfx } T U) \implies \mathcal{T} \{\!\{^* \lesssim^* \}\!\} \mathcal{U}$   
**by** (*metis* *P.Cong-class-rep* *P.rep-in-Cong-class* *Q.arr-char* *Q.quot.preserves-prfx*  
*resid-def*)

**qed**

**lemma** *quotient-reflects-con*:

**assumes** *con* (*Q.quot t*) (*Q.quot u*)

**shows** *P.con t u*

**using** *assms* *P.arr-in-Cong-class* *P.con-char* *con-char<sub>CC'</sub>* *resid-def* **by** *force*

**lemma** *is-extensional-rts-with-composites*:

**shows** *extensional-rts-with-composites resid*

**proof**

**fix**  $\mathcal{T} \mathcal{U}$

**assume** *seq*: *seq*  $\mathcal{T} \mathcal{U}$

**obtain**  $T$  **where**  $T: \mathcal{T} = \{\!\{T\}\!\}$

**by** (*metis* *P.Cong-class-rep* *Q.arr-char* *Q.seqE<sub>WE</sub>* *resid-def* *seq*)

**obtain**  $U$  **where**  $U: \mathcal{U} = \{\!\{U\}\!\}$

**by** (*metis* *P.Cong-class-rep* *Q.arr-char* *Q.seqE<sub>WE</sub>* *resid-def* *seq*)

**have**  $1: P.\text{Arr } T \wedge P.\text{Arr } U$

**using** *P.arr-char* *T U* *resid-def* *seq* **by** *auto*

**have**  $2: P.\text{Trgs } T = P.\text{Srcs } U$

**proof** –

**have**  $P.\text{Trgs } (P.\text{Cong-class-rep } \mathcal{T}) = P.\text{Srcs } (P.\text{Cong-class-rep } \mathcal{U})$

**by** (*metis* (*mono-tags, lifting*) *P.Con-imp-eq-Srcs* *P.rep-in-Cong-class*  
*Q.arr-char* *con-arr-src(2)* *con-char<sub>CC'</sub>* *mem-Collect-eq* *resid-def*  
*seq* *seqE<sub>WE</sub>* *trg-char'*)

**thus** *?thesis*

**by** (*metis*  $1$  *P.Cong-class-memb-Cong-rep* *P.arrI<sub>P</sub>* *P.arr-in-Cong-class*  
*P.is-Cong-classI* *Srcs-respects-Cong* *T* *Trgs-respects-Cong* *U*)

**qed**

**have**  $P.\text{Arr } (T @ U)$

**using**  $1$   $2$  **by** *simp*

**moreover** **have**  $P.\text{Ide } (T^* \setminus^* (T @ U))$

**by** (*metis*  $1$  *P.Con-append(2)* *P.Con-sym* *P.Resid-Arr-self* *P.Resid-Ide-Arr-ind*  
*P.Resid-append(2)* *P.Trgs.simps(1)* *calculation* *P.Arr-has-Trg*)

**moreover** **have**  $(T @ U)^* \setminus^* T^* \approx^* U$

**by** (*metis*  $1$  *P.Arr.simps(1)* *P.Con-implies-Arr(2)* *P.Cong<sub>0</sub>-implies-Cong*  
*P.con-sym-ax* *P.null-char* *calculation(2)* *P.Cong<sub>0</sub>-append-resid-NPath*  
*P.Cong<sub>0</sub>-cancel-left<sub>CS</sub>* *P.Ide-implies-NPath*)

**ultimately** **have** *composite-of*  $\mathcal{T} \mathcal{U} \{\!\{T @ U\}\!\}$

**by** (*metis*  $1$  *P.Arr.simps(1)* *P.append-is-composite-of* *P.arrI<sub>P</sub>* *P.arr-append-imp-seq*  
*Q.quot.preserves-composites* *T U* *resid-def*)

**thus** *composable*  $\mathcal{T} \mathcal{U}$

**using** *composable-def* **by** *auto*  
**qed**

**sublocale** *extensional-rts-with-composites resid*  
**using** *is-extensional-rts-with-composites* **by** *simp*

**notation** *comp* (**infixr**  $\{\{*\cdot*\}\}$  55)

## 2.5.1 Inclusion Map

**definition** *incl*  
**where**  $incl \equiv Q.quot \circ P.incl$

**sublocale** *incl: simulation R resid incl*  
**unfolding** *incl-def resid-def*  
**using** *P.incl-is-simulation Q.quotient-is-simulation composite-simulation.intro*  
**by** *blast*  
**sublocale** *incl: simulation-to-extensional-rts R resid incl ..*

**lemma** *incl-is-simulation:*  
**shows** *simulation R resid incl*  
**..**

**lemma** *incl-simp* [*simp*]:  
**shows**  $incl\ t = \{\{t\}\}$   
**by** (*metis (mono-tags, lifting) P.Arr.simps(2) P.Arr-iff-Con-self P.Ide.simps(1)*  
*P.cong-reflexive P.ide-char Q.quot.extensionality incl.extensionality incl-def*  
*o-apply resid-def*)

**lemma** *incl-reflects-con:*  
**assumes**  $incl\ t\ \{\{*\curvearrowright*\}\}\ incl\ u$   
**shows**  $R.con\ t\ u$   
**by** (*metis P.Con-rec(1) P.arr-in-Cong-class assms con-char<sub>CC</sub>' incl-simp*  
*Q.quot.preserves-reflects-arr resid-def*)

**lemma** *cong-iff-eq-incl:*  
**assumes**  $R.arr\ t$  **and**  $R.arr\ u$   
**shows**  $incl\ t = incl\ u \longleftrightarrow R.cong\ t\ u$   
**by** (*metis P.Ide.simps(2) P.arr-in-Cong-class assms(1) incl-reflects-con*  
*conE cong-char ide-char<sub>CC</sub> incl.preserves-prfx incl.preserves-reflects-arr*  
*incl.preserves-resid incl-simp prfx-implies-con Q.quot.extensionality resid-def*)

**lemma** *incl-cancel-left:*  
**assumes** *transformation X R F G T* **and** *transformation X R F' G' T'*  
**and** *extensional-rts R*  
**and**  $incl \circ T = incl \circ T'$   
**shows**  $T = T'$   
**proof** –  
**interpret**  $R: extensional-rts\ R$

```

    using assms(3) by blast
  interpret T: transformation X R F G T
    using assms(1) by blast
  interpret T': transformation X R F' G' T'
    using assms(2) by blast
  show T = T'
  proof
    fix x
    show T x = T' x
    by (metis R.cong-char T'.extensionality T.A.cong-reflexive T.extensionality
       T.respects-cong assms(4) comp-apply cong-iff-eq-incl incl.preserves-reflects-arr)
  qed
qed

```

The inclusion is surjective on identities.

```

lemma img-incl-ide:
shows incl ' (Collect R.ide) = Collect ide
proof
  show incl ' Collect R.ide ⊆ Collect ide
    using incl.preserves-ide by force
  show Collect ide ⊆ incl ' Collect R.ide
  proof
    fix A
    assume A: A ∈ Collect ide
    obtain A where A: A ∈ A
      using A Q.ide-char resid-def by auto
    have P.NPath A
      by (metis A Ball-Collect A Q.ide-char' mem-Collect-eq resid-def)
    obtain a where a: a ∈ P.Srcs A
      using ⟨P.NPath A⟩
      by (meson P.NPath-implies-Arr equals0I P.Arr-has-Src)
    have A = {[a]}
  proof -
    have A *≈0* [a]
      by (metis P.Con-Arr-self P.Con-imp-eq-Srcs P.Con-implies-Arr(1)
         P.Con-sym P.NPath-implies-Arr P.Resid-Arr-Src P.conIP P.ideI
         P.ide-closed P.resid-ide-arr ⟨P.NPath A⟩ a mem-Collect-eq)
    thus ?thesis
      by (metis A CollectD P.Cong0-imp-con P.Cong0-subst-left(1)
         P.Cong-class-eqI P.Cong-closure-props(3) P.resid-arr-ide
         P.ide-iff-NPath Q.ideE A ⟨P.NPath A⟩ resid-def resid-simp)
  qed
  thus A ∈ incl ' Collect R.ide
    using P.Srcs-are-ide a by auto
  qed
qed
end

```

## 2.5.2 Composite Completion of a Weakly Extensional RTS

```

locale composite-completion-of-weakly-extensional-rts =
  R: weakly-extensional-rts R +
  composite-completion
begin

  sublocale P: paths-in-weakly-extensional-rts R ..
  sublocale incl: simulation-between-weakly-extensional-rts R resid incl ..

  notation comp (infixr  $\{\ast.\ast\}$  55)

  lemma src-charCCWE:
  shows src  $\mathcal{T} =$  (if arr  $\mathcal{T}$  then incl (P.Src (P.Cong-class-rep  $\mathcal{T}$ )) else null)
  proof (cases arr  $\mathcal{T}$ )
    show  $\neg$  arr  $\mathcal{T} \implies$  ?thesis
      using src-def by auto
    assume  $\mathcal{T}$ : arr  $\mathcal{T}$ 
    have src  $\mathcal{T} = \{A. P.Ide A \wedge P.Srcs (P.Cong-class-rep \mathcal{T}) = P.Srcs A\}$ 
      using  $\mathcal{T}$  src-char' [of  $\mathcal{T}$ ] by simp
    moreover have 1:  $\bigwedge A. P.Ide A \implies$ 
      P.Srcs (P.Cong-class-rep  $\mathcal{T}) = P.Srcs A \longleftrightarrow$ 
      P.Src (P.Cong-class-rep  $\mathcal{T}) = P.Src A$ 
      by (metis  $\mathcal{T}$  P.Con-implies-Arr(2) P.Ide-implies-Arr con-arr-self con-charCC
        insertCI P.Srcs-simpPWE singletonD)
    ultimately have 2: src  $\mathcal{T} = \{A. P.Ide A \wedge P.Src (P.Cong-class-rep \mathcal{T}) = P.Src A\}$ 
      by blast
    also have ... = incl (P.Src (P.Cong-class-rep  $\mathcal{T}$ ))
    proof -
      have incl (P.Src (P.Cong-class-rep  $\mathcal{T}$ )) = Q.quot [R.src (hd (P.Cong-class-rep  $\mathcal{T}$ ))]
        by auto
      also have ... =  $\{A. P.Ide A \wedge P.Src (P.Cong-class-rep \mathcal{T}) = P.Src A\}$ 
        apply auto[1]
        apply (metis Q.null-char R.ide-iff-src-self R.src-src empty-iff ide-charCC
          incl.extensionality incl.preserves-ide incl-simp resid-def)
        apply (metis 1 CollectD P.Cong-class-def P.Ide.simps(2)
          P.paths-in-weakly-extensional-rts-axioms Q.null-char R.rts-axioms R.src-trg
          R.trg-src Srcs-respects-Cong calculation composite-completion.ide-charCC
          composite-completion-axioms empty-iff incl.extensionality incl.preserves-ide
          list.sel(1) paths-in-weakly-extensional-rts.Src.elims resid-def rts.ide-src)
        by (metis (mono-tags, lifting) 2 P.Cong-class-def P.Ide.simps(2)
          P.Ide-imp-Ide-hd P.Src.elims P.is-Cong-classE Q.arr-char Q.src-src R.src-ide
          list.sel(1) mem-Collect-eq resid-def src-char')
      finally show ?thesis by blast
    qed
    finally show ?thesis
      using  $\mathcal{T}$  by auto
  qed

  lemma trg-charCCWE:

```

**shows**  $\text{trg } \mathcal{T} = (\text{if arr } \mathcal{T} \text{ then incl } (P.\text{Trg } (P.\text{Cong-class-rep } \mathcal{T})) \text{ else null})$   
**proof** (*cases arr*  $\mathcal{T}$ )  
**show**  $\neg \text{arr } \mathcal{T} \implies ?thesis$   
**using** *trg-def* **by** *auto*  
**assume**  $\mathcal{T}: \text{arr } \mathcal{T}$   
**have**  $\text{trg } \mathcal{T} = \{A. P.\text{Ide } A \wedge P.\text{Trg } (P.\text{Cong-class-rep } \mathcal{T}) = P.\text{Src } A\}$   
**proof** –  
**have**  $\text{trg } \mathcal{T} = \{B. P.\text{Ide } B \wedge P.\text{Trgs } (P.\text{Cong-class-rep } \mathcal{T}) = P.\text{Srcs } B\}$   
**using**  $\mathcal{T}$  *trg-char'* [*of*  $\mathcal{T}$ ] **by** *simp*  
**moreover have**  $\bigwedge B. P.\text{Ide } B \implies$   
 $P.\text{Trgs } (P.\text{Cong-class-rep } \mathcal{T}) = P.\text{Srcs } B \longleftrightarrow$   
 $P.\text{Trg } (P.\text{Cong-class-rep } \mathcal{T}) = P.\text{Src } B$   
**by** (*metis* *P.Con-implies-Arr(1)* *P.Ide-implies-Arr*  $\mathcal{T}$  *con-arr-self con-char<sub>CC</sub>*  
*P.Srcs-simp<sub>PWE</sub>* *P.Trgs-simp<sub>PWE</sub>* *singleton-inject*)  
**ultimately show** *?thesis* **by** *blast*  
**qed**  
**also have**  $\dots = \text{incl } (P.\text{Trg } (P.\text{Cong-class-rep } \mathcal{T}))$   
**using** *incl.preserves-trg*  
**by** (*metis* (*mono-tags, lifting*) *P.rep-in-Cong-class*  $\mathcal{T}$  *arr-trg-iff-arr*  
*calculation mem-Collect-eq Q.arr-char resid-def src-char<sub>CCWE</sub> src-trg*)  
**finally show** *?thesis*  
**using**  $\mathcal{T}$  **by** *auto*  
**qed**

When applied to a weakly extensional RTS, the composite completion construction does not identify any states that are distinct in the original RTS.

**lemma** *incl-injective-on-ide*:  
**shows** *inj-on incl* (*Collect R.ide*)  
**by** (*metis* *R.con-ide-are-eq ideE incl.preserves-ide incl-reflects-con inj-onCI*  
*mem-Collect-eq*)

When applied to a weakly extensional RTS, the composite completion construction is a bijection between the states of the original RTS and the states of its completion.

**lemma** *incl-bijective-on-ide*:  
**shows**  $\text{incl} \in \text{Collect } R.\text{ide} \rightarrow \text{Collect } \text{ide}$   
**and**  $(\lambda A. P.\text{Src } (P.\text{Cong-class-rep } A)) \in \text{Collect } \text{ide} \rightarrow \text{Collect } R.\text{ide}$   
**and**  $\bigwedge a. R.\text{ide } a \implies (\lambda A. P.\text{Src } (P.\text{Cong-class-rep } A)) (\text{incl } a) = a$   
**and**  $\bigwedge A. \text{ide } A \implies \text{incl } ((\lambda A. P.\text{Src } (P.\text{Cong-class-rep } A)) A) = A$   
**and** *bij-betw incl* (*Collect R.ide*) (*Collect ide*)  
**and** *bij-betw*  $(\lambda A. P.\text{Src } (P.\text{Cong-class-rep } A))$  (*Collect ide*) (*Collect R.ide*)  
**proof** –  
**show** 1:  $\text{incl} \in \text{Collect } R.\text{ide} \rightarrow \text{Collect } \text{ide}$   
**using** *img-incl-ide* **by** *auto*  
**show** 2:  $(\lambda A. P.\text{Src } (P.\text{Cong-class-rep } A)) \in \text{Collect } \text{ide} \rightarrow \text{Collect } R.\text{ide}$   
**by** (*metis* (*no-types, lifting*) *P.Src.elims Pi-I'* *R.ide-iff-trg-self*  
*R.trg-src ide-implies-arr incl.preserves-reflects-arr mem-Collect-eq*  
*src-char<sub>CCWE</sub> src-ide*)  
**show** 3:  $\bigwedge a. R.\text{ide } a \implies (\lambda A. P.\text{Src } (P.\text{Cong-class-rep } A)) (\text{incl } a) = a$   
**by** (*metis* *R.ide-backward-stable R.weak-extensionality cong-iff-eq-incl*)

```

    incl.preserves-ide incl.preserves-reflects-arr ide-implies-arr
    src-charCCWE src-ide)
  show  $\lambda A. \text{ide } A \implies \text{incl } ((\lambda A. P.\text{Src } (P.\text{Cong-class-rep } A)) A) = A$ 
    using src-charCCWE src-ide by auto
  show bij-betw incl (Collect R.ide) (Collect ide)
    using incl-injective-on-ide img-incl-ide bij-betw-def by blast
  show bij-betw ( $\lambda A. P.\text{Src } (P.\text{Cong-class-rep } A)$ ) (Collect ide) (Collect R.ide)
    using 1 2 3 4 bij-betwI by force
qed

```

end

### 2.5.3 Composite Completion of an Extensional RTS

```

locale composite-completion-of-extensional-rts =
  R: extensional-rts R +
  composite-completion
begin

  sublocale composite-completion-of-weakly-extensional-rts ..
  sublocale incl: simulation-between-extensional-rts R resid incl ..

end

```

### 2.5.4 Freeness of Composite Completion

In this section we show that the composite completion construction is free: any simulation from RTS  $A$  to an extensional RTS with composites  $B$  extends uniquely to a simulation on the composite completion of  $A$ .

```

type-synonym 'a comp = 'a  $\Rightarrow$  'a  $\Rightarrow$  'a

locale rts-with-chosen-composites =
  rts +
fixes comp :: 'a comp (infixr · 55)
assumes comp-extensionality-ax:  $\bigwedge t u :: 'a. \neg \text{seq } t u \implies t \cdot u = \text{null}$ 
and composite-of-comp-ax:  $\bigwedge t u v :: 'a. \text{seq } t u \implies \text{composite-of } t u (t \cdot u)$ 
and comp-assoc-ax:  $\bigwedge t u v :: 'a. \llbracket \text{seq } t u; \text{seq } u v \rrbracket \implies (t \cdot u) \cdot v = t \cdot (u \cdot v)$ 
and resid-comp-right-ax:  $t \cdot u \frown w \implies w \setminus (t \cdot u) = (w \setminus t) \setminus u$ 
and resid-comp-left-ax:  $(t \cdot u) \setminus w = (t \setminus w) \cdot (u \setminus (w \setminus t))$ 
begin

  lemma comp-assocCC:
  shows  $t \cdot u \cdot v = (t \cdot u) \cdot v$ 
    using comp-extensionality-ax comp-assoc-ax
    by (metis (mono-tags, lifting) composite-of-comp-ax not-arr-null seq-def
        sources-composite-of targets-composite-of)

  lemma comp-nullCC:
  shows  $t \cdot \text{null} = \text{null}$  and  $\text{null} \cdot t = \text{null}$ 

```

**using** *comp-extensionality-ax not-arr-null* **apply** *blast*  
**by** (*simp add: comp-extensionality-ax seq-def*)

**lemma** *composable-iff-arr-comp<sub>CC</sub>*:  
**shows** *composable t u  $\longleftrightarrow$  arr (t · u)*  
**by** (*metis arr-composite-of comp-extensionality-ax composable-def*  
*composable-imp-seq composite-of-comp-ax not-arr-null*)

**lemma** *composable-iff-comp-not-null<sub>CC</sub>*:  
**shows** *composable t u  $\longleftrightarrow$  t · u  $\neq$  null*  
**by** (*metis comp-extensionality-ax composable-def composable-iff-arr-comp<sub>CC</sub>*  
*composite-of-comp-ax not-arr-null*)

**lemma** *con-comp-iff<sub>CC</sub>*:  
**shows** *w  $\frown$  t · u  $\longleftrightarrow$  composable t u  $\wedge$  w  $\setminus$  t  $\frown$  u*  
**by** (*metis composable-iff-comp-not-null<sub>CC</sub> composable-imp-seq composite-of-comp-ax*  
*con-composite-of-iff not-con-null(2)*)

**lemma** *con-compI<sub>CC</sub> [intro]*:  
**assumes** *composable t u* **and** *w  $\setminus$  t  $\frown$  u*  
**shows** *w  $\frown$  t · u* **and** *t · u  $\frown$  w*  
**using** *assms con-comp-iff<sub>CC</sub> con-sym* **by** *blast+*

**sublocale** *rts-with-composites resid*  
**using** *composite-of-comp-ax composable-def*  
**by** *unfold-locales auto*

**end**

**context** *paths-in-weakly-extensional-rts*  
**begin**

**abbreviation** *Comp*  
**where** *Comp T U  $\equiv$  if seq T U then T @ U else null*

**sublocale** *rts-with-chosen-composites Resid Comp*  
**proof**

**show**  $\bigwedge T U. \neg \text{seq } T U \implies \text{Comp } T U = \text{null}$   
**by** *argo*

**show**  $\bigwedge t u v. \text{seq } t u \implies \text{composite-of } t u (\text{Comp } t u)$   
**by** (*simp add: append-is-composite-of*)

**show**  $\bigwedge t u v. \llbracket \text{seq } t u; \text{seq } u v \rrbracket \implies \text{Comp } (\text{Comp } t u) v = \text{Comp } t (\text{Comp } u v)$   
**by** (*simp add: seq-def*)

**show**  $\bigwedge t u w. \text{con } (\text{Comp } t u) w \implies w \text{ ** } \text{Comp } t u = (w \text{ ** } t) \text{ ** } u$   
**by** (*metis (full-types) Arr.simps(1) Con-append(2) con-implies-arr(1)*  
*not-arr-null Resid.simps(1) Resid-append(2) paths-in-rts.seq-char*  
*paths-in-rts-axioms*)

**show**  $\bigwedge t u w. \text{Comp } t u \text{ ** } w = \text{Comp } (t \text{ ** } w) (u \text{ ** } (w \text{ ** } t))$   
**proof** –

```

fix t u w
have  $\llbracket \text{Arr } t; \text{Arr } u; \{ \text{Trg } t \} = \text{Srcs } u; (t @ u) \text{ ** } w \neq [] \rrbracket$ 
       $\implies \text{Trg } (t \text{ ** } w) \in \text{Srcs } (u \text{ ** } (w \text{ ** } t))$ 
  by (metis Arr.simps(1) Con-imp-Arr-Resid Con-implies-Arr(2) Resid-append-ind
      Srcs-Resid Trgs-Resid-sym Trgs-simpPWE insertI1)
thus  $\text{Comp } t \text{ ** } w = \text{Comp } (t \text{ ** } w) (u \text{ ** } (w \text{ ** } t))$ 
  using seq-char con-char null-char Con-implies-Arr
  apply auto[1]
  by (metis Arr-has-Src Con-append(1) Resid-append(1) Src-resid Srcs.simps(1)
      Srcs-simpPWE Trg-resid-sym con-imp-arr-resid singleton-iff Con-imp-eq-Srcs)+

  qed
qed

lemma extends-to-rts-with-chosen-composites:
shows rts-with-chosen-composites Resid Comp
  ..

end

context extensional-rts-with-composites
begin

  lemma extends-to-rts-with-chosen-composites:
  shows rts-with-chosen-composites resid comp
  using composable-iff-comp-not-null comp-is-composite-of(1) comp-assocEC
      resid-comp(1-2) comp-null(2) conI con-comp-iffEC(2) mediating-transition
  apply unfold-locales
  apply auto[4]
  by (metis comp-null(2) composable-imp-seq resid-comp(2))

  sublocale rts-with-chosen-composites resid comp
  using extends-to-rts-with-chosen-composites by blast

end

locale extension-to-paths =
  A: rts A +
  B: rts-with-chosen-composites B compB +
  F: simulation A B F +
  paths-in-rts A
for A :: 'a resid (infix <\A> 70)
and B :: 'b resid (infix <\B> 70)
and compB :: 'b comp (infixr <\B> 55)
and F :: 'a  $\Rightarrow$  'b
begin

  notation Resid (infix <\A*> 70)
  notation Resid1x (infix <\A1> 70)

```

**notation** *Residx1* (**infix**  $\langle * \setminus_A^1 \rangle$  70)  
**notation** *Con* (**infix**  $\langle * \frown_A^* \rangle$  70)  
**notation** *B.con* (**infix**  $\langle \frown_B \rangle$  50)

**fun** *map*  
**where** *map* [] = *B.null*  
| *map* [t] = *F t*  
| *map* (t # T) = (if *arr* (t # T) then *F t* ·<sub>B</sub> *map T* else *B.null*)

**lemma** *map-o-incl-eq*:  
**shows** *map* (*incl t*) = *F t*  
**by** (*simp add: null-char F.extensionality*)

**lemma** *extensionality*:  
**shows**  $\neg$  *arr T*  $\implies$  *map T* = *B.null*  
**using** *F.extensionality arr-char*  
**by** (*metis Arr.simps(2) map.elims*)

**lemma** *preserves-comp*:  
**shows**  $\llbracket T \neq []; U \neq []; \text{Arr} (T @ U) \rrbracket \implies \text{map} (T @ U) = \text{map } T \cdot_B \text{map } U$   
**proof** (*induct T arbitrary: U*)  
**show**  $\bigwedge U. [] \neq [] \implies \text{map} ([] @ U) = \text{map } [] \cdot_B \text{map } U$   
**by** *simp*  
**fix** *t* and *T U* :: 'a list  
**assume** *ind*:  $\bigwedge U. \llbracket T \neq []; U \neq []; \text{Arr} (T @ U) \rrbracket$   
 $\implies \text{map} (T @ U) = \text{map } T \cdot_B \text{map } U$   
**assume** *U*: *U*  $\neq []$   
**assume** *Arr*: *Arr* ((*t* # *T*) @ *U*)  
**hence** *1*: *Arr* (*t* # (*T* @ *U*))  
**by** *simp*  
**have** *2*: *Arr* (*t* # *T*)  
**by** (*metis Con-Arr-self Con-append(1) Con-implies-Arr(1) Arr U append-is-Nil-conv list.distinct(1)*)  
**show** *map* ((*t* # *T*) @ *U*) = *comp*<sub>B</sub> (*map* (*t* # *T*)) (*map U*)  
**proof** (*cases T = []*)  
**show** *T* = []  $\implies$  ?*thesis*  
**by** (*metis (full-types) 1 arr-char U append-Cons append-Nil list.exhaust map.simps(2) map.simps(3)*)  
**assume** *T*: *T*  $\neq []$   
**have** *map* ((*t* # *T*) @ *U*) = *map* (*t* # (*T* @ *U*))  
**by** *simp*  
**also have** ... = *F t* ·<sub>B</sub> *map* (*T* @ *U*)  
**using** *T 1*  
**by** (*metis arr-char Nil-is-append-conv list.exhaust map.simps(3)*)  
**also have** ... = *F t* ·<sub>B</sub> (*map T* ·<sub>B</sub> *map U*)  
**using** *ind*  
**by** (*metis 1 Con-Arr-self Con-implies-Arr(1) Con-rec(4) T U append-is-Nil-conv*)  
**also have** ... = (*F t* ·<sub>B</sub> *map T*) ·<sub>B</sub> *map U*  
**using** *B.comp-assoc<sub>CC</sub>* **by** *blast*

```

also have ... = map (t # T) ·B map U
  using T 2
  by (metis arr-char list.exhaust map.simps(3))
finally show map ((t # T) @ U) = map (t # T) ·B map U by simp
qed
qed

lemma preserves-arr-ind:
shows  $\llbracket \text{arr } T; a \in \text{Srcs } T \rrbracket \implies B.\text{arr } (\text{map } T) \wedge F a \in B.\text{sources } (\text{map } T)$ 
proof (induct T arbitrary: a)
  show  $\bigwedge a. \llbracket \text{arr } []; a \in \text{Srcs } [] \rrbracket \implies B.\text{arr } (\text{map } []) \wedge F a \in B.\text{sources } (\text{map } [])$ 
    using arr-char by simp
  fix a t T
  assume a: a ∈ Srcs (t # T)
  assume tT: arr (t # T)
  assume ind:  $\bigwedge a. \llbracket \text{arr } T; a \in \text{Srcs } T \rrbracket \implies B.\text{arr } (\text{map } T) \wedge F a \in B.\text{sources } (\text{map } T)$ 
  have 1: a ∈ A.sources t
    using a tT Con-imp-eq-Srcs Con-initial-right Srcs.simps(2) con-char
    by blast
  show B.arr (map (t # T)) ∧ F a ∈ B.sources (map (t # T))
  proof (cases T = [])
    show T = []  $\implies$  ?thesis
      using 1arr-char tT by auto
    assume T: T ≠ []
    obtain a' where a': a' ∈ A.targets t
      using tT 1 A.resid-source-in-targets by auto
    have 2: a' ∈ Srcs T
      using a' tT
    by (metis Con-Arr-self A.sources-resid Srcs.simps(2) arr-char T
      Con-imp-eq-Srcs Con-rec(4))
    have B.arr (map (t # T))  $\longleftrightarrow$  B.arr (F t ·B map T)
      using tT T by (metis map.simps(3) neq-Nil-conv)
    also have ...  $\longleftrightarrow$  True
  proof –
    have B.arr (F t ·B map T)
    proof –
      have B.seq (F t) (map T)
    proof
      show B.arr (map T)
        by (metis 2 A.rts-axioms Con-implies-Arr(2) Resid-cons(2)
          T arrE arr-resid ind not-arr-null null-char
          paths-in-rts.arr-char paths-in-rts.intro tT)
      show B.trg (F t)  $\sim_B$  B.src (map T)
    proof (intro B.coinitial-ide-are-cong)
      show B.ide (B.trg (F t))
        by (meson 1 A.in-sourcesE A.residuation-axioms B.ide-trg
          F.preserves-reflects-arr residuation.con-implies-arr(1))
      show B.ide (B.src (map T))
        by (simp add:  $\langle B.\text{arr } (\text{map } T) \rangle$ )
  
```

```

show  $B.coinitial (B.trg (F t)) (B.src (map T))$ 
  using  $a' ind extensionality$ 
  by ( $metis 1 2 A.con-implies-arr(1) A.in-sourcesE A.in-targetsE$ 
     $B.con-sym B.cong-implies-coinitial B.in-sourcesE B.not-arr-null$ 
     $B.sources-con-closed B.src-congI F.preserves-con F.preserves-trg$ 
     $\langle B.arr (map T) \rangle \langle B.ide (B.trg (F t)) \rangle$ )
qed
qed
thus  $?thesis$ 
  using  $B.composable-iff-arr-comp_{CC}$  by  $blast$ 
qed
thus  $?thesis$  by  $blast$ 
qed
finally have  $B.arr (map (t \# T))$  by  $blast$ 
moreover have  $F a \in B.sources (map (t \# T))$ 
proof –
  have  $F a \in B.sources (F t)$ 
    using  $1 tT F.preserves-sources$  by  $blast$ 
  moreover have  $B.sources (F t) = B.sources (map (t \# T))$ 
proof –
  have  $B.sources (F t) = B.sources (F t \cdot_B map T)$ 
    by ( $metis B.comp-extensionality-ax B.composite-of-comp-ax$ 
     $B.not-arr-null B.sources-composite-of \langle B.arr (F t \cdot_B map T) = True \rangle$ )
  also have  $\dots = B.sources (map (t \# T))$ 
    by ( $metis T list.exhaust map.simps(3) tT$ )
  finally show  $?thesis$  by  $blast$ 
qed
ultimately show  $?thesis$  by  $blast$ 
qed
ultimately show  $?thesis$  by  $blast$ 
qed
qed

```

```

lemma  $preserves-arr$ :
shows  $arr T \implies B.arr (map T)$ 
  using  $preserves-arr-ind arr-char Arr-has-Src$  by  $blast$ 

```

```

lemma  $preserves-sources$ :
assumes  $arr T$  and  $a \in Srcs T$ 
shows  $F a \in B.sources (map T)$ 
  using  $assms preserves-arr-ind$  by  $simp$ 

```

```

lemma  $preserves-targets$ :
shows  $\llbracket arr T; b \in Trgs T \rrbracket \implies F b \in B.targets (map T)$ 
proof ( $induct T$ )
  show  $\llbracket arr []; b \in Trgs [] \rrbracket \implies F b \in B.targets (map [])$ 
    by  $simp$ 
  fix  $t T$ 
  assume  $tT: arr (t \# T)$ 

```

```

assume  $b: b \in \text{Trgs } (t \# T)$ 
assume  $\text{ind}: \llbracket \text{arr } T; b \in \text{Trgs } T \rrbracket \implies F b \in B.\text{targets } (\text{map } T)$ 
show  $F b \in B.\text{targets } (\text{map } (t \# T))$ 
proof ( $\text{cases } T = []$ )
  show  $T = [] \implies ?thesis$ 
    using  $tT b \text{ arr-char}$  by  $\text{auto}$ 
  assume  $T: T \neq []$ 
  show  $?thesis$ 
  proof –
    have  $F b \in B.\text{targets } (\text{map } T)$ 
      by ( $\text{metis Resid-cons}(2) T \text{ Trgs.simps}(3) \text{ arrE } b \text{ con-char}$ 
         $\text{con-implies-arr}(2) \text{ ind neq-Nil-conv } tT$ )
    moreover have  $B.\text{targets } (\text{map } T) = B.\text{targets } (\text{map } (t \# T))$ 
    proof –
      have  $B.\text{targets } (\text{map } T) = B.\text{targets } (F t \cdot_B \text{map } T)$ 
        by ( $\text{metis B.comp-extensionality-ax B.composite-of-comp-ax}$ 
           $B.\text{not-arr-null } B.\text{targets-composite-of } T \text{ append-Cons } \text{append-Nil}$ 
           $\text{arr-char preserves-arr list.distinct}(1) \text{ map.simps}(2)$ 
           $\text{preserves-comp } tT$ )
      also have  $\dots = B.\text{targets } (\text{map } (t \# T))$ 
        by ( $\text{metis } T \text{ list.exhaust map.simps}(3) tT$ )
      finally show  $?thesis$  by  $\text{blast}$ 
    qed
  ultimately show  $?thesis$  by  $\text{blast}$ 
qed
qed
qed

```

```

lemma  $\text{preserves-Resid1x-ind}$ :
shows  $t^1 \setminus_A^* U \neq A.\text{null} \implies F t \frown_B \text{map } U \wedge F (t^1 \setminus_A^* U) = F t \setminus_B \text{map } U$ 
proof ( $\text{induct } U \text{ arbitrary: } t$ )
  show  $\bigwedge t. t^1 \setminus_A^* [] \neq A.\text{null} \implies F t \frown_B \text{map } [] \wedge F (t^1 \setminus_A^* []) = F t \setminus_B \text{map } []$ 
    by  $\text{simp}$ 
  fix  $t u U$ 
  assume  $uU: t^1 \setminus_A^* (u \# U) \neq A.\text{null}$ 
  assume  $\text{ind}: \bigwedge t. t^1 \setminus_A^* U \neq A.\text{null}$ 
     $\implies F t \frown_B \text{map } U \wedge F (t^1 \setminus_A^* U) = F t \setminus_B \text{map } U$ 
  show  $F t \frown_B \text{map } (u \# U) \wedge F (t^1 \setminus_A^* (u \# U)) = F t \setminus_B \text{map } (u \# U)$ 
  proof
    show  $1: F t \frown_B \text{map } (u \# U)$ 
    proof ( $\text{cases } U = []$ )
      show  $U = [] \implies ?thesis$ 
        using  $\text{Resid1x.simps}(2) \text{ map.simps}(2) F.\text{preserves-con } uU$  by  $\text{fastforce}$ 
      assume  $U: U \neq []$ 
      have  $\exists: [t]^* \setminus_A^* [u] \neq [] \wedge ([t]^* \setminus_A^* [u])^* \setminus_A^* U \neq []$ 
        using  $\text{Con-cons}(2) [\text{of } [t] U u]$ 
        by ( $\text{meson Resid1x-as-Resid}' U \text{ not-Cons-self2 } uU$ )
      hence  $2: F t \frown_B F u \wedge F t \setminus_B F u \frown_B \text{map } U$ 
      by ( $\text{metis Con-rec}(1) \text{ Con-sym } \text{Con-sym1 Resid1x-as-Resid Resid-rec}(1)$ )
    qed
  qed

```

$F.preserves-con\ F.preserves-resid\ ind)$   
**moreover have**  $B.seq\ (F\ u)\ (map\ U)$   
**by** (*metis*  $B.coinitialE\ B.con-imp-coinitial\ calculation\ B.seqI(1)$   
 $B.sources-resid$ )  
**ultimately have**  $F\ t\ \frown_B\ map\ ([u]\ @\ U)$   
**by** (*metis*  $3\ B.composite-of-comp-ax\ B.con-composite-of-iff\ Con-consI(2)$   
 $Con-implies-Arr(2)\ append-Cons\ list.simps(3)\ map.simps(2)\ preserves-comp$   
 $self-append-conv2$ )  
**thus** *?thesis* **by** *simp*  
**qed**  
**show**  $F\ (t\ ^1\backslash_A^*\ (u\ \#\ U)) = F\ t\ \backslash_B\ map\ (u\ \#\ U)$   
**proof** (*cases*  $U = []$ )  
**show**  $U = [] \implies ?thesis$   
**using**  $Resid1x.simps(2)\ F.preserves-resid\ map.simps(2)\ uU$  **by** *fastforce*  
**assume**  $U: U \neq []$   
**have**  $F\ (t\ ^1\backslash_A^*\ (u\ \#\ U)) = F\ ((t\ \backslash_A\ u)\ ^1\backslash_A^*\ U)$   
**using**  $Resid1x-as-Resid'\ Resid-rec(3)\ U\ uU$  **by** *metis*  
**also have**  $\dots = F\ (t\ \backslash_A\ u)\ \backslash_B\ map\ U$   
**using**  $uU\ U\ ind\ Con-rec(3)\ Resid1x-as-Resid\ [of\ t\ \backslash_A\ u\ U]$   
**by** (*metis*  $Resid1x.simps(3)\ list.exhaust$ )  
**also have**  $\dots = (F\ t\ \backslash_B\ F\ u)\ \backslash_B\ map\ U$   
**using**  $uU\ U$   
**by** (*metis*  $Resid1x-as-Resid'\ F.preserves-resid\ Con-rec(3)$ )  
**also have**  $\dots = F\ t\ \backslash_B\ (F\ u\ \cdot_B\ map\ U)$   
**proof** –  
**have**  $F\ u\ \cdot_B\ map\ U\ \frown_B\ F\ t$   
**proof**  
**show**  $B.composable\ (F\ u)\ (map\ U)$   
**by** (*metis*  $1\ B.composable-iff-comp-not-null_{CC}\ B.not-con-null(2)$   
 $U\ append.left-neutral\ append-Cons\ arr-char\ extensionality$   
 $map.simps(2)\ not-Cons-self\ preserves-comp$ )  
**show**  $F\ t\ \backslash_B\ F\ u\ \frown_B\ map\ U$   
**by** (*metis*  $B.arr-resid-iff-con\ Resid1x.simps(3)\ U\ calculation$   
 $ind\ list.exhaust\ uU$ )  
**qed**  
**thus** *?thesis*  
**using**  $B.resid-comp-right-ax\ [of\ F\ u\ map\ U\ F\ t]$  **by** *argo*  
**qed**  
**also have**  $\dots = F\ t\ \backslash_B\ map\ (u\ \#\ U)$   
**by** (*metis*  $Resid1x-as-Resid'\ con-char\ U\ map.simps(3)\ neq-Nil-conv$   
 $con-implies-arr(2)\ uU$ )  
**finally show** *?thesis* **by** *simp*  
**qed**  
**qed**  
**qed**

**lemma** *preserves-Resid1-ind:*

**shows**  $U\ *\backslash_A^1\ t \neq [] \implies map\ U\ \frown_B\ F\ t \wedge map\ (U\ *\backslash_A^1\ t) = map\ U\ \backslash_B\ F\ t$

**proof** (*induct*  $U$  *arbitrary: t*)

```

show  $\bigwedge t. [] * \setminus_A^1 t \neq [] \implies \text{map } [] \frown_B F t \wedge \text{map } ([] * \setminus_A^1 t) = \text{map } [] \setminus_B F t$ 
  by simp
fix  $t u U$ 
assume ind:  $\bigwedge t. U * \setminus_A^1 t \neq [] \implies \text{map } U \frown_B F t \wedge \text{map } (U * \setminus_A^1 t) = \text{map } U \setminus_B F t$ 
assume  $uU: (u \# U) * \setminus_A^1 t \neq []$ 
show  $\text{map } (u \# U) \frown_B F t \wedge \text{map } ((u \# U) * \setminus_A^1 t) = \text{map } (u \# U) \setminus_B F t$ 
proof (cases  $U = []$ )
  show  $U = [] \implies ?thesis$ 
    using Residx1.simps(2) F.preserves-con F.preserves-resid map.simps(2)  $uU$ 
    by presburger
  assume  $U: U \neq []$ 
  show ?thesis
proof
  show  $\text{map } (u \# U) \frown_B F t$ 
    using  $uU$  U Con-sym1 B.con-sym preserves-Resid1x-ind by blast
  show  $\text{map } ((u \# U) * \setminus_A^1 t) = \text{map } (u \# U) \setminus_B F t$ 
proof -
  have  $\text{map } ((u \# U) * \setminus_A^1 t) = \text{map } ((u \setminus_A t) \# U * \setminus_A^1 (t \setminus_A u))$ 
    using  $uU$  U Residx1-as-Resid Resid-rec(2) by fastforce
  also have  $\dots = F (u \setminus_A t) \cdot_B \text{map } (U * \setminus_A^1 (t \setminus_A u))$ 
    by (metis Residx1-as-Resid arr-char U Con-imp-Arr-Resid Con-rec(2) Resid-rec(2) list.exhaust map.simps(3)  $uU$ )
  also have  $\dots = F (u \setminus_A t) \cdot_B \text{map } U \setminus_B F (t \setminus_A u)$ 
    using  $uU$  U ind Con-rec(2) Residx1-as-Resid by force
  also have  $\dots = (F u \setminus_B F t) \cdot_B \text{map } U \setminus_B (F t \setminus_B F u)$ 
    using  $uU$  U
    by (metis Con-initial-right Con-rec(1) Con-sym1 Resid1x-as-Resid' Residx1-as-Resid F.preserves-resid)
  also have  $\dots = (F u \cdot_B \text{map } U) \setminus_B F t$ 
    using B.resid-comp-left-ax by auto
  also have  $\dots = \text{map } (u \# U) \setminus_B F t$ 
    by (metis Con-implies-Arr(2) Con-sym Residx1-as-Resid U arr-char map.simps(3) neq-Nil-conv  $uU$ )
  finally show ?thesis by simp
qed
qed
qed
qed

```

**lemma** *preserves-resid-ind*:

**shows**  $\text{con } T U \implies \text{map } T \frown_B \text{map } U \wedge \text{map } (T * \setminus_A^* U) = \text{map } T \setminus_B \text{map } U$

**proof** (*induct*  $T$  *arbitrary*:  $U$ )

**show**  $\bigwedge U. \text{con } [] U \implies \text{map } [] \frown_B \text{map } U \wedge \text{map } ([] * \setminus_A^* U) = \text{map } [] \setminus_B \text{map } U$

**using** *con-char* *Resid.simps(1)* **by** *blast*

**fix**  $t T U$

**assume**  $tT: \text{con } (t \# T) U$

**assume** *ind*:  $\bigwedge U. \text{con } T U \implies$

$\text{map } T \frown_B \text{map } U \wedge \text{map } (T * \setminus_A^* U) = \text{map } T \setminus_B \text{map } U$

**show**  $\text{map } (t \# T) \frown_B \text{map } U \wedge \text{map } ((t \# T) * \setminus_A^* U) = \text{map } (t \# T) \setminus_B \text{map } U$

```

proof (cases T = [])
  assume T: T = []
  show ?thesis
    using T tT
    apply simp
    by (metis Resid1x-as-Resid Resid1x-as-Resid con-char
      Con-sym Con-sym1 map.simps(2) preserves-Resid1x-ind)
next
assume T: T ≠ []
have 1: map (t # T) = F t ·B map T
  using tT T
  by (metis con-implies-arr(1) list.exhaust map.simps(3))
show ?thesis
proof
  show 2: B.con (map (t # T)) (map U)
    using T tT
    by (metis 1 B.composable-iff-comp-not-nullCC B.con-compICC(2) B.con-sym
      B.not-arr-null Con-cons(1) Resid1x-as-Resid con-char con-implies-arr(1-2)
      preserves-arr ind not-arr-null null-char preserves-Resid1x-ind)
  show map ((t # T) *A* U) = map (t # T) \B map U
proof -
  have map ((t # T) *A* U) = map (([t] *A* U) @ (T *A* (U *A* [t])))
    by (metis Resid.simps(1) Resid-cons(1) con-char ex-un-null tT)
  also have ... = map ([t] *A* U) ·B map (T *A* (U *A* [t]))
    by (metis Arr.simps(1) Con-imp-Arr-Resid Con-implies-Arr(2) Con-sym
      Resid-cons(1-2) con-char T preserves-comp tT)
  also have ... = (map [t] \B map U) ·B map (T *A* (U *A* [t]))
    by (metis Con-initial-right Con-sym Resid1x-as-Resid
      Resid1x-as-Resid con-char Con-sym1 map.simps(2)
      preserves-Resid1x-ind tT)
  also have ... = (map [t] \B map U) ·B (map T \B map (U *A* [t]))
    using tT T ind
    by (metis Con-cons(1) Con-sym Resid.simps(1) con-char)
  also have ... = (map [t] \B map U) ·B (map T \B (map U \B map [t]))
    using tT T
    by (metis Con-cons(1) Con-sym Resid.simps(2) Resid1x-as-Resid
      con-char map.simps(2) preserves-Resid1x-ind)
  also have ... = (F t \B map U) ·B (map T \B (map U \B F t))
    using tT T by simp
  also have ... = map (t # T) \B map U
    using 1 B.resid-comp-left-ax by auto
  finally show ?thesis by simp
qed
qed
qed
qed

```

**lemma** preserves-con:  
**assumes** con T U

```

shows map T  $\frown_B$  map U
  using assms preserves-resid-ind by simp

lemma preserves-resid:
assumes con T U
shows map (T  $\ast_A$  U) = map T  $\frown_B$  map U
  using assms preserves-resid-ind by simp

sublocale simulation Resid B map
  using con-char preserves-con preserves-resid extensionality
  by unfold-locales auto

lemma is-simulation:
shows simulation Resid B map
  ..

lemma is-extension:
shows map  $\circ$  incl = F
  using map-o-incl-eq by auto

lemma is-universal:
shows simulation Resid B map and map  $\circ$  incl = F
and  $\bigwedge F'. \llbracket \text{simulation Resid B } F'; F' \circ \text{incl} = F \rrbracket$ 
   $\implies \forall T. \text{arr } T \longrightarrow B.\text{cong } (F' T) (\text{map } T)$ 

proof –
  show simulation Resid B map and map  $\circ$  incl = F
    using map-o-incl-eq simulation-axioms by auto
  show  $\bigwedge F'. \llbracket \text{simulation Resid B } F'; F' \circ \text{incl} = F \rrbracket \implies \forall T. \text{arr } T \longrightarrow F' T \sim_B \text{map } T$ 
  proof (intro allI impI)
    fix F' T
    assume F': simulation Resid B F'
    assume 1: F'  $\circ$  incl = F
    interpret F': simulation Resid B F'
      using F' by simp
    show arr T  $\implies B.\text{cong } (F' T) (\text{map } T)$ 
    proof (induct T)
      show arr []  $\implies F' [] \sim_B \text{map } []$ 
        by (simp add: arr-char F'.extensionality)
      fix t T
      assume ind: arr T  $\implies F' T \sim_B \text{map } T$ 
      assume arr: arr (t # T)
      show F' (t # T)  $\sim_B \text{map } (t \# T)$ 
      proof (cases Arr (t # T))
        show  $\neg \text{Arr } (t \# T) \implies ?thesis$ 
          using arr arr-char by blast
        assume tT: Arr (t # T)
        show ?thesis
          proof (cases T = [])
            show 2: T = []  $\implies ?thesis$ 

```

```

    using  $F' \ 1 \ tT \ B.\text{prfx-reflexive} \ \text{arr} \ \text{map}.\text{simps}(2)$  by force
  assume  $T: T \neq []$ 
  have  $F' (t \# T) \sim_B F' [t] \cdot_B \text{map } T$ 
  proof –
    have  $F' (t \# T) = F' ([t] @ T)$ 
      by simp
    also have  $\dots \sim_B F' [t] \cdot_B F' T$ 
    proof –
      have composite-of  $[t] \ T \ ([t] @ T)$ 
        using  $T \ tT$ 
        by (metis (full-types) Arr.simps(2) Con-Arr-self
          append-is-composite-of Con-implies-Arr(1) Con-imp-eq-Srcs
          Con-rec(4) Resid-rec(1) Srcs-Resid seq-char A.arrI)
      thus ?thesis
        using  $F'.
        by (meson  $B.\text{comp-extensionality-ax}$   $B.\text{composable-def}$ 
           $B.\text{composite-of-unq-upto-cong}$   $B.\text{composable-iff-comp-not-null}_{CC}$ )
    qed
  also have  $F' [t] \cdot_B F' T \sim_B F' [t] \cdot_B \text{map } T$ 
  proof
    show  $0: F' [t] \cdot_B F' T \lesssim_B F' [t] \cdot_B \text{map } T$ 
    proof –
      have  $F' [t] \cdot_B F' T \frown_B F' [t] \cdot_B \text{map } T$ 
      proof
        show  $1: B.\text{composable} (F' [t]) (F' T)$ 
          using  $B.\text{composable-iff-comp-not-null}_{CC}$  calculation by force
        show  $(F' [t] \cdot_B \text{map } T) \setminus_B F' [t] \frown_B F' T$ 
          by (meson  $1 \ B.\text{composableD}(1-2)$   $B.\text{composableE}$   $B.\text{composable-def}$ 
             $B.\text{con-comp}_{CC}(1)$   $B.\text{cong-respects-seq}$   $B.\text{cong-subst-left}(1)$ 
             $B.\text{has-composites}$   $B.\text{prfx-implies-con}$   $B.\text{prfx-reflexive}$ 
             $B.\text{resid-composite-of}(2)$   $B.\text{rts-axioms}$   $F'.ind rts.composite-ofE)
      qed
      thus ?thesis
        by (metis  $B.\text{con-implies-arr}(2)$   $B.\text{con-sym}$   $B.\text{not-arr-null}$ 
           $B.\text{prfx-implies-con}$   $B.\text{prfx-transitive}$   $B.\text{resid-comp-right-ax}$ 
           $F'.calculation ind)
    qed
  show  $F' [t] \cdot_B \text{map } T \lesssim_B F' [t] \cdot_B F' T$ 
  proof –
    have  $1: F' [t] \cdot_B \text{map } T \frown_B F' [t] \cdot_B F' T$ 
    proof
      show  $B.\text{composable} (F' [t]) (\text{map } T)$ 
        using  $0 \ B.\text{composable-iff-comp-not-null}_{CC}$ 
        by force
      show  $(F' [t] \cdot_B F' T) \setminus_B F' [t] \frown_B \text{map } T$ 
        by (meson  $B.\text{con-comp-iff}_{CC}$   $B.\text{prfx-implies-con}$ 
           $\langle F' [t] \cdot_B F' T \lesssim_B F' [t] \cdot_B \text{map } T \rangle$ )
    qed$$$ 
```

```

hence  $(F' [t] \cdot_B \text{map } T) \setminus_B (F' [t] \cdot_B F' T) =$ 
 $((F' [t] \cdot_B \text{map } T) \setminus_B F' [t]) \setminus_B F' T$ 
using B.resid-comp-right-ax B.con-sym by blast
thus ?thesis
by (metis 1 B.con-arr-self B.con-implies-arr(1) B.cong-reflexive
B.not-ide-null B.null-is-zero(2) B.prfx-transitive
B.resid-comp-right-ax extensionality ind)
qed
qed
finally show ?thesis by blast
qed
also have  $F' [t] \cdot_B \text{map } T = (F' \circ \text{incl}) t \cdot_B \text{map } T$ 
using tT
by (simp add: arr-char null-char F'.extensionality)
also have  $\dots = F t \cdot_B \text{map } T$ 
using F' 1 by simp
also have  $\dots = \text{map } (t \# T)$ 
using T tT
by (metis arr-char list.exhaust map.simps(3))
finally show ?thesis by simp
qed
qed
qed
qed
qed
end

lemma extension-to-paths-comp:
assumes rts-with-chosen-composites B compB
and rts-with-chosen-composites C compC
and simulation A B F and simulation B C G
and  $\bigwedge t u. \text{rts.composable } B t u \implies G (\text{comp}_B t u) = \text{comp}_C (G t) (G u)$ 
shows extension-to-paths.map A C compC (G \circ F) = G \circ extension-to-paths.map A B compB
F
proof –
interpret A: rts A
using assms(3) simulation-def simulation-axioms-def by blast
interpret B: rts-with-chosen-composites B compB
using assms(1) by blast
interpret C: rts-with-chosen-composites C compC
using assms(2) by blast
interpret F: simulation A B F
using assms(3) by blast
interpret G: simulation B C G
using assms(4) by blast
interpret GoF: composite-simulation A B C F G ..
interpret Ap: paths-in-rts A ..
interpret Fx: extension-to-paths A B compB F ..

```

```

interpret G-o-Fx: composite-simulation Ap.Resid B C Fx.map G ..
interpret GoF-x: extension-to-paths A C compC ⟨G ∘ F⟩ ..
show GoF-x.map = G-o-Fx.map
proof
  fix T
  show GoF-x.map T = G-o-Fx.map T
  proof (cases Ap.arr T)
    show  $\neg$  Ap.arr T  $\implies$  ?thesis
      using G-o-Fx.extensionality GoF-x.extensionality by presburger
    assume T: Ap.arr T
    show ?thesis
  proof (induct T rule: Ap.Arr-induct)
    show Ap.Arr T
      using T Ap.arr-char by simp
    show  $\bigwedge t. Ap.Arr [t] \implies GoF-x.map [t] = G-o-Fx.map [t]$ 
      by auto
    show  $\bigwedge t U. \llbracket Ap.Arr (t \# U); U \neq []; GoF-x.map U = G-o-Fx.map U \rrbracket$ 
       $\implies GoF-x.map (t \# U) = G-o-Fx.map (t \# U)$ 
  proof –
    fix t U
    assume t: Ap.Arr (t \# U) and U: U ≠ []
    assume ind: GoF-x.map U = G-o-Fx.map U
    show GoF-x.map (t \# U) = G-o-Fx.map (t \# U)
    proof –
      have GoF-x.map (t \# U) = compC (GoF-x.map [t]) (GoF-x.map U)
        by (metis GoF-x.preserves-comp U append-Cons append-Nil
          list.distinct(1) t)
      also have  $\dots = comp_C (G (Fx.map [t])) (G (Fx.map U))$ 
        using ind by simp
      also have  $\dots = G (comp_B (Fx.map [t]) (Fx.map U))$ 
        by (metis B.composable-iff-comp-not-nullCC B.not-arr-null
          Fx.extension-to-paths-axioms Fx.preserves-comp U append-Cons
          append-Nil assms(5) extension-to-paths.preserves-arr
          extension-to-paths-def not-Cons-self2 paths-in-rts.arr-char t)
      also have  $\dots = G (Fx.map ([t] @ U))$ 
        by (metis Fx.preserves-comp U append.left-neutral append-Cons
          not-Cons-self2 t)
      also have  $\dots = G-o-Fx.map (t \# U)$ 
        by simp
      finally show ?thesis by blast
    qed
  qed
qed
qed
qed
qed
qed

```

**locale** *extension-to-composite-completion* =  
*A: rts A +*

```

B: extensional-rts-with-composites B +
simulation A B F
for A :: 'a resid    (infix <\A> 70)
and B :: 'b resid    (infix <\B> 70)
and F :: 'a ⇒ 'b
begin

  interpretation Ac: composite-completion A ..

    notation Ac.P.Resid    (infix <*\A*> 70)
    notation Ac.P.Resid1x (infix <^1\A*> 70)
    notation Ac.P.Residx1 (infix <*\A1> 70)
    notation Ac.P.Con      (infix <*\∩A*> 70)
    notation B.comp        (infixr <·B> 55)
    notation B.con         (infix <∩B> 50)

  interpretation F-ext: extension-to-paths A B B.comp F ..

  definition map
  where map = Ac.Q.ext-to-quotient B F-ext.map

  sublocale simulation Ac.resid B map
  unfolding map-def Ac.resid-def
  using Ac.Q.ext-to-quotient-props [of B F-ext.map] F-ext.simulation-axioms
    F-ext.preserves-ide B.extensional-rts-axioms Ac.P.ide-char Ac.P.ide-iff-NPath
  by blast

  lemma is-simulation:
  shows simulation Ac.resid B map
  ..

  lemma is-extension:
  shows map ∘ Ac.incl = F
  proof –
    have map ∘ Ac.incl = map ∘ Ac.Q.quot ∘ Ac.P.incl
    using Ac.incl-def by auto
    also have ... = F-ext.map ∘ Ac.P.incl
    using Ac.Q.ext-to-quotient-props [of B F-ext.map]
    by (simp add: B.extensional-rts-axioms F-ext.simulation-axioms
      Ac.P.ide-iff-NPath Ac.P.ide-char map-def)
    also have ... = F
    by (simp add: F-ext.is-extension)
    finally show ?thesis by blast
  qed

  lemma is-universal:
  shows  $\exists! F'. \text{simulation } Ac.resid B F' \wedge F' \circ Ac.incl = F$ 
  proof
    show 0: simulation Ac.resid B map ∧ map ∘ Ac.incl = F

```

```

    using simulation-axioms is-extension by auto
  fix F'
  assume F': simulation Ac.resid B F' ∧ F' ∘ Ac.incl = F
  interpret F': simulation Ac.resid B F'
    using F' by blast
  show F' = map
  proof -
    have F' ∘ Ac.Q.quot = F-ext.map
    proof -
      interpret F'-o-quot: simulation Ac.P.Resid B ⟨F' ∘ Ac.Q.quot⟩
        using F' Ac.Q.quotient-is-simulation Ac.resid-def by auto
      interpret incl: simulation A Ac.P.Resid Ac.P.incl
        using Ac.P.incl-is-simulation by blast
      interpret F'-o-quot-o-incl: composite-simulation A Ac.P.Resid B Ac.P.incl
        ⟨F' ∘ Ac.Q.quot⟩
      ..
      have (F' ∘ Ac.Q.quot) ∘ Ac.P.incl = F
        using F' Ac.incl-def by auto
      hence ∀ T. Ac.P.arr T ⟶ (F' ∘ Ac.Q.quot) T ∼B F-ext.map T
        using F-ext.is-universal(3) F'-o-quot.simulation-axioms by blast
      hence ∀ T. Ac.P.arr T ⟶ (F' ∘ Ac.Q.quot) T = F-ext.map T
        using B.cong-char by blast
      thus ?thesis
    proof -
      have ∀ as. (F' ∘ Ac.Q.quot) as = F-ext.map as ∨ ¬ Ac.P.arr as
        using ⟨∀ T. Ac.P.arr T ⟶ (F' ∘ Ac.Q.quot) T = F-ext.map T⟩ by blast
      then show ?thesis
        using F'-o-quot.extensionality F-ext.extensionality by fastforce
    qed
  qed
  thus ?thesis
  by (metis (no-types, lifting) 0 B.extensional-rts-axioms F'
    F-ext.preserves-ide F-ext.simulation-axioms Ac.P-ide-iff-NPath
    Ac.Q.ext-to-quotient-props(2) Ac.Q.is-couniversal map-def mem-Collect-eq
    Ac.resid-def)
  qed
  qed
end

context composite-completion
begin

lemma arrows-factor-as-paths:
  assumes arr T
  shows ∃ T. P.arr T ∧ extension-to-paths.map R resid comp incl T = T
  proof -
    interpret inclx: extension-to-paths R resid comp incl ..
    let ?T = P.Cong-class-rep T

```

```

have P.arr ?T
  by (metis P.Cong-class-memb-is-arr P.rep-in-Cong-class
    Q.quotient-by-coherent-normal-axioms assms
    quotient-by-coherent-normal.arr-char resid-def)
moreover have inclx.map ?T = T
proof -
  have  $\bigwedge T. P.arr T \implies inclx.map T = \{T\}$ 
  proof -
    fix T
    show P.arr T  $\implies inclx.map T = \{T\}$ 
    proof (induct T)
      show P.arr []  $\implies inclx.map [] = Q.quot []$ 
      using P.not-arr-null P.null-char by auto
    fix a U
    assume ind: P.arr U  $\implies inclx.map U = Q.quot U$ 
    assume aU: P.arr (a # U)
    show inclx.map (a # U) = Q.quot (a # U)
      using Q.quotient-is-simulation aU cong-char incl-def
      inclx.is-universal(3) resid-def
      by force
    qed
  qed
  thus ?thesis
  using Q.arr-char assms calculation resid-def by force
qed
ultimately show ?thesis by blast
qed

```

end

```

lemma extension-to-composite-completion-comp:
  assumes extensional-rts-with-composites B
  and extensional-rts-with-composites C
  and simulation A B F and simulation B C G
  shows extension-to-composite-completion.map A C (G o F) =
    G o extension-to-composite-completion.map A B F
proof -
  interpret B: extensional-rts-with-composites B
  using assms(1) by blast
  interpret C: extensional-rts-with-composites C
  using assms(2) by blast
  interpret F: simulation A B F
  using assms(3) by blast
  interpret G: simulation B C G
  using assms(4) by blast
  interpret GoF: composite-simulation A B C F G ..
  interpret Ac: composite-completion A ..
  interpret Fc: extension-to-composite-completion A B F ..
  interpret GoFc: extension-to-composite-completion A C GoF.map ..

```

```

show  $GoFc.map = G \circ Fc.map$ 
proof -
  have  $G \circ Fc.map \circ Ac.incl = GoFc.map \circ Ac.incl$ 
  using  $GoFc.is-extension$   $Fc.is-extension$   $comp-assoc$  by metis
  thus ?thesis
  using  $GoFc.is-extension$   $GoFc.is-universal$   $GoFc.is-simulation$ 
   $G.simulation-axioms$   $Fc.is-simulation$   $simulation-comp$ 
  by metis
qed
qed

```

**lemma** *composite-completion-of-rts:*

**assumes**  $rts\ A$

**shows**  $\exists(A' :: 'a\ composite-completion.arr\ resid)\ I.$

$extensional-rts-with-composites\ A' \wedge simulation\ A\ A'\ I \wedge$   
 $(\forall B\ (J :: 'a \Rightarrow 'c).\ extensional-rts-with-composites\ B \wedge simulation\ A\ B\ J$   
 $\longrightarrow (\exists!J'.\ simulation\ A'\ B\ J' \wedge J' \circ I = J))$

**proof** (*intro exI conjI*)

**interpret**  $A: rts\ A$

**using** *assms* by *auto*

**interpret**  $A': composite-completion\ A\ ..$

**show**  $extensional-rts-with-composites\ A'.resid$

..

**show**  $simulation\ A\ A'.resid\ A'.incl$

**using**  $A'.incl-is-simulation$  by *simp*

**show**  $\forall B\ (J :: 'a \Rightarrow 'c).\ extensional-rts-with-composites\ B \wedge simulation\ A\ B\ J$   
 $\longrightarrow (\exists!J'.\ simulation\ A'.resid\ B\ J' \wedge J' \circ A'.incl = J)$

**proof** (*intro allI impI*)

**fix**  $B :: 'c\ resid$  **and**  $J$

**assume**  $1: extensional-rts-with-composites\ B \wedge simulation\ A\ B\ J$

**interpret**  $B: extensional-rts-with-composites\ B$

**using**  $1$  by *simp*

**interpret**  $J: simulation\ A\ B\ J$

**using**  $1$  by *simp*

**interpret**  $J: extension-to-composite-completion\ A\ B\ J$

..

**show**  $\exists!J'.\ simulation\ A'.resid\ B\ J' \wedge J' \circ A'.incl = J$

**using**  $J.is-universal$  by *auto*

**qed**

**qed**

## 2.6 Constructions on RTS's

### 2.6.1 Products of RTS's

**locale** *product-rts* =

$A: rts\ A$  +

$B: rts\ B$

**for**  $A :: 'a \text{ resid}$     (**infix**  $\langle \backslash_A \rangle$  70)  
**and**  $B :: 'b \text{ resid}$     (**infix**  $\langle \backslash_B \rangle$  70)  
**begin**

**notation**  $A.con$     (**infix**  $\langle \frown_A \rangle$  50)  
**notation**  $A.prfx$    (**infix**  $\langle \lesssim_A \rangle$  50)  
**notation**  $A.cong$     (**infix**  $\langle \sim_A \rangle$  50)

**notation**  $B.con$     (**infix**  $\langle \frown_B \rangle$  50)  
**notation**  $B.prfx$    (**infix**  $\langle \lesssim_B \rangle$  50)  
**notation**  $B.cong$     (**infix**  $\langle \sim_B \rangle$  50)

**type-synonym**  $('c, 'd) \text{ arr} = 'c * 'd$

**abbreviation**  $(input) \text{ Null} :: ('a, 'b) \text{ arr}$   
**where**  $\text{Null} \equiv (A.null, B.null)$

**definition**  $\text{resid} :: ('a, 'b) \text{ arr} \Rightarrow ('a, 'b) \text{ arr} \Rightarrow ('a, 'b) \text{ arr}$   
**where**  $\text{resid } t \ u = (if \text{fst } t \ \frown_A \ \text{fst } u \ \wedge \ \text{snd } t \ \frown_B \ \text{snd } u$   
            $\text{then } (\text{fst } t \ \backslash_A \ \text{fst } u, \ \text{snd } t \ \backslash_B \ \text{snd } u)$   
            $\text{else } \text{Null})$

**notation**  $\text{resid}$     (**infix**  $\langle \backslash \rangle$  70)

**sublocale**  $\text{partial-magma resid}$   
**by**  $\text{unfold-locales}$   
     $(metis \ A.con\text{-implies-arr}(1-2) \ A.\text{not-arr-null} \ \text{fst-conv} \ \text{resid-def})$

**lemma**  $\text{is-partial-magma}$ :  
**shows**  $\text{partial-magma resid}$   
   ..

**lemma**  $\text{null-char [simp]}$ :  
**shows**  $\text{null} = \text{Null}$   
   **by**  $(metis \ B.null\text{-is-zero}(1) \ \text{ex-un-null} \ \text{null-is-zero}(1) \ \text{resid-def} \ B.conE \ \text{snd-conv})$

**sublocale**  $\text{residuation resid}$   
**proof**

**show**  $\bigwedge t \ u. \ t \ \backslash \ u \neq \text{null} \implies u \ \backslash \ t \neq \text{null}$   
   **by**  $(metis \ A.con\text{-def} \ A.con\text{-sym} \ \text{null-char} \ \text{prod.inject} \ \text{resid-def} \ B.con\text{-sym})$

**show**  $\bigwedge t \ u. \ t \ \backslash \ u \neq \text{null} \implies (t \ \backslash \ u) \ \backslash \ (t \ \backslash \ u) \neq \text{null}$   
   **by**  $(metis \ (\text{no-types, lifting}) \ A.\text{arrE} \ B.con\text{-def} \ B.con\text{-imp-arr-resid} \ \text{fst-conv} \ \text{null-char}$   
        $\text{resid-def} \ A.\text{arr-resid} \ \text{snd-conv})$

**show**  $\bigwedge v \ t \ u. \ (v \ \backslash \ t) \ \backslash \ (u \ \backslash \ t) \neq \text{null} \implies (v \ \backslash \ t) \ \backslash \ (u \ \backslash \ t) = (v \ \backslash \ u) \ \backslash \ (t \ \backslash \ u)$   
**proof** –

**fix**  $t \ u \ v$   
**assume**  $1: (v \ \backslash \ t) \ \backslash \ (u \ \backslash \ t) \neq \text{null}$   
**have**  $(\text{fst } v \ \backslash_A \ \text{fst } t) \ \backslash_A \ (\text{fst } u \ \backslash_A \ \text{fst } t) \neq A.null$   
   **by**  $(metis \ 1 \ A.\text{not-arr-null} \ \text{fst-conv} \ \text{null-char} \ \text{null-is-zero}(1-2))$

$resid-def\ A.arr-resid$   
**moreover have**  $(snd\ v \setminus_B\ snd\ t) \setminus_B\ (snd\ u \setminus_B\ snd\ t) \neq B.null$   
**by**  $(metis\ 1\ B.not-arr-null\ snd-conv\ null-char\ null-is-zero(1-2))$   
 $resid-def\ B.arr-resid$   
**ultimately show**  $(v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$   
**using**  $resid-def\ null-char\ A.con-def\ B.con-def\ A.cube\ B.cube$   
**apply**  $simp$   
**by**  $(metis\ (no-types,\ lifting)\ A.conI\ A.con-sym-ax\ A.resid-reflects-con\ B.con-sym-ax\ B.null-is-zero(1))$   
**qed**  
**qed**

**lemma**  $is-residuation$ :  
**shows**  $residuation\ resid$   
 $..$

**notation**  $con$     **(infix**  $\langle \frown \rangle$   $50$ )

**lemma**  $arr-char$  [ $iff$ ]:  
**shows**  $arr\ t \longleftrightarrow A.arr\ (fst\ t) \wedge B.arr\ (snd\ t)$   
**by**  $(metis\ (no-types,\ lifting)\ A.arr-def\ B.arr-def\ B.conE\ null-char\ resid-def\ arr-def\ con-def\ snd-eqD)$

**lemma**  $ide-char$  [ $iff$ ]:  
**shows**  $ide\ t \longleftrightarrow A.ide\ (fst\ t) \wedge B.ide\ (snd\ t)$   
**by**  $(metis\ (no-types,\ lifting)\ A.residuation-axioms\ B.residuation-axioms\ arr-char\ arr-def\ fst-conv\ null-char\ prod.collapse\ resid-def\ residuation.conE\ residuation.ide-def\ residuation.ide-implies-arr\ residuation-axioms\ snd-conv)$

**lemma**  $con-char$  [ $iff$ ]:  
**shows**  $t \frown u \longleftrightarrow fst\ t \frown_A\ fst\ u \wedge snd\ t \frown_B\ snd\ u$   
**by**  $(simp\ add:\ con-def\ resid-def\ B.con-def)$

**lemma**  $trg-char$ :  
**shows**  $trg\ t = (if\ arr\ t\ then\ (A.trg\ (fst\ t),\ B.trg\ (snd\ t))\ else\ Null)$   
**using**  $A.trg-def\ B.trg-def\ resid-def\ trg-def$  **by**  $auto$

**sublocale**  $rts\ resid$

**proof**

**show**  $\bigwedge t. arr\ t \implies ide\ (trg\ t)$   
**by**  $(simp\ add:\ trg-char)$   
**show**  $1: \bigwedge a\ t. \llbracket ide\ a; t \frown a \rrbracket \implies t \setminus a = t$   
**by**  $(simp\ add:\ A.resid-arr-ide\ B.resid-arr-ide\ resid-def)$   
**thus**  $\bigwedge a\ t. \llbracket ide\ a; a \frown t \rrbracket \implies ide\ (a \setminus t)$   
**using**  $arr-resid\ cube$   
**apply**  $(elim\ ideE,\ intro\ ideI)$   
**apply**  $auto$   
**by**  $(metis\ 1\ conI\ con-sym-ax\ ideI\ null-is-zero(2))$   
**show**  $\bigwedge t\ u. t \frown u \implies \exists a. ide\ a \wedge a \frown t \wedge a \frown u$

```

proof –
  fix  $t\ u$ 
  assume  $tu: t \frown u$ 
  obtain  $a1$  where  $a1: a1 \in A.sources\ (fst\ t) \cap A.sources\ (fst\ u)$ 
    by  $(meson\ A.con-imp-common-source\ all-not-in-conv\ con-char\ tu)$ 
  obtain  $a2$  where  $a2: a2 \in B.sources\ (snd\ t) \cap B.sources\ (snd\ u)$ 
    by  $(meson\ B.con-imp-common-source\ all-not-in-conv\ con-char\ tu)$ 
  have  $ide\ (a1,\ a2) \wedge (a1,\ a2) \frown t \wedge (a1,\ a2) \frown u$ 
    using  $a1\ a2\ ide-char\ con-char$ 
    by  $(metis\ A.con-imp-common-source\ A.in-sourcesE\ A.sources-eqI\ B.con-imp-common-source\ B.in-sourcesE\ B.sources-eqI\ con-sym\ fst-conv\ inf-idem\ snd-conv\ tu)$ 
  thus  $\exists a. ide\ a \wedge a \frown t \wedge a \frown u$  by  $blast$ 
qed
show  $\bigwedge t\ u\ v. \llbracket ide\ (t \setminus u); u \frown v \rrbracket \implies t \setminus u \frown v \setminus u$ 
proof –
  fix  $t\ u\ v$ 
  assume  $tu: ide\ (t \setminus u)$ 
  assume  $uv: u \frown v$ 
  have  $A.ide\ (fst\ t \setminus_A\ fst\ u) \wedge B.ide\ (snd\ t \setminus_B\ snd\ u)$ 
    using  $tu\ ide-char$ 
    by  $(metis\ conI\ con-char\ fst-eqD\ ide-implies-arr\ not-arr-null\ resid-def\ snd-conv)$ 
  moreover have  $fst\ u \frown_A\ fst\ v \wedge snd\ u \frown_B\ snd\ v$ 
    using  $uv\ con-char$  by  $blast$ 
  ultimately show  $t \setminus u \frown v \setminus u$ 
    by  $(simp\ add: A.con-target\ A.con-sym\ A.prfx-implies-con\ B.con-target\ B.con-sym\ B.prfx-implies-con\ resid-def)$ 
qed
qed

```

**lemma** *is-rts*:  
**shows**  $rts\ resid$   
 ..

**notation**  $prfx$  (infix  $\langle \lesssim \rangle$  50)  
**notation**  $cong$  (infix  $\langle \sim \rangle$  50)

**lemma** *sources-char*:  
**shows**  $sources\ t = A.sources\ (fst\ t) \times B.sources\ (snd\ t)$   
**by** *force*

**lemma** *targets-char*:  
**shows**  $targets\ t = A.targets\ (fst\ t) \times B.targets\ (snd\ t)$   
**proof**  
**show**  $targets\ t \subseteq A.targets\ (fst\ t) \times B.targets\ (snd\ t)$   
**using**  $targets-def\ ide-char\ con-char\ resid-def\ trg-def$  **by** *auto*  
**show**  $A.targets\ (fst\ t) \times B.targets\ (snd\ t) \subseteq targets\ t$   
**proof**  
**fix**  $a$

**assume**  $a: a \in A.targets\ (fst\ t) \times B.targets\ (snd\ t)$   
**show**  $a \in targets\ t$   
**proof**  
  **show**  $ide\ a$   
    **using**  $a\ ide-char\ by\ auto$   
  **show**  $trg\ t \frown a$   
    **using**  $a\ trg-char\ con-char\ [of\ trg\ t\ a]$   
    **by**  $(metis\ (no-types,\ lifting)\ SigmaE\ arr-char\ con-char\ con-implies-arr(1)\ fst-conv\ A.in-targetsE\ B.in-targetsE\ A.arr-resid-iff-con\ B.arr-resid-iff-con\ A.trg-def\ B.trg-def\ snd-conv)$   
**qed**  
**qed**  
**qed**

**lemma**  $prfx-char$ :  
**shows**  $t \lesssim u \iff fst\ t \lesssim_A\ fst\ u \wedge snd\ t \lesssim_B\ snd\ u$   
  **using**  $A.prfx-implies-con\ B.prfx-implies-con\ resid-def\ by\ auto$

**lemma**  $cong-char$ :  
**shows**  $t \sim u \iff fst\ t \sim_A\ fst\ u \wedge snd\ t \sim_B\ snd\ u$   
  **using**  $prfx-char\ by\ auto$

**lemma**  $join-of-char$ :  
**shows**  $join-of\ t\ u\ v \iff A.join-of\ (fst\ t)\ (fst\ u)\ (fst\ v) \wedge B.join-of\ (snd\ t)\ (snd\ u)\ (snd\ v)$   
**and**  $joinable\ t\ u \iff A.joinable\ (fst\ t)\ (fst\ u) \wedge B.joinable\ (snd\ t)\ (snd\ u)$   
**proof** –

**show**  $\bigwedge v. join-of\ t\ u\ v \iff A.join-of\ (fst\ t)\ (fst\ u)\ (fst\ v) \wedge B.join-of\ (snd\ t)\ (snd\ u)\ (snd\ v)$

**proof**

**fix**  $v$

**show**  $join-of\ t\ u\ v \implies$

$A.join-of\ (fst\ t)\ (fst\ u)\ (fst\ v) \wedge B.join-of\ (snd\ t)\ (snd\ u)\ (snd\ v)$

**proof** –

**assume**  $1: join-of\ t\ u\ v$

**have**  $2: t \frown u \wedge t \frown v \wedge u \frown v \wedge u \frown t \wedge v \frown t \wedge v \frown u$

**by**  $(meson\ 1\ bounded-imp-con\ con-prfx-composite-of(1)\ join-ofE\ con-sym)$

**show**  $A.join-of\ (fst\ t)\ (fst\ u)\ (fst\ v) \wedge B.join-of\ (snd\ t)\ (snd\ u)\ (snd\ v)$

**using**  $1\ 2\ prfx-char\ resid-def$

**by**  $(elim\ conjE\ join-ofE\ composite-ofE\ congE\ conE,$

$intro\ conjI\ A.join-ofI\ B.join-ofI\ A.composite-ofI\ B.composite-ofI)$

$auto$

**qed**

**show**  $A.join-of\ (fst\ t)\ (fst\ u)\ (fst\ v) \wedge B.join-of\ (snd\ t)\ (snd\ u)\ (snd\ v)$

$\implies join-of\ t\ u\ v$

**using**  $cong-char\ resid-def$

**by**  $(elim\ conjE\ A.join-ofE\ B.join-ofE\ A.composite-ofE\ B.composite-ofE,$

$intro\ join-ofI\ composite-ofI)$

$auto$

**qed**

```

thus joinable t u  $\longleftrightarrow$  A.joinable (fst t) (fst u)  $\wedge$  B.joinable (snd t) (snd u)
using joinable-def A.joinable-def B.joinable-def by simp
qed

end

locale product-of-weakly-extensional-rts =
  A: weakly-extensional-rts A +
  B: weakly-extensional-rts B +
  product-rts
begin

  sublocale weakly-extensional-rts resid
  proof
    show  $\bigwedge t u. \llbracket t \sim u; \text{ide } t; \text{ide } u \rrbracket \implies t = u$ 
    by (metis cong-char ide-char prod.exhaust-sel A.weak-extensionality B.weak-extensionality)
  qed

  lemma is-weakly-extensional-rts:
  shows weakly-extensional-rts resid
  ..

  lemma src-char:
  shows src t = (if arr t then (A.src (fst t), B.src (snd t)) else null)
  proof (cases arr t)
    show  $\neg \text{arr } t \implies ?thesis$ 
    using src-def by presburger
    assume t: arr t
    show ?thesis
    using t con-char arr-char
    by (intro src-eqI) auto
  qed

end

locale product-of-extensional-rts =
  A: extensional-rts A +
  B: extensional-rts B +
  product-of-weakly-extensional-rts
begin

  sublocale extensional-rts resid
  proof
    show  $\bigwedge t u. t \sim u \implies t = u$ 
    by (metis A.extensionality B.extensionality cong-char prod.collapse)
  qed

  lemma is-extensional-rts:
  shows extensional-rts resid

```

..  
end

## Product Simulations

```

locale product-simulation =
  A1: rts A1 +
  A0: rts A0 +
  B1: rts B1 +
  B0: rts B0 +
  A1xA0: product-rts A1 A0 +
  B1xB0: product-rts B1 B0 +
  F1: simulation A1 B1 F1 +
  F0: simulation A0 B0 F0
for A1 :: 'a1 resid (infix <\A1> 70)
and A0 :: 'a0 resid (infix <\A0> 70)
and B1 :: 'b1 resid (infix <\B1> 70)
and B0 :: 'b0 resid (infix <\B0> 70)
and F1 :: 'a1 ⇒ 'b1
and F0 :: 'a0 ⇒ 'b0
begin

```

```

definition map
where map = (λa. if A1xA0.arr a then (F1 (fst a), F0 (snd a))
              else (F1 A1.null, F0 A0.null))

```

```

lemma map-simp [simp]:
assumes A1.arr a1 and A0.arr a0
shows map (a1, a0) = (F1 a1, F0 a0)
using assms map-def by auto

```

```

sublocale simulation A1xA0.resid B1xB0.resid map
proof

```

```

  show  $\bigwedge t. \neg A1xA0.arr t \implies map\ t = B1xB0.null$ 
    using map-def F1.extensionality F0.extensionality by auto
  show  $\bigwedge t\ u. A1xA0.con\ t\ u \implies B1xB0.con\ (map\ t)\ (map\ u)$ 
    using A1xA0.con-char B1xB0.con-char A1.con-implies-arr A0.con-implies-arr by auto
  show  $\bigwedge t\ u. A1xA0.con\ t\ u \implies map\ (A1xA0.resid\ t\ u) = B1xB0.resid\ (map\ t)\ (map\ u)$ 
    using A1xA0.resid-def B1xB0.resid-def A1.con-implies-arr A0.con-implies-arr
    by auto

```

qed

```

lemma is-simulation:
shows simulation A1xA0.resid B1xB0.resid map

```

..  
end

## Binary Simulations

```

locale binary-simulation =
  A1: rts A1 +
  A0: rts A0 +
  A: product-rts A1 A0 +
  B: rts B +
  simulation A.resid B F
for A1 :: 'a1 resid (infix <\A1> 70)
and A0 :: 'a0 resid (infix <\A0> 70)
and B :: 'b resid (infix <\B> 70)
and F :: 'a1 * 'a0 ⇒ 'b
begin

  lemma fixing-ide-gives-simulation-1:
  assumes A1.ide a1
  shows simulation A0 B (λt0. F (a1, t0))
  proof
    show  $\bigwedge t0. \neg A0.arr\ t0 \implies F\ (a1, t0) = B.null$ 
      using assms extensionality A.arr-char by simp
    show  $\bigwedge t0\ u0. A0.con\ t0\ u0 \implies B.con\ (F\ (a1, t0))\ (F\ (a1, u0))$ 
      using assms A.con-char preserves-con by auto
    show  $\bigwedge t0\ u0. A0.con\ t0\ u0 \implies F\ (a1, t0 \setminus_{A0}\ u0) = F\ (a1, t0) \setminus_B\ F\ (a1, u0)$ 
      using assms A.con-char A.resid-def preserves-resid
      by (metis A1.ideE fst-conv snd-conv)
  qed

  lemma fixing-ide-gives-simulation-0:
  assumes A0.ide a0
  shows simulation A1 B (λt1. F (t1, a0))
  proof
    show  $\bigwedge t1. \neg A1.arr\ t1 \implies F\ (t1, a0) = B.null$ 
      using assms extensionality A.arr-char by simp
    show  $\bigwedge t1\ u1. A1.con\ t1\ u1 \implies B.con\ (F\ (t1, a0))\ (F\ (u1, a0))$ 
      using assms A.con-char preserves-con by auto
    show  $\bigwedge t1\ u1. A1.con\ t1\ u1 \implies F\ (t1 \setminus_{A1}\ u1, a0) = F\ (t1, a0) \setminus_B\ F\ (u1, a0)$ 
      using assms A.con-char A.resid-def preserves-resid
      by (metis A0.ideE fst-conv snd-conv)
  qed

end

```

### 2.6.2 Sub-RTS's

A sub-RTS of an RTS  $R$  may be determined by specifying a subset of the transitions of  $R$  that is closed under residuation and in addition includes some common source for every consistent pair of transitions contained in it.

```

locale sub-rts =
  R: rts R

```

```

for  $R :: 'a \text{ resid}$  (infix  $\langle \backslash_R \rangle$  70)
and  $Arr :: 'a \Rightarrow \text{bool}$  +
assumes inclusion:  $Arr\ t \Longrightarrow R.\text{arr}\ t$ 
and resid-closed:  $\llbracket Arr\ t; Arr\ u; R.\text{con}\ t\ u \rrbracket \Longrightarrow Arr\ (t \backslash_R u)$ 
and enough-sources:  $\llbracket Arr\ t; Arr\ u; R.\text{con}\ t\ u \rrbracket \Longrightarrow$ 
 $\exists a. Arr\ a \wedge a \in R.\text{sources}\ t \wedge a \in R.\text{sources}\ u$ 
begin

  notation  $R.\text{con}$     (infix  $\langle \frown_R \rangle$  50)
  notation  $R.\text{prfx}$  (infix  $\langle \lesssim_R \rangle$  50)
  notation  $R.\text{cong}$   (infix  $\langle \sim_R \rangle$  50)

  definition resid ::  $'a \text{ resid}$  (infix  $\langle \backslash \rangle$  70)
  where  $t \backslash u \equiv \text{if } Arr\ t \wedge Arr\ u \wedge t \frown_R u \text{ then } t \backslash_R u \text{ else } R.\text{null}$ 

  sublocale partial-magma resid
    using  $R.\text{not-con-null}(2)$   $R.\text{null-is-zero}(1)$  resid-def
    by unfold-locales metis

  lemma is-partial-magma:
  shows partial-magma resid
  ..

  lemma null-char:
  shows  $\text{null} = R.\text{null}$ 
    by (metis R.not-arr-null inclusion null-eqI resid-def)

  sublocale residuation resid
    using  $R.\text{conE}$   $R.\text{con-sym}$   $R.\text{not-con-null}(1)$   $\text{null-is-zero}(1)$  resid-def
    apply unfold-locales
    apply metis
    apply (metis R.con-def R.con-imp-arr-resid resid-closed)
    by (metis (no-types, lifting) R.con-def R.cube resid-closed)

  lemma is-residuation:
  shows residuation resid
  ..

  notation  $\text{con}$     (infix  $\langle \frown \rangle$  50)

  lemma arr-char:
  shows  $\text{arr}\ t \longleftrightarrow Arr\ t$ 
    by (metis R.con-arr-self R.con-def R.not-arr-null arrE con-def inclusion
      null-is-zero(2) resid-def residuation.con-implies-arr(1) residuation-axioms)

  lemma ide-char:
  shows  $\text{ide}\ t \longleftrightarrow Arr\ t \wedge R.\text{ide}\ t$ 
    by (metis R.ide-def arrI arr-char con-def ide-def not-arr-null resid-def)

```

**lemma** *con-char*:  
**shows**  $con\ t\ u \iff Arr\ t \wedge Arr\ u \wedge R.con\ t\ u$   
**by** (*metis R.conE arr-char con-def not-arr-null null-is-zero(1) resid-def*)

**lemma** *trg-char*:  
**shows**  $trg = (\lambda t. \text{if } arr\ t \text{ then } R.trg\ t \text{ else } null)$   
**using** *arr-char trg-def R.trg-def resid-def* **by** *fastforce*

**sublocale** *rts resid*  
**using** *arr-char ide-char con-char trg-char resid-def resid-closed inclusion*  
**apply** *unfold-locales*  
**using** *R.prfx-reflexive trg-def* **apply** *force*  
**apply** (*simp add: R.resid-arr-ide*)  
**apply** *simp*  
**apply** (*meson R.con-sym R.in-sourcesE enough-sources*)  
**by** (*metis (no-types, lifting) R.con-target arr-resid-iff-con con-sym-ax null-char*)

**lemma** *is-rts*:  
**shows** *rts resid*  
**..**

**notation** *prfx* (**infix**  $\langle \lesssim \rangle$  50)  
**notation** *cong* (**infix**  $\langle \sim \rangle$  50)

**lemma** *sources-subset*:  
**shows**  $sources\ t \subseteq \{a. Arr\ t \wedge a \in R.sources\ t\}$   
**using** *con-char ide-char* **by** *fastforce*

**lemma** *targets-subset*:  
**shows**  $targets\ t \subseteq \{b. Arr\ t \wedge b \in R.targets\ t\}$   
**proof**  
**fix** *b*  
**assume** *b: b ∈ targets t*  
**show**  $b \in \{b. Arr\ t \wedge b \in R.targets\ t\}$   
**by** (*metis CollectI R.rts-axioms arr-char arr-iff-has-target b con-char emptyE ide-char in-targetsE rts.in-targetsI trg-char*)  
**qed**

**lemma** *prfx-char<sub>SRTS</sub>*:  
**shows**  $prfx\ t\ u \iff Arr\ t \wedge Arr\ u \wedge R.prfx\ t\ u$   
**using** *arr-char con-char ide-char*  
**by** (*metis R.prfx-implies-con prfx-implies-con resid-closed resid-def*)

**lemma** *cong-char<sub>SRTS</sub>*:  
**shows**  $t \sim u \iff Arr\ t \wedge Arr\ u \wedge t \sim_R u$   
**using** *prfx-char<sub>SRTS</sub>* **by** *force*

**lemma** *composite-of-char*:  
**shows**  $composite-of\ t\ u\ v \iff Arr\ t \wedge Arr\ u \wedge Arr\ v \wedge R.composite-of\ t\ u\ v$

**proof**

**show**  $composite\text{-of } t \ u \ v \implies Arr \ t \wedge Arr \ u \wedge Arr \ v \wedge R.composite\text{-of } t \ u \ v$   
**by** (*metis*  $R.composite\text{-of-def}$   $R.con\text{-sym}$   $composite\text{-ofE}$   $con\text{-char}$   $prfx\text{-char}_{SRTS}$   
 $resid\text{-def}$   $rts.prfx\text{-implies-con}$   $rts\text{-axioms}$ )  
**show**  $Arr \ t \wedge Arr \ u \wedge Arr \ v \wedge R.composite\text{-of } t \ u \ v \implies composite\text{-of } t \ u \ v$   
**using**  $composite\text{-of-def}$   $resid\text{-closed}$   $resid\text{-def}$   $rts.composite\text{-ofE}$   $ide\text{-char}$   
**by** *fastforce*

**qed**

**lemma** *join-of-char*:

**shows**  $join\text{-of } t \ u \ v \longleftrightarrow Arr \ t \wedge Arr \ u \wedge Arr \ v \wedge R.join\text{-of } t \ u \ v$   
**using**  $composite\text{-of-char}$   
**by** (*metis*  $R.bounded\text{-imp-con}$   $R.join\text{-of-def}$   $join\text{-of-def}$   $resid\text{-closed}$   $resid\text{-def}$ )

**lemma** *preserves-weakly-extensional-rts*:

**assumes**  $weakly\text{-extensional-rts } R$   
**shows**  $weakly\text{-extensional-rts } resid$   
**by** (*metis*  $assms$   $cong\text{-char}_{SRTS}$   $ide\text{-char}$   $rts\text{-axioms}$   $weakly\text{-extensional-rts.intro}$   
 $weakly\text{-extensional-rts.weak-extensionality}$   $weakly\text{-extensional-rts-axioms.intro}$ )

**lemma** *preserves-extensional-rts*:

**assumes**  $extensional\text{-rts } R$   
**shows**  $extensional\text{-rts } resid$   
**by** (*meson*  $assms$   $extensional\text{-rts.cong-char}$   $extensional\text{-rts.intro}$   
 $extensional\text{-rts-axioms.intro}$   $prfx\text{-char}_{SRTS}$   $rts\text{-axioms}$ )

**abbreviation** *incl*

**where**  $incl \ t \equiv \text{if } arr \ t \text{ then } t \text{ else } null$

**sublocale** *Incl*:  $simulation \ resid \ R \ incl$

**using**  $resid\text{-closed}$   $resid\text{-def}$   
**by**  $unfold\text{-locales}$  (*auto simp add: null-char arr-char con-char*)

**lemma** *inclusion-is-simulation*:

**shows**  $simulation \ resid \ R \ incl$

..

**lemma** *incl-cancel-left*:

**assumes**  $transformation \ X \ resid \ F \ G \ T$  **and**  $transformation \ X \ resid \ F' \ G' \ T'$

**and**  $incl \circ T = incl \circ T'$

**shows**  $T = T'$

**proof**

**fix**  $x$

**interpret**  $T$ :  $transformation \ X \ resid \ F \ G \ T$

**using**  $assms(1)$  **by** *blast*

**interpret**  $T'$ :  $transformation \ X \ resid \ F' \ G' \ T'$

**using**  $assms(2)$  **by** *blast*

**show**  $T \ x = T' \ x$

**proof** –

```

have  $T x = (incl \circ T) x$ 
  using  $T.extensionality$   $T.A.prfx-reflexive$   $T.respects-cong$   $arr-char$   $prfx-char_{SRTS}$ 
  by auto
also have  $\dots = (incl \circ T') x$ 
  using  $assms(3)$  by auto
also have  $\dots = T' x$ 
  using  $T'.extensionality$   $T.A.prfx-reflexive$   $T'.respects-cong$   $arr-char$   $prfx-char_{SRTS}$ 
  by fastforce
finally show ?thesis by blast
qed
qed

```

```

lemma incl-reflects-con:
assumes  $R.con$   $(incl\ t)$   $(incl\ u)$ 
shows  $con\ t\ u$ 
  by (metis (full-types)  $R.not-con-null(1)$   $R.not-con-null(2)$   $arr-char$ 
     $assms\ con-char\ null-char$ )

```

```

lemma corestriction-of-simulation:
assumes  $simulation\ X\ R\ F$ 
and  $\bigwedge x. residuation.arr\ X\ x \implies Arr\ (F\ x)$ 
shows  $simulation\ X\ resid\ F$  and  $incl \circ F = F$ 
proof –
  interpret  $X: rts\ X$ 
    using  $assms(1)$  simulation-def by blast
  interpret  $F: simulation\ X\ R\ F$ 
    using  $assms(1)$  by blast
  interpret  $F': simulation\ X\ resid\ F$ 
    using  $assms(2)$   $con-char\ resid-def$   $F.extensionality\ null-char$ 
     $X.con-implies-arr(1-2)$ 
    by unfold-locales auto
  show  $1: simulation\ X\ resid\ F$  ..
  show  $incl \circ F = F$ 
    using  $F.extensionality\ null-char$  by fastforce
qed

```

```

lemma corestriction-of-transformation:
assumes  $simulation\ X\ resid\ F$  and  $simulation\ X\ resid\ G$ 
and  $transformation\ X\ R\ F\ G\ T$ 
and  $\bigwedge x. residuation.arr\ X\ x \implies Arr\ (T\ x)$ 
shows  $transformation\ X\ resid\ F\ G\ T$  and  $incl \circ T = T$ 
proof –
  interpret  $X: rts\ X$ 
    using  $assms(3)$  transformation-def by blast
  interpret  $R: weakly-extensional-rts\ R$ 
    using  $assms(3)$  transformation-def by blast
  interpret  $S: weakly-extensional-rts\ resid$ 
    by (simp add: R.weakly-extensional-rts-axioms preserves-weakly-extensional-rts)
  interpret  $F: simulation\ X\ resid\ F$ 

```

```

    using assms(1) transformation-def by blast
  interpret G: simulation X resid G
    using assms(2) transformation-def by blast
  interpret T: transformation X R F G T
    using assms(3) by blast
  interpret T': transformation X resid F G T
  proof
    show  $\bigwedge f. \neg X.arr\ f \implies T\ f = null$ 
      by (simp add: T.extensionality null-char)
    show  $\bigwedge x\ x'. \llbracket X.ide\ x; X.cong\ x\ x' \rrbracket \implies T\ x = T\ x'$ 
      using T.respects-cong-ide by blast
    show  $\bigwedge f. X.ide\ f \implies src\ (T\ f) = F\ f$ 
      by (metis F.preserves-ide F.preserves-reflects-arr R.arr-resid-iff-con
          R.arr-src-iff-arr R.ide-implies-arr R.resid-arr-src S.con-imp-eq-src
          S.src-ide T.F.preserves-ide T.preserves-src X.con-implies-arr(2)
          X.ideE arr-char assms(4) con-char)
    show  $\bigwedge f. X.ide\ f \implies trg\ (T\ f) = G\ f$ 
      by (simp add: T.preserves-trg arr-char assms(4) trg-char)
    show  $\bigwedge a\ f. a \in X.sources\ f \implies T\ a \setminus F\ f = T\ (X\ a\ f)$ 
      by (metis F.preserves-reflects-arr R.residuation-axioms T.naturality1-ax
          X.arr-iff-has-source X.ex-un-null X.ide-implies-arr X.in-sourcesE
          X.not-arr-null X.null-eqI X.source-is-prfx arr-char assms(4) resid-def
          residuation.conI)
    show  $\bigwedge a\ f. a \in X.sources\ f \implies F\ f \setminus T\ a = G\ f$ 
      by (metis F.preserves-reflects-arr R.arr-resid-iff-con
          T.G.preserves-reflects-arr T.naturality2-ax X.in-sourcesE
          X.residuation-axioms arr-char assms(4) resid-def
          residuation.con-implies-arr(1) residuation.ide-implies-arr)
    show  $\bigwedge a\ f. a \in X.sources\ f \implies join-of\ (T\ a)\ (F\ f)\ (T\ f)$ 
      by (meson F.preserves-reflects-arr T.naturality3 X.con-implies-arr(1)
          X.ide-implies-arr X.in-sourcesE arr-char assms(4) join-of-char)
  qed
  show 1: transformation X resid F G T ..
  show incl  $\circ T = T$ 
    using T.extensionality arr-char assms(4) null-char by fastforce
  qed
end

locale source-replete-sub-rts =
  R: rts R
  for R :: 'a resid (infix  $\setminus_R$  70)
  and Arr :: 'a  $\implies bool$  +
  assumes inclusion: Arr t  $\implies R.arr\ t$ 
  and resid-closed:  $\llbracket Arr\ t; Arr\ u; R.con\ t\ u \rrbracket \implies Arr\ (t \setminus_R\ u)$ 
  and source-replete: Arr t  $\implies R.sources\ t \subseteq Collect\ Arr$ 
  begin

  sublocale sub-rts

```

```

using inclusion resid-closed source-replete
apply unfold-locales
apply auto[2]
by (metis Collect-mem-eq Collect-mono-iff R.con-imp-common-source
      R.sources-eqI R.src-in-sources)

lemma is-sub-rts:
shows sub-rts R Arr
  ..

lemma sources-charSRTS:
shows sources t = {a. Arr t ∧ a ∈ R.sources t}
  using source-replete sources-subset
  apply auto[1]
  by (metis Ball-Collect R.in-sourcesE con-char ide-char in-sourcesI)

lemma targets-charSRTS:
shows targets t = {b. Arr t ∧ b ∈ R.targets t}
proof
  show targets t ⊆ {b. Arr t ∧ b ∈ R.targets t}
    using targets-subset by blast
  show {b. Arr t ∧ b ∈ R.targets t} ⊆ targets t
  proof
    fix b
    assume b: b ∈ {b. Arr t ∧ b ∈ R.targets t}
    show b ∈ targets t
    by (metis (no-types, lifting) R.in-targetsE R.rts-axioms arr-char b
          con-arr-self mem-Collect-eq rts.in-sourcesI sources-charSRTS sources-resid
          trg-char trg-def trg-in-targets)
  qed
qed

interpretation PR: paths-in-rts R
  ..
interpretation P: paths-in-rts resid
  ..

lemma path-reflection:
shows [[PR.Arr T; set T ⊆ Collect Arr]] ⇒ P.Arr T
proof (induct T, simp)
  fix t T
  assume ind: [[PR.Arr T; set T ⊆ Collect Arr]] ⇒ P.Arr T
  assume tT: PR.Arr (t # T)
  assume set: set (t # T) ⊆ Collect Arr
  have 1: R.arr t
  using tT
  by (metis PR.Arr-imp-arr-hd list.sel(1))

```

```

show  $P.Arr (t \# T)$ 
proof (cases  $T = []$ )
  show  $T = [] \implies ?thesis$ 
    using 1 set arr-char by simp
  assume  $T: T \neq []$ 
  show ?thesis
proof
  show arr t
    using 1 arr-char set by simp
  show  $P.Arr T$ 
    using  $T tT P_R.Arr-imp-Arr-tl$ 
    by (metis ind insert-subset list.sel(3) list.simps(15) set)
  show  $targets\ t \subseteq P.Srcs\ T$ 
proof -
  have  $targets\ t \subseteq R.targets\ t$ 
    using targets-subset by blast
  also have  $\dots \subseteq R.sources\ (hd\ T)$ 
    using  $T tT$ 
    by (metis P_R.Arr.simps(3) P_R.Srcs-simp_P list.collapse)
  also have  $\dots \subseteq P.Srcs\ T$ 
    using  $P.Arr-imp-arr-hd P.Srcs-simp_P \langle P.Arr\ T \rangle sources-char_{SRTS}\ arr-char$ 
    by force
  finally show ?thesis by blast
qed
qed
qed
qed
end

locale sub-rts-of-weakly-extensional-rts =
  R: weakly-extensional-rts R +
  sub-rts R Arr
for  $R :: 'a\ resid$  (infix  $\setminus_R\ 70$ )
and  $Arr :: 'a \Rightarrow bool$ 
begin

  sublocale weakly-extensional-rts resid
    using R.weakly-extensional-rts-axioms preserves-weakly-extensional-rts
    by blast

  lemma is-weakly-extensional-rts:
  shows weakly-extensional-rts resid
  ..

  lemma src-char:
  shows  $src = (\lambda t. \text{if } arr\ t \text{ then } R.src\ t \text{ else } null)$ 
proof
  fix  $t$ 

```

```

show src t = (if arr t then R.src t else null)
  by (metis R.src-eqI con-arr-src(2) con-char ide-char ide-src src-def)
qed

```

```

lemma targets-char:
assumes arr t
shows targets t = {R.trg t}
  using assms trg-char trg-in-targets arr-has-un-target by auto

```

**end**

```

locale sub-rts-of-extensional-rts =
  R: extensional-rts R +
  sub-rts R Arr
for R :: 'a resid (infix \_R 70)
and Arr :: 'a ⇒ bool
begin

```

```

  sublocale sub-rts-of-weakly-extensional-rts ..

```

```

  sublocale extensional-rts resid
    using R.extensional-rts-axioms preserves-extensional-rts
    by blast

```

```

  lemma is-extensional-rts:
shows extensional-rts resid
  ..

```

**end**

Here we justify the terminology “normal sub-RTS”, which was introduced earlier, by showing that a normal sub-RTS really is a sub-RTS.

```

lemma (in normal-sub-rts) is-sub-rts:
shows source-replete-sub-rts resid (λt. t ∈ ℳ)
  using elements-are-arr ide-closed
  apply unfold-locales
  apply blast
  apply (meson R.con-def R.con-imp-coinitial R.con-sym-ax forward-stable)
  by blast

```

**end**

## Chapter 3

# The Lambda Calculus

In this second part of the article, we apply the residuated transition system framework developed in the first part to the theory of reductions in Church’s  $\lambda$ -calculus. The underlying idea is to exhibit  $\lambda$ -terms as states (identities) of an RTS, with reduction steps as non-identity transitions. We represent both states and transitions in a unified, variable-free syntax based on de Bruijn indices. A difficulty one faces in regarding the  $\lambda$ -calculus as an RTS is that “elementary reductions”, in which just one redex is contracted, are not preserved by residuation: an elementary reduction can have zero or more residuals along another elementary reduction. However, “parallel reductions”, which permit the contraction of multiple redexes existing in a term to be contracted in a single step, are preserved by residuation. For this reason, in our syntax each term represents a parallel reduction of zero or more redexes; a parallel reduction of zero redexes representing an identity. We have syntactic constructors for variables,  $\lambda$ -abstractions, and applications. An additional constructor represents a  $\beta$ -redex that has been marked for contraction. This is a slightly different approach than that taken by other authors (*e.g.* [1] or [7]), in which it is the application constructor that is marked to indicate a redex to be contracted, but it seems more natural in the present setting in which a single syntax is used to represent both terms and reductions.

Once the syntax has been defined, we define the residuation operation and prove that it satisfies the conditions for a weakly extensional RTS. In this RTS, the source of a term is obtained by “erasing” the markings on redexes, leaving an identity term. The target of a term is the contractum of the parallel reduction it represents. As the definition of residuation involves the use of substitution, a necessary prerequisite is to develop the theory of substitution using de Bruijn indices. In addition, various properties concerning the commutation of residuation and substitution have to be proved. This part of the work has benefited greatly from previous work of Huet [7], in which the theory of residuation was formalized in the proof assistant Coq. In particular, it was very helpful to have already available known-correct statements of various lemmas regarding indices, substitution, and residuation. The development of the theory culminates in the proof of Lévy’s “Cube Lemma” [8], which is the key axiom in the definition of RTS.

Once reductions in the  $\lambda$ -calculus have been cast as transitions of an RTS, we are

able to take advantage of generic results already proved for RTS's; in particular, the construction of the RTS of paths, which represent reduction sequences. Very little additional effort is required at this point to prove the Church-Rosser Theorem. Then, after proving a series of miscellaneous lemmas about reduction paths, we turn to the study of developments. A development of a term is a reduction path from that term in which the only redexes that are contracted are those that are residuals of redexes in the original term. We prove the Finite Developments Theorem: all developments are finite. The proof given here follows that given by de Vrijer [5], except that here we make the adaptations necessary for a syntax based on de Bruijn indices, rather than the classical named-variable syntax used by de Vrijer. Using the Finite Developments Theorem, we define a function that takes a term and constructs a “complete development” of that term, which is a development in which no residuals of original redexes remain to be contracted.

We then turn our attention to “standard reduction paths”, which are reduction paths in which redexes are contracted in a left-to-right order, perhaps with some skips. After giving a definition of standard reduction paths, we define a function that takes a term and constructs a complete development that is also standard. Using this function as a base case, we then define a function that takes an arbitrary parallel reduction path and transforms it into a standard reduction path that is congruent to the given path. The algorithm used is roughly analogous to insertion sort. We use this function to prove strong form of the Standardization Theorem: every reduction path is congruent to a standard reduction path. As a corollary of the Standardization Theorem, we prove the Leftmost Reduction Theorem: leftmost reduction is a normalizing reduction strategy.

It should be noted that, in this article, we consider only the  $\lambda\beta$ -calculus. In the early stages of this work, I made an exploratory attempt to incorporate  $\eta$ -reduction as well, but after encountering some unanticipated difficulties I decided not to attempt that extension until the  $\beta$ -only case had been well-developed.

```
theory LambdaCalculus
imports Main ResiduatedTransitionSystem
begin
```

### 3.1 Syntax

```
locale lambda-calculus
begin
```

The syntax of terms has constructors *Var* for variables, *Lam* for  $\lambda$ -abstraction, and *App* for application. In addition, there is a constructor *Beta* which is used to represent a  $\beta$ -redex that has been marked for contraction. The idea is that a term *Beta t u* represents a marked version of the term *App (Lam t) u*. Finally, there is a constructor *Nil* which is used to represent the null element required for the residuation operation.

```
datatype (discs-sels) lambda =
  Nil
| Var nat
| Lam lambda
| App lambda lambda
```

| *Beta lambda lambda*

The following notation renders  $Beta\ t\ u$  as a “marked” version of  $App\ (Lam\ t)\ u$ , even though the former is a single constructor, whereas the latter contains two constructors.

**notation**  $Nil$  ( $\langle \# \rangle$ )  
**notation**  $Var$  ( $\langle \langle - \rangle \rangle$ )  
**notation**  $Lam$  ( $\langle \lambda[-] \rangle$ )  
**notation**  $App$  (**infixl**  $\langle \circ \rangle$  55)  
**notation**  $Beta$  ( $\langle (\lambda[-] \bullet -) \rangle$  [55, 56] 55)

The following function computes the set of free variables of a term. Note that since variables are represented by numeric indices, this is a set of numbers.

```
fun  $FV$ 
where  $FV\ \# = \{\}$ 
  |  $FV\ \langle i \rangle = \{i\}$ 
  |  $FV\ \lambda[t] = (\lambda n. n - 1) \cdot (FV\ t - \{0\})$ 
  |  $FV\ (t \circ u) = FV\ t \cup FV\ u$ 
  |  $FV\ (\lambda[t] \bullet u) = (\lambda n. n - 1) \cdot (FV\ t - \{0\}) \cup FV\ u$ 
```

### 3.1.1 Some Orderings for Induction

We will need to do some simultaneous inductions on pairs and triples of subterms of given terms. We prove the well-foundedness of the associated relations using the following size measure.

```
fun  $size :: lambda \Rightarrow nat$ 
where  $size\ \# = 0$ 
  |  $size\ \langle - \rangle = 1$ 
  |  $size\ \lambda[t] = size\ t + 1$ 
  |  $size\ (t \circ u) = size\ t + size\ u + 1$ 
  |  $size\ (\lambda[t] \bullet u) = (size\ t + 1) + size\ u + 1$ 
```

**lemma** *wf-if-img-lt:*

**fixes**  $r :: ('a * 'a)\ set$  **and**  $f :: 'a \Rightarrow nat$

**assumes**  $\bigwedge x\ y. (x, y) \in r \implies f\ x < f\ y$

**shows**  $wf\ r$

**using** *assms*

**by** (*metis in-measure wf-iff-no-infinite-down-chain wf-measure*)

**inductive** *subterm*

**where**  $\bigwedge t. subterm\ t\ \lambda[t]$

|  $\bigwedge t\ u. subterm\ t\ (t \circ u)$

|  $\bigwedge t\ u. subterm\ u\ (t \circ u)$

|  $\bigwedge t\ u. subterm\ t\ (\lambda[t] \bullet u)$

|  $\bigwedge t\ u. subterm\ u\ (\lambda[t] \bullet u)$

|  $\bigwedge t\ u\ v. [subterm\ t\ u; subterm\ u\ v] \implies subterm\ t\ v$

**lemma** *subterm-implies-smaller:*

**shows**  $subterm\ t\ u \implies size\ t < size\ u$

by (*induct rule: subterm.induct*) *auto*

**abbreviation** *subterm-rel*

**where** *subterm-rel*  $\equiv \{(t, u). \text{subterm } t \ u\}$

**lemma** *wf-subterm-rel:*

**shows** *wf subterm-rel*

**using** *subterm-implies-smaller wf-if-img-lt*

**by** (*metis case-prod-conv mem-Collect-eq*)

**abbreviation** *subterm-pair-rel*

**where** *subterm-pair-rel*  $\equiv \{((t1, t2), u1, u2). \text{subterm } t1 \ u1 \ \wedge \ \text{subterm } t2 \ u2\}$

**lemma** *wf-subterm-pair-rel:*

**shows** *wf subterm-pair-rel*

**using** *subterm-implies-smaller*

*wf-if-img-lt [of subterm-pair-rel  $\lambda(t1, t2). \text{max (size } t1) \ (\text{size } t2)$ ]*

**by** *fastforce*

**abbreviation** *subterm-triple-rel*

**where** *subterm-triple-rel*  $\equiv$

$\{((t1, t2, t3), u1, u2, u3). \text{subterm } t1 \ u1 \ \wedge \ \text{subterm } t2 \ u2 \ \wedge \ \text{subterm } t3 \ u3\}$

**lemma** *wf-subterm-triple-rel:*

**shows** *wf subterm-triple-rel*

**using** *subterm-implies-smaller*

*wf-if-img-lt [of subterm-triple-rel*

$\lambda(t1, t2, t3). \text{max (max (size } t1) \ (\text{size } t2)) \ (\text{size } t3)$ ]

**by** *fastforce*

**lemma** *subterm-lemmas:*

**shows** *subterm t  $\lambda[t]$*

**and** *subterm t ( $\lambda[t] \circ u$ )  $\wedge$  subterm u ( $\lambda[t] \circ u$ )*

**and** *subterm t ( $t \circ u$ )  $\wedge$  subterm u ( $t \circ u$ )*

**and** *subterm t ( $\lambda[t] \bullet u$ )  $\wedge$  subterm u ( $\lambda[t] \bullet u$ )*

**by** (*metis subterm.simps*)<sup>+</sup>

### 3.1.2 Arrows and Identities

Here we define some special classes of terms. An “arrow” is a term that contains no occurrences of *Nil*. An “identity” is an arrow that contains no occurrences of *Beta*. It will be important for the commutation of substitution and residuation later on that substitution not be used in a way that could create any marked redexes; for example, we don’t want the substitution of *Lam* (*Var* *0*) for *Var* *0* in an application *App* (*Var* *0*) (*Var* *0*) to create a new “marked” redex. The use of the separate constructor *Beta* for marked redexes automatically avoids this.

**fun** *Arr*

**where** *Arr*  $\# = \text{False}$

```

| Arr «-» = True
| Arr λ[t] = Arr t
| Arr (t ◦ u) = (Arr t ∧ Arr u)
| Arr (λ[t] • u) = (Arr t ∧ Arr u)

```

**lemma** *Arr-not-Nil*:  
**assumes** *Arr t*  
**shows**  $t \neq \#$   
**using** *assms* **by** *auto*

```

fun Ide
where Ide # = False
| Ide «-» = True
| Ide λ[t] = Ide t
| Ide (t ◦ u) = (Ide t ∧ Ide u)
| Ide (λ[t] • u) = False

```

**lemma** *Ide-implies-Arr*:  
**shows**  $Ide\ t \implies Arr\ t$   
**by** (*induct t*) *auto*

**lemma** *ArrE [elim]*:  
**assumes** *Arr t*  
**and**  $\bigwedge i. t = \langle i \rangle \implies T$   
**and**  $\bigwedge u. t = \lambda[u] \implies T$   
**and**  $\bigwedge u\ v. t = u \circ v \implies T$   
**and**  $\bigwedge u\ v. t = \lambda[u] \bullet v \implies T$   
**shows**  $T$   
**using** *assms*  
**by** (*cases t*) *auto*

### 3.1.3 Raising Indices

For substitution, we need to be able to raise the indices of all free variables in a subterm by a specified amount. To do this recursively, we need to keep track of the depth of nesting of  $\lambda$ 's and only raise the indices of variables that are already greater than or equal to that depth, as these are the variables that are free in the current context. This leads to defining a function *Raise* that has two arguments: the depth threshold  $d$  and the increment  $n$  to be added to indices above that threshold.

```

fun Raise
where Raise - - # = #
| Raise d n «i» = (if  $i \geq d$  then « $i+n$ » else « $i$ »)
| Raise d n λ[t] = λ[Raise (Suc d) n t]
| Raise d n (t ◦ u) = Raise d n t ◦ Raise d n u
| Raise d n (λ[t] • u) = λ[Raise (Suc d) n t] • Raise d n u

```

Ultimately, the definition of substitution will only directly involve the function that raises all indices of variables that are free in the outermost context; in a term, so we introduce an abbreviation for this special case.

**abbreviation** *raise*  
**where** *raise* == *Raise 0*

**lemma** *size-Raise*:  
**shows**  $\bigwedge d. \text{size } (\text{Raise } d \ n \ t) = \text{size } t$   
**by** (*induct t*) *auto*

**lemma** *Raise-not-Nil*:  
**assumes**  $t \neq \#$   
**shows**  $\text{Raise } d \ n \ t \neq \#$   
**using** *assms*  
**by** (*cases t*) *auto*

**lemma** *FV-Raise*:  
**shows**  $FV (\text{Raise } d \ n \ t) = (\lambda x. \text{if } x \geq d \text{ then } x + n \text{ else } x) \text{ ` } FV \ t$   
**apply** (*induct t arbitrary: d n*)  
**apply** *auto[3]*  
**apply** *force*  
**apply** *force*  
**apply** *force*  
**apply** *force*  
**apply** *force*

**proof** –  
**fix**  $t \ u \ d \ n$   
**assume** *ind1*:  $\bigwedge d \ n. FV (\text{Raise } d \ n \ t) = (\lambda x. \text{if } d \leq x \text{ then } x + n \text{ else } x) \text{ ` } FV \ t$   
**assume** *ind2*:  $\bigwedge d \ n. FV (\text{Raise } d \ n \ u) = (\lambda x. \text{if } d \leq x \text{ then } x + n \text{ else } x) \text{ ` } FV \ u$   
**have**  $FV (\text{Raise } d \ n \ (\lambda[t] \bullet u)) =$   
 $(\lambda x. x - \text{Suc } 0) \text{ ` } ((\lambda x. x + n) \text{ ` } ($   
 $(FV \ t \cap \{x. \text{Suc } d \leq x\}) \cup FV \ t \cap \{x. \neg \text{Suc } d \leq x\} - \{0\}) \cup$   
 $((\lambda x. x + n) \text{ ` } (FV \ u \cap \{x. d \leq x\}) \cup FV \ u \cap \{x. \neg d \leq x\}))$   
**using** *ind1 ind2* **by** *simp*  
**also have**  $\dots = (\lambda x. \text{if } d \leq x \text{ then } x + n \text{ else } x) \text{ ` } FV (\lambda[t] \bullet u)$   
**by** *auto force+*  
**finally show**  $FV (\text{Raise } d \ n \ (\lambda[t] \bullet u)) =$   
 $(\lambda x. \text{if } d \leq x \text{ then } x + n \text{ else } x) \text{ ` } FV (\lambda[t] \bullet u)$   
**by** *blast*

**qed**

**lemma** *Arr-Raise*:  
**shows**  $\text{Arr } t \longleftrightarrow \text{Arr } (\text{Raise } d \ n \ t)$   
**using** *FV-Raise*  
**by** (*induct t arbitrary: d n*) *auto*

**lemma** *Ide-Raise*:  
**shows**  $\text{Ide } t \longleftrightarrow \text{Ide } (\text{Raise } d \ n \ t)$   
**by** (*induct t arbitrary: d n*) *auto*

**lemma** *Raise-0*:  
**shows**  $\text{Raise } d \ 0 \ t = t$

by (induct t arbitrary: d) auto

**lemma** *Raise-Suc*:

**shows**  $\text{Raise } d \text{ (Suc } n) t = \text{Raise } d \ 1 \ (\text{Raise } d \ n \ t)$

by (induct t arbitrary: d n) auto

**lemma** *Raise-Var*:

**shows**  $\text{Raise } d \ n \ \langle i \rangle = \langle \text{if } i < d \text{ then } i \text{ else } i + n \rangle$

by auto

The following development of the properties of raising indices, substitution, and residuation has benefited greatly from the previous work by Huet [7]. In particular, it was very helpful to have correct statements of various lemmas available, rather than having to reconstruct them.

**lemma** *Raise-plus*:

**shows**  $\text{Raise } d \ (m + n) \ t = \text{Raise } (d + m) \ n \ (\text{Raise } d \ m \ t)$

by (induct t arbitrary: d m n) auto

**lemma** *Raise-plus'*:

**shows**  $\llbracket d' \leq d + n; d \leq d' \rrbracket \implies \text{Raise } d \ (m + n) \ t = \text{Raise } d' \ m \ (\text{Raise } d \ n \ t)$

by (induct t arbitrary: n m d d') auto

**lemma** *Raise-Raise*:

**shows**  $i \leq n \implies \text{Raise } i \ p \ (\text{Raise } n \ k \ t) = \text{Raise } (p + n) \ k \ (\text{Raise } i \ p \ t)$

by (induct t arbitrary: i k n p) auto

**lemma** *raise-plus*:

**shows**  $d \leq n \implies \text{raise } (m + n) \ t = \text{Raise } d \ m \ (\text{raise } n \ t)$

using *Raise-plus'* by auto

**lemma** *raise-Raise*:

**shows**  $\text{raise } p \ (\text{Raise } n \ k \ t) = \text{Raise } (p + n) \ k \ (\text{raise } p \ t)$

by (simp add: *Raise-Raise*)

**lemma** *Raise-inj*:

**shows**  $\text{Raise } d \ n \ t = \text{Raise } d \ n \ u \implies t = u$

**proof** (induct t arbitrary: d n u)

show  $\bigwedge d \ n \ u. \text{Raise } d \ n \ \# = \text{Raise } d \ n \ u \implies \# = u$

by (metis *Raise.simps(1)* *Raise-not-Nil*)

show  $\bigwedge x \ d \ n. \text{Raise } d \ n \ \langle x \rangle = \text{Raise } d \ n \ u \implies \langle x \rangle = u$  **for**  $u$

using *Raise-Var*

apply (cases u, auto)

by (metis *add-lessD1* *add-right-imp-eq*)

show  $\bigwedge t \ d \ n. \llbracket \bigwedge d \ n \ u'. \text{Raise } d \ n \ t = \text{Raise } d \ n \ u' \implies t = u';$

$\text{Raise } d \ n \ \lambda[t] = \text{Raise } d \ n \ u \rrbracket$

$\implies \lambda[t] = u$

**for**  $u$

apply (cases u, auto)

by (metis *lambda.distinct(9)*)

```

show  $\bigwedge t1\ t2\ d\ n. \llbracket \bigwedge d\ n\ u'. \text{Raise } d\ n\ t1 = \text{Raise } d\ n\ u' \implies t1 = u';$ 
 $\bigwedge d\ n\ u'. \text{Raise } d\ n\ t2 = \text{Raise } d\ n\ u' \implies t2 = u';$ 
 $\text{Raise } d\ n\ (t1 \circ t2) = \text{Raise } d\ n\ u \rrbracket$ 
 $\implies t1 \circ t2 = u$ 

for  $u$ 
apply (cases  $u$ , auto)
by (metis lambda.distinct(11))
show  $\bigwedge t1\ t2\ d\ n. \llbracket \bigwedge d\ n\ u'. \text{Raise } d\ n\ t1 = \text{Raise } d\ n\ u' \implies t1 = u';$ 
 $\bigwedge d\ n\ u'. \text{Raise } d\ n\ t2 = \text{Raise } d\ n\ u' \implies t2 = u';$ 
 $\text{Raise } d\ n\ (\lambda[t1] \bullet t2) = \text{Raise } d\ n\ u \rrbracket$ 
 $\implies \lambda[t1] \bullet t2 = u$ 

for  $u$ 
apply (cases  $u$ , auto)
by (metis lambda.distinct(13))
qed

```

### 3.1.4 Substitution

Following [7], we now define a generalized substitution operation with adjustment of indices. The ultimate goal is to define the result of contraction of a marked redex  $Beta\ t\ u$  to be  $subst\ u\ t$ . However, to be able to give a proper recursive definition of  $subst$ , we need to introduce a parameter  $n$  to keep track of the depth of nesting of  $Lam$ 's as we descend into the term  $t$ . So, instead of  $subst\ u\ t$  simply substituting  $u$  for occurrences of  $Var\ 0$ ,  $Subst\ n\ u\ t$  will be substituting for occurrences of  $Var\ n$ , and the term  $u$  will have the indices of its free variables raised by  $n$  before replacing  $Var\ n$ . In addition, any variables in  $t$  that have indices greater than  $n$  will have these indices lowered by one, to account for the outermost  $Lam$  that is being removed by the contraction. We can then define  $subst\ u\ t$  to be  $Subst\ 0\ u\ t$ .

```

fun Subst
where Subst - -  $\# = \#$ 
| Subst  $n\ v\ \langle i \rangle =$  (if  $n < i$  then  $\langle i-1 \rangle$  else if  $n = i$  then raise  $n\ v$  else  $\langle i \rangle$ )
| Subst  $n\ v\ \lambda[t] = \lambda[Subst\ (Suc\ n)\ v\ t]$ 
| Subst  $n\ v\ (t \circ u) = Subst\ n\ v\ t \circ Subst\ n\ v\ u$ 
| Subst  $n\ v\ (\lambda[t] \bullet u) = \lambda[Subst\ (Suc\ n)\ v\ t] \bullet Subst\ n\ v\ u$ 

```

```

abbreviation subst
where subst  $\equiv Subst\ 0$ 

```

```

lemma Subst-Nil:
shows Subst  $n\ v\ \# = \#$ 
by (cases  $v = \#$ ) auto

```

```

lemma Subst-not-Nil:
assumes  $v \neq \#$  and  $t \neq \#$ 
shows  $t \neq \# \implies Subst\ n\ v\ t \neq \#$ 
using assms Raise-not-Nil
by (induct  $t$ ) auto

```

The following expression summarizes how the set of free variables of a term  $Subst\ d\ u\ t$ , obtained by substituting  $u$  into  $t$  at depth  $d$ , relates to the sets of free variables of  $t$  and  $u$ . This expression is not used in the subsequent formal development, but it has been left here as an aid to understanding.

**abbreviation**  $FVS$

**where**  $FVS\ d\ v\ t \equiv (FV\ t \cap \{x. x < d\}) \cup$   
 $(\lambda x. x - 1) \text{ ' } \{x. x > d \wedge x \in FV\ t\} \cup$   
 $(\lambda x. x + d) \text{ ' } \{x. d \in FV\ t \wedge x \in FV\ v\}$

**lemma**  $FV\text{-}Subst$ :

**shows**  $FV\ (Subst\ d\ v\ t) = FVS\ d\ v\ t$

**proof** (*induct t arbitrary: d v*)

**have**  $\bigwedge d\ t\ v. (\lambda x. x - 1) \text{ ' } (FVS\ (Suc\ d)\ v\ t - \{0\}) = FVS\ d\ v\ \lambda[t]$

**proof** –

**fix**  $d\ t\ v$

**have**  $FVS\ d\ v\ \lambda[t] =$

$(\lambda x. x - Suc\ 0) \text{ ' } (FV\ t - \{0\}) \cap \{x. x < d\} \cup$   
 $(\lambda x. x - Suc\ 0) \text{ ' } \{x. d < x \wedge x \in (\lambda x. x - Suc\ 0) \text{ ' } (FV\ t - \{0\})\} \cup$   
 $(\lambda x. x + d) \text{ ' } \{x. d \in (\lambda x. x - Suc\ 0) \text{ ' } (FV\ t - \{0\}) \wedge x \in FV\ v\}$

**by** *simp*

**also have**  $\dots = (\lambda x. x - 1) \text{ ' } (FVS\ (Suc\ d)\ v\ t - \{0\})$

**by** *auto force+*

**finally show**  $(\lambda x. x - 1) \text{ ' } (FVS\ (Suc\ d)\ v\ t - \{0\}) = FVS\ d\ v\ \lambda[t]$

**by** *metis*

**qed**

**thus**  $\bigwedge d\ t\ v. (\bigwedge d\ v. FV\ (Subst\ d\ v\ t) = FVS\ d\ v\ t)$   
 $\implies FV\ (Subst\ d\ v\ \lambda[t]) = FVS\ d\ v\ \lambda[t]$

**by** *simp*

**have**  $\bigwedge t\ u\ v\ d. (\lambda x. x - 1) \text{ ' } (FVS\ (Suc\ d)\ v\ t - \{0\}) \cup FVS\ d\ v\ u = FVS\ d\ v\ (\lambda[t] \bullet u)$

**proof** –

**fix**  $t\ u\ v\ d$

**have**  $FVS\ d\ v\ (\lambda[t] \bullet u) =$

$((\lambda x. x - Suc\ 0) \text{ ' } (FV\ t - \{0\}) \cup FV\ u) \cap \{x. x < d\} \cup$   
 $(\lambda x. x - Suc\ 0) \text{ ' } \{x. d < x \wedge (x \in (\lambda x. x - Suc\ 0) \text{ ' } (FV\ t - \{0\}) \vee x \in FV\ u)\} \cup$   
 $(\lambda x. x + d) \text{ ' } \{x. (d \in (\lambda x. x - Suc\ 0) \text{ ' } (FV\ t - \{0\}) \vee d \in FV\ u) \wedge x \in FV\ v\}$

**by** *simp*

**also have**  $\dots = (\lambda x. x - 1) \text{ ' } (FVS\ (Suc\ d)\ v\ t - \{0\}) \cup FVS\ d\ v\ u$

**by** *force*

**finally show**  $(\lambda x. x - 1) \text{ ' } (FVS\ (Suc\ d)\ v\ t - \{0\}) \cup FVS\ d\ v\ u = FVS\ d\ v\ (\lambda[t] \bullet u)$

**by** *metis*

**qed**

**thus**  $\bigwedge t\ u\ v\ d. [\bigwedge d\ v. FV\ (Subst\ d\ v\ t) = FVS\ d\ v\ t;$   
 $\bigwedge d\ v. FV\ (Subst\ d\ v\ u) = FVS\ d\ v\ u]$   
 $\implies FV\ (Subst\ d\ v\ (\lambda[t] \bullet u)) = FVS\ d\ v\ (\lambda[t] \bullet u)$

**by** *simp*

**qed** (*auto simp add: FV-Raise*)

**lemma**  $Arr\text{-}Subst$ :

**assumes**  $Arr\ v$

**shows**  $Arr\ t \implies Arr\ (Subst\ n\ v\ t)$   
**using** *assms Arr-Raise FV-Subst*  
**by** (*induct t arbitrary: n*) *auto*

**lemma** *vacuous-Subst*:  
**shows**  $\llbracket Arr\ v; i \notin FV\ t \rrbracket \implies Raise\ i\ 1\ (Subst\ i\ v\ t) = t$   
**apply** (*induct t arbitrary: i v, auto*)  
**by** *force+*

**lemma** *Ide-Subst-iff*:  
**shows**  $Ide\ (Subst\ n\ v\ t) \iff Ide\ t \wedge (n \in FV\ t \longrightarrow Ide\ v)$   
**using** *Ide-Raise vacuous-Subst*  
**apply** (*induct t arbitrary: n*)  
**apply** *auto[5]*  
**apply** *fastforce*  
**by** (*metis Diff-empty Diff-insert0 One-nat-def diff-Suc-1 image-iff insertE insert-Diff nat.distinct(1)*)

**lemma** *Ide-Subst*:  
**shows**  $\llbracket Ide\ t; Ide\ v \rrbracket \implies Ide\ (Subst\ n\ v\ t)$   
**using** *Ide-Raise*  
**by** (*induct t arbitrary: n*) *auto*

**lemma** *Raise-Subst*:  
**shows**  $Raise\ (p + n)\ k\ (Subst\ p\ v\ t) = Subst\ p\ (Raise\ n\ k\ v)\ (Raise\ (Suc\ (p + n))\ k\ t)$   
**using** *raise-Raise*  
**apply** (*induct t arbitrary: v k n p, auto*)  
**by** (*metis add-Suc*)**+**

**lemma** *Raise-Subst'*:  
**assumes**  $t \neq \#\$   
**shows**  $\llbracket v \neq \#\; k \leq n \rrbracket \implies Raise\ k\ p\ (Subst\ n\ v\ t) = Subst\ (p + n)\ v\ (Raise\ k\ p\ t)$   
**using** *assms raise-plus*  
**apply** (*induct t arbitrary: v k n p, auto*)  
**apply** (*metis Raise.simps(1) Subst-Nil Suc-le-mono add-Suc-right*)  
**apply** *fastforce*  
**apply** *fastforce*  
**apply** (*metis Raise.simps(1) Subst-Nil Suc-le-mono add-Suc-right*)  
**by** *fastforce*

**lemma** *Raise-subst*:  
**shows**  $Raise\ n\ k\ (subst\ v\ t) = subst\ (Raise\ n\ k\ v)\ (Raise\ (Suc\ n)\ k\ t)$   
**using** *Raise-0*  
**apply** (*induct t arbitrary: v k n, auto*)  
**by** (*metis One-nat-def Raise-Subst plus-1-eq-Suc*)**+**

**lemma** *raise-Subst*:  
**assumes**  $t \neq \#\$   
**shows**  $v \neq \#\ \implies raise\ p\ (Subst\ n\ v\ t) = Subst\ (p + n)\ v\ (raise\ p\ t)$

**using** *assms Raise-plus raise-Raise Raise-Subst'*  
**apply** (*induct t arbitrary: v n p*)  
**by** (*meson zero-le*)+

**lemma** *Subst-Raise:*

**shows**  $\llbracket v \neq \#; d \leq m; m \leq n + d \rrbracket \implies \text{Subst } m \ v \ (\text{Raise } d \ (\text{Suc } n) \ t) = \text{Raise } d \ n \ t$   
**by** (*induct t arbitrary: v m n d*) *auto*

**lemma** *Subst-raise:*

**shows**  $\llbracket v \neq \#; m \leq n \rrbracket \implies \text{Subst } m \ v \ (\text{raise } (\text{Suc } n) \ t) = \text{raise } n \ t$   
**using** *Subst-Raise*  
**by** (*induct t arbitrary: v m n*) *auto*

**lemma** *Subst-Subst:*

**shows**  $\llbracket v \neq \#; w \neq \# \rrbracket \implies$   
 $\text{Subst } (m + n) \ w \ (\text{Subst } m \ v \ t) = \text{Subst } m \ (\text{Subst } n \ w \ v) \ (\text{Subst } (\text{Suc } (m + n)) \ w \ t)$   
**using** *Raise-0 raise-Subst Subst-not-Nil Subst-raise*  
**apply** (*induct t arbitrary: v w m n, auto*)  
**by** (*metis add-Suc*)+

The Substitution Lemma, as given by Huet [7].

**lemma** *substitution-lemma:*

**shows**  $\llbracket v \neq \#; w \neq \# \rrbracket \implies \text{Subst } n \ v \ (\text{subst } w \ t) = \text{subst } (\text{Subst } n \ v \ w) \ (\text{Subst } (\text{Suc } n) \ v \ t)$   
**by** (*metis Subst-Subst add-0*)

## 3.2 Lambda-Calculus as an RTS

### 3.2.1 Residuation

We now define residuation on terms. Residuation is an operation which, when defined for terms  $t$  and  $u$ , produces terms  $t \setminus u$  and  $u \setminus t$  that represent, respectively, what remains of the reductions of  $t$  after performing the reductions in  $u$ , and what remains of the reductions of  $u$  after performing the reductions in  $t$ .

The definition ensures that, if residuation is defined for two terms, then those terms in must be arrows that are *coinitial* (*i.e.* they are the same after erasing marks on redexes). The residual  $t \setminus u$  then has marked redexes at positions corresponding to redexes that were originally marked in  $t$  and that were not contracted by any of the reductions of  $u$ .

This definition has also benefited from the presentation in [7].

**fun** *resid* (**infix**  $\setminus$  70)

**where**  $\llbracket i \rrbracket \setminus \llbracket i' \rrbracket = (\text{if } i = i' \text{ then } \llbracket i \rrbracket \text{ else } \#)$   
 $\lambda[t] \setminus \lambda[t'] = (\text{if } t \setminus t' = \# \text{ then } \# \text{ else } \lambda[t \setminus t'])$   
 $(t \circ u) \setminus (t' \circ u') = (\text{if } t \setminus t' = \# \vee u \setminus u' = \# \text{ then } \# \text{ else } (t \setminus t') \circ (u \setminus u'))$   
 $(\lambda[t] \bullet u) \setminus (\lambda[t'] \bullet u') = (\text{if } t \setminus t' = \# \vee u \setminus u' = \# \text{ then } \# \text{ else } \text{subst } (u \setminus u') \ (t \setminus t'))$   
 $(\lambda[t] \circ u) \setminus (\lambda[t'] \bullet u') = (\text{if } t \setminus t' = \# \vee u \setminus u' = \# \text{ then } \# \text{ else } \text{subst } (u \setminus u') \ (t \setminus t'))$   
 $(\lambda[t] \bullet u) \setminus (\lambda[t'] \circ u') = (\text{if } t \setminus t' = \# \vee u \setminus u' = \# \text{ then } \# \text{ else } \lambda[t \setminus t'] \bullet (u \setminus u'))$   
 $\text{resid } - \ - = \#$

Terms  $t$  and  $u$  are *consistent* if residuation is defined for them.

**abbreviation** *Con* (**infix**  $\langle \frown \rangle$  50)  
**where** *Con*  $t \frown u \equiv \text{resid } t \frown u \neq \#$

**lemma** *ConE* [*elim*]:

**assumes**  $t \frown t'$   
**and**  $\bigwedge i. \llbracket t = \langle i \rangle; t' = \langle i \rangle; \text{resid } t \frown t' = \langle i \rangle \rrbracket \implies T$   
**and**  $\bigwedge u \ u'. \llbracket t = \lambda[u]; t' = \lambda[u']; u \frown u'; t \setminus t' = \lambda[u \setminus u'] \rrbracket \implies T$   
**and**  $\bigwedge u \ v \ u' \ v'. \llbracket t = u \circ v; t' = u' \circ v'; u \frown u'; v \frown v';$   
 $t \setminus t' = (u \setminus u') \circ (v \setminus v') \rrbracket \implies T$   
**and**  $\bigwedge u \ v \ u' \ v'. \llbracket t = \lambda[u] \bullet v; t' = \lambda[u'] \bullet v'; u \frown u'; v \frown v';$   
 $t \setminus t' = \text{subst } (v \setminus v') (u \setminus u') \rrbracket \implies T$   
**and**  $\bigwedge u \ v \ u' \ v'. \llbracket t = \lambda[u] \circ v; t' = \text{Beta } u' \ v'; u \frown u'; v \frown v';$   
 $t \setminus t' = \text{subst } (v \setminus v') (u \setminus u') \rrbracket \implies T$   
**and**  $\bigwedge u \ v \ u' \ v'. \llbracket t = \lambda[u] \bullet v; t' = \lambda[u'] \circ v'; u \frown u'; v \frown v';$   
 $t \setminus t' = \lambda[u \setminus u'] \bullet (v \setminus v') \rrbracket \implies T$

**shows**  $T$

**using** *assms*  
**apply** (*cases*  $t$ ; *cases*  $t'$ )  
**apply** *simp-all*  
**apply** *metis*  
**apply** *metis*  
**apply** *metis*  
**apply** (*cases un-App1*  $t$ , *simp-all*)  
**apply** *metis*  
**apply** (*cases un-App1*  $t'$ , *simp-all*)  
**apply** *metis*  
**by** *metis*

A term can only be consistent with another if both terms are “arrows”.

**lemma** *Con-implies-Arr1*:

**shows**  $t \frown u \implies \text{Arr } t$

**proof** (*induct*  $t$  *arbitrary*:  $u$ )

**fix**  $u \ v \ t'$   
**assume** *ind1*:  $\bigwedge u'. u \frown u' \implies \text{Arr } u$   
**assume** *ind2*:  $\bigwedge v'. v \frown v' \implies \text{Arr } v$   
**show**  $u \circ v \frown t' \implies \text{Arr } (u \circ v)$   
**using** *ind1 ind2*  
**apply** (*cases*  $t'$ , *simp-all*)  
**apply** *metis*  
**apply** (*cases*  $u$ , *simp-all*)  
**by** (*metis lambda.distinct*(3) *resid.simps*(2))  
**show**  $\lambda[u] \bullet v \frown t' \implies \text{Arr } (\lambda[u] \bullet v)$   
**using** *ind1 ind2*  
**apply** (*cases*  $t'$ , *simp-all*)  
**apply** (*cases un-App1*  $t'$ , *simp-all*)  
**by** *metis+*

**qed** *auto*

**lemma** *Con-implies-Arr2*:

```

shows  $t \frown u \implies \text{Arr } u$ 
proof (induct u arbitrary: t)
  fix u' v' t
  assume ind1:  $\bigwedge u. u \frown u' \implies \text{Arr } u'$ 
  assume ind2:  $\bigwedge v. v \frown v' \implies \text{Arr } v'$ 
  show  $t \frown u' \circ v' \implies \text{Arr } (u' \circ v')$ 
  using ind1 ind2
  apply (cases t, simp-all)
  apply metis
  apply (cases u', simp-all)
  by (metis lambda.distinct(3) resid.simps(2))
  show  $t \frown (\lambda[u\eta] \bullet v') \implies \text{Arr } (\lambda[u\eta] \bullet v')$ 
  using ind1 ind2
  apply (cases t, simp-all)
  apply (cases un-App1 t, simp-all)
  by metis+
qed auto

```

```

lemma ConD:
shows  $t \circ u \frown t' \circ u' \implies t \frown t' \wedge u \frown u'$ 
and  $\lambda[v] \bullet u \frown \lambda[v'] \bullet u' \implies \lambda[v] \frown \lambda[v'] \wedge u \frown u'$ 
and  $\lambda[v] \bullet u \frown t' \circ u' \implies \lambda[v] \frown t' \wedge u \frown u'$ 
and  $t \circ u \frown \lambda[v'] \bullet u' \implies t \frown \lambda[v'] \wedge u \frown u'$ 
by auto

```

Residuation on consistent terms preserves arrows.

```

lemma Arr-resid:
shows  $t \frown u \implies \text{Arr } (t \setminus u)$ 
proof (induct t arbitrary: u)
  fix t1 t2 u
  assume ind1:  $\bigwedge u. t1 \frown u \implies \text{Arr } (t1 \setminus u)$ 
  assume ind2:  $\bigwedge u. t2 \frown u \implies \text{Arr } (t2 \setminus u)$ 
  show  $t1 \circ t2 \frown u \implies \text{Arr } ((t1 \circ t2) \setminus u)$ 
  using ind1 ind2 Arr-Subst
  apply (cases u, auto)
  apply (cases t1, auto)
  by (metis Arr.simps(3) ConD(2) resid.simps(2) resid.simps(4))
  show  $\lambda[t1] \bullet t2 \frown u \implies \text{Arr } ((\lambda[t1] \bullet t2) \setminus u)$ 
  using ind1 ind2 Arr-Subst
  by (cases u) auto
qed auto

```

### 3.2.2 Source and Target

Here we give syntactic versions of the *source* and *target* of a term. These will later be shown to agree (on arrows) with the versions derived from the residuation. The underlying idea here is that a term stands for a reduction sequence in which all marked redexes (corresponding to instances of the constructor *Beta*) are contracted in a bottom-up fashion. A term without any marked redexes stands for an empty reduction sequence;

such terms will be shown to be the identities derived from the residuation. The source of term is the identity obtained by erasing all markings; that is, by replacing all subterms of the form  $Beta\ t\ u$  by  $App\ (Lam\ t)\ u$ . The target of a term is the identity that is the result of contracting all the marked redexes.

```

fun Src
where Src  $\# = \#$ 
  | Src  $\langle\langle i \rangle\rangle = \langle\langle i \rangle\rangle$ 
  | Src  $\lambda[t] = \lambda[Src\ t]$ 
  | Src  $(t \circ u) = Src\ t \circ Src\ u$ 
  | Src  $(\lambda[t] \bullet u) = \lambda[Src\ t] \circ Src\ u$ 

```

```

fun Trg
where Trg  $\langle\langle i \rangle\rangle = \langle\langle i \rangle\rangle$ 
  | Trg  $\lambda[t] = \lambda[Trg\ t]$ 
  | Trg  $(t \circ u) = Trg\ t \circ Trg\ u$ 
  | Trg  $(\lambda[t] \bullet u) = subst\ (Trg\ u)\ (Trg\ t)$ 
  | Trg  $- = \#$ 

```

```

lemma Ide-Src:
shows Arr  $t \implies Ide\ (Src\ t)$ 
  by (induct  $t$ ) auto

```

```

lemma Ide-iff-Src-self:
assumes Arr  $t$ 
shows Ide  $t \longleftrightarrow Src\ t = t$ 
  using assms Ide-Src
  by (induct  $t$ ) auto

```

```

lemma Arr-Src [simp]:
assumes Arr  $t$ 
shows Arr  $(Src\ t)$ 
  using assms Ide-Src Ide-implies-Arr by blast

```

```

lemma Con-Src:
shows  $\llbracket size\ t + size\ u \leq n; t \frown u \rrbracket \implies Src\ t \frown Src\ u$ 
  by (induct  $n$  arbitrary:  $t\ u$ ) auto

```

```

lemma Src-eq-iff:
shows Src  $\langle\langle i \rangle\rangle = Src\ \langle\langle i' \rangle\rangle \longleftrightarrow i = i'$ 
and Src  $(t \circ u) = Src\ (t' \circ u') \longleftrightarrow Src\ t = Src\ t' \wedge Src\ u = Src\ u'$ 
and Src  $(\lambda[t] \bullet u) = Src\ (\lambda[t'] \bullet u') \longleftrightarrow Src\ t = Src\ t' \wedge Src\ u = Src\ u'$ 
and Src  $(\lambda[t] \circ u) = Src\ (\lambda[t'] \bullet u') \longleftrightarrow Src\ t = Src\ t' \wedge Src\ u = Src\ u'$ 
  by auto

```

```

lemma Src-Raise:
shows Src  $(Raise\ d\ n\ t) = Raise\ d\ n\ (Src\ t)$ 
  by (induct  $t$  arbitrary:  $d$ ) auto

```

```

lemma Src-Subst [simp]:

```

**shows**  $\llbracket \text{Arr } t; \text{Arr } u \rrbracket \implies \text{Src } (\text{Subst } d \ t \ u) = \text{Subst } d \ (\text{Src } t) \ (\text{Src } u)$   
**using** *Src-Raise*  
**by** (*induct u arbitrary: d X*) *auto*

**lemma** *Ide-Trg*:  
**shows**  $\text{Arr } t \implies \text{Ide } (\text{Trg } t)$   
**using** *Ide-Subst*  
**by** (*induct t*) *auto*

**lemma** *Ide-iff-Trg-self*:  
**shows**  $\text{Arr } t \implies \text{Ide } t \longleftrightarrow \text{Trg } t = t$   
**apply** (*induct t*)  
**apply** *auto*  
**by** (*metis Ide.simps(5) Ide-Subst Ide-Trg*) $+$

**lemma** *Arr-Trg [simp]*:  
**assumes**  $\text{Arr } X$   
**shows**  $\text{Arr } (\text{Trg } X)$   
**using** *assms Ide-Trg Ide-implies-Arr* **by** *blast*

**lemma** *Src-Src [simp]*:  
**assumes**  $\text{Arr } t$   
**shows**  $\text{Src } (\text{Src } t) = \text{Src } t$   
**using** *assms Ide-Src Ide-iff-Src-self Ide-implies-Arr* **by** *blast*

**lemma** *Trg-Src [simp]*:  
**assumes**  $\text{Arr } t$   
**shows**  $\text{Trg } (\text{Src } t) = \text{Src } t$   
**using** *assms Ide-Src Ide-iff-Trg-self Ide-implies-Arr* **by** *blast*

**lemma** *Trg-Trg [simp]*:  
**assumes**  $\text{Arr } t$   
**shows**  $\text{Trg } (\text{Trg } t) = \text{Trg } t$   
**using** *assms Ide-Trg Ide-iff-Trg-self Ide-implies-Arr* **by** *blast*

**lemma** *Src-Trg [simp]*:  
**assumes**  $\text{Arr } t$   
**shows**  $\text{Src } (\text{Trg } t) = \text{Trg } t$   
**using** *assms Ide-Trg Ide-iff-Src-self Ide-implies-Arr* **by** *blast*

Two terms are syntactically *coinitial* if they are arrows with the same source; that is, they represent two reductions from the same starting term.

**abbreviation** *Coinitial*  
**where**  $\text{Coinitial } t \ u \equiv \text{Arr } t \wedge \text{Arr } u \wedge \text{Src } t = \text{Src } u$

We now show that terms are consistent if and only if they are coinitial.

**lemma** *Coinitial-cases*:  
**assumes**  $\text{Arr } t$  **and**  $\text{Arr } t'$  **and**  $\text{Src } t = \text{Src } t'$   
**shows**  $(t = \# \wedge t' = \#) \vee$

$(\exists x. t = \langle\langle x \rangle\rangle \wedge t' = \langle\langle x \rangle\rangle) \vee$   
 $(\exists u u'. t = \lambda[u] \wedge t' = \lambda[u']) \vee$   
 $(\exists u v u' v'. t = u \circ v \wedge t' = u' \circ v') \vee$   
 $(\exists u v u' v'. t = \lambda[u] \bullet v \wedge t' = \lambda[u'] \bullet v') \vee$   
 $(\exists u v u' v'. t = \lambda[u] \circ v \wedge t' = \lambda[u'] \circ v') \vee$   
 $(\exists u v u' v'. t = \lambda[u] \bullet v \wedge t' = \lambda[u'] \circ v')$

**using** *assms*  
**by** (*cases t; cases t'*) *auto*

**lemma** *Con-implies-Coinitial-ind*:  
**shows**  $\llbracket \text{size } t + \text{size } u \leq n; t \frown u \rrbracket \Longrightarrow \text{Coinitial } t \ u$   
**using** *Con-implies-Arr1 Con-implies-Arr2*  
**by** (*induct n arbitrary: t u*) *auto*

**lemma** *Coinitial-implies-Con-ind*:  
**shows**  $\llbracket \text{size } (\text{Src } t) \leq n; \text{Coinitial } t \ u \rrbracket \Longrightarrow t \frown u$   
**proof** (*induct n arbitrary: t u*)  
**show**  $\bigwedge t \ u. \llbracket \text{size } (\text{Src } t) \leq 0; \text{Coinitial } t \ u \rrbracket \Longrightarrow t \frown u$   
**by** *auto*  
**fix** *n t u*  
**assume** *Coinitial: Coinitial t u*  
**assume** *n: size (Src t) ≤ Suc n*  
**assume** *ind:  $\bigwedge t \ u. \llbracket \text{size } (\text{Src } t) \leq n; \text{Coinitial } t \ u \rrbracket \Longrightarrow t \frown u$*   
**show**  $t \frown u$   
**using** *n ind Coinitial Coinitial-cases [of t u] Subst-not-Nil* **by** *auto*  
**qed**

**lemma** *Coinitial-iff-Con*:  
**shows**  $\text{Coinitial } t \ u \longleftrightarrow t \frown u$   
**using** *Coinitial-implies-Con-ind Con-implies-Coinitial-ind* **by** *blast*

**lemma** *Coinitial-Raise-Raise*:  
**shows**  $\text{Coinitial } t \ u \Longrightarrow \text{Coinitial } (\text{Raise } d \ n \ t) (\text{Raise } d \ n \ u)$   
**using** *Arr-Raise Src-Raise*  
**apply** (*induct t arbitrary: d n u, auto*)  
**by** (*metis Raise.simps(3-4)*)

**lemma** *Con-sym*:  
**shows**  $t \frown u \longleftrightarrow u \frown t$   
**by** (*metis Coinitial-iff-Con*)

**lemma** *ConI [intro, simp]*:  
**assumes** *Arr t and Arr u and Src t = Src u*  
**shows**  $\text{Con } t \ u$   
**using** *assms Coinitial-iff-Con* **by** *blast*

**lemma** *Con-Arr-Src [simp]*:  
**assumes** *Arr t*  
**shows**  $t \frown \text{Src } t$  **and**  $\text{Src } t \frown t$

**using** *assms*  
**by** (*auto simp add: Ide-Src Ide-implies-Arr*)

**lemma** *resid-Arr-self*:  
**shows**  $\text{Arr } t \implies t \setminus t = \text{Trg } t$   
**by** (*induct t auto*)

The following result is not used in the formal development that follows, but it requires some proof and might eventually be useful.

**lemma** *finite-branching*:  
**shows**  $\text{Ide } a \implies \text{finite } \{t. \text{Arr } t \wedge \text{Src } t = a\}$   
**proof** (*induct a*)  
**show**  $\text{Ide } \# \implies \text{finite } \{t. \text{Arr } t \wedge \text{Src } t = \#\}$   
**by** *simp*  
**fix**  $x$   
**have**  $\bigwedge t. \text{Src } t = \langle x \rangle \longleftrightarrow t = \langle x \rangle$   
**using** *Src.elims* **by** *blast*  
**thus**  $\text{finite } \{t. \text{Arr } t \wedge \text{Src } t = \langle x \rangle\}$   
**by** *simp*  
**next**  
**fix**  $a$   
**assume**  $a: \text{Ide } \lambda[a]$   
**assume**  $\text{ind}: \text{Ide } a \implies \text{finite } \{t. \text{Arr } t \wedge \text{Src } t = a\}$   
**have**  $\{t. \text{Arr } t \wedge \text{Src } t = \lambda[a]\} = \text{Lam } \langle \{t. \text{Arr } t \wedge \text{Src } t = a\}$   
**using** *Coinitial-cases* **by** *fastforce*  
**thus**  $\text{finite } \{t. \text{Arr } t \wedge \text{Src } t = \lambda[a]\}$   
**using**  $a$  *ind* **by** *simp*  
**next**  
**fix**  $a1 a2$   
**assume**  $\text{ind1}: \text{Ide } a1 \implies \text{finite } \{t. \text{Arr } t \wedge \text{Src } t = a1\}$   
**assume**  $\text{ind2}: \text{Ide } a2 \implies \text{finite } \{t. \text{Arr } t \wedge \text{Src } t = a2\}$   
**assume**  $a: \text{Ide } (\lambda[a1] \bullet a2)$   
**show**  $\text{finite } \{t. \text{Arr } t \wedge \text{Src } t = \lambda[a1] \bullet a2\}$   
**using**  $a$  *ind1 ind2* **by** *simp*  
**next**  
**fix**  $a1 a2$   
**assume**  $\text{ind1}: \text{Ide } a1 \implies \text{finite } \{t. \text{Arr } t \wedge \text{Src } t = a1\}$   
**assume**  $\text{ind2}: \text{Ide } a2 \implies \text{finite } \{t. \text{Arr } t \wedge \text{Src } t = a2\}$   
**assume**  $a: \text{Ide } (a1 \circ a2)$   
**have**  $\{t. \text{Arr } t \wedge \text{Src } t = a1 \circ a2\} =$   
 $(\{t. \text{is-App } t\} \cap (\{t. \text{Arr } t \wedge \text{Src } (\text{un-App1 } t) = a1 \wedge \text{Src } (\text{un-App2 } t) = a2\})) \cup$   
 $(\{t. \text{is-Beta } t \wedge \text{is-Lam } a1\} \cap$   
 $(\{t. \text{Arr } t \wedge \text{is-Lam } a1 \wedge \text{Src } (\text{un-Beta1 } t) = \text{un-Lam } a1 \wedge \text{Src } (\text{un-Beta2 } t) = a2\}))$   
**by** *fastforce*  
**have**  $\{t. \text{Arr } t \wedge \text{Src } t = a1 \circ a2\} =$   
 $(\lambda(t1, t2). t1 \circ t2) \langle (\{t1. \text{Arr } t1 \wedge \text{Src } t1 = a1\} \times \{t2. \text{Arr } t2 \wedge \text{Src } t2 = a2\}) \cup$   
 $(\lambda(t1, t2). \lambda[t1] \bullet t2) \langle$   
 $(\{t1t2. \text{is-Lam } a1\} \cap$   
 $\{t1. \text{Arr } t1 \wedge \text{Src } t1 = \text{un-Lam } a1\} \times \{t2. \text{Arr } t2 \wedge \text{Src } t2 = a2\})$

**proof**  
**show**  $(\lambda(t1, t2). t1 \circ t2) \text{ ‘ } (\{t1. Arr\ t1 \wedge Src\ t1 = a1\} \times \{t2. Arr\ t2 \wedge Src\ t2 = a2\}) \cup$   
 $(\lambda(t1, t2). \lambda[t1] \bullet t2) \text{ ‘}$   
 $(\{t1t2. is-Lam\ a1\} \cap$   
 $\{t1. Arr\ t1 \wedge Src\ t1 = un-Lam\ a1\} \times \{t2. Arr\ t2 \wedge Src\ t2 = a2\})$   
 $\subseteq \{t. Arr\ t \wedge Src\ t = a1 \circ a2\}$   
**by auto**  
**show**  $\{t. Arr\ t \wedge Src\ t = a1 \circ a2\}$   
 $\subseteq (\lambda(t1, t2). t1 \circ t2) \text{ ‘}$   
 $(\{t1. Arr\ t1 \wedge Src\ t1 = a1\} \times \{t2. Arr\ t2 \wedge Src\ t2 = a2\}) \cup$   
 $(\lambda(t1, t2). \lambda[t1] \bullet t2) \text{ ‘}$   
 $(\{t1t2. is-Lam\ a1\} \cap$   
 $\{t1. Arr\ t1 \wedge Src\ t1 = un-Lam\ a1\} \times \{t2. Arr\ t2 \wedge Src\ t2 = a2\})$   
**proof**  
**fix**  $t$   
**assume**  $t \in \{t. Arr\ t \wedge Src\ t = a1 \circ a2\}$   
**have**  $is-App\ t \vee is-Beta\ t$   
**using**  $t$  **by auto**  
**moreover have**  $is-App\ t \implies t \in (\lambda(t1, t2). t1 \circ t2) \text{ ‘}$   
 $(\{t1. Arr\ t1 \wedge Src\ t1 = a1\} \times \{t2. Arr\ t2 \wedge Src\ t2 = a2\})$   
**using**  $t$  *image-iff is-App-def* **by fastforce**  
**moreover have**  $is-Beta\ t \implies$   
 $t \in (\lambda(t1, t2). \lambda[t1] \bullet t2) \text{ ‘}$   
 $(\{t1t2. is-Lam\ a1\} \cap$   
 $\{t1. Arr\ t1 \wedge Src\ t1 = un-Lam\ a1\} \times \{t2. Arr\ t2 \wedge Src\ t2 = a2\})$   
**using**  $t$  *is-Beta-def* **by fastforce**  
**ultimately show**  $t \in (\lambda(t1, t2). t1 \circ t2) \text{ ‘}$   
 $(\{t1. Arr\ t1 \wedge Src\ t1 = a1\} \times \{t2. Arr\ t2 \wedge Src\ t2 = a2\}) \cup$   
 $(\lambda(t1, t2). \lambda[t1] \bullet t2) \text{ ‘}$   
 $(\{t1t2. is-Lam\ a1\} \cap$   
 $\{t1. Arr\ t1 \wedge Src\ t1 = un-Lam\ a1\} \times \{t2. Arr\ t2 \wedge Src\ t2 = a2\})$   
**by blast**  
**qed**  
**qed**  
**moreover have**  $finite\ (\{t1. Arr\ t1 \wedge Src\ t1 = a1\} \times \{t2. Arr\ t2 \wedge Src\ t2 = a2\})$   
**using**  $a$  *ind1 ind2 Ide.simps(4)* **by blast**  
**moreover have**  $is-Lam\ a1 \implies$   
 $finite\ (\{t1. Arr\ t1 \wedge Src\ t1 = un-Lam\ a1\} \times \{t2. Arr\ t2 \wedge Src\ t2 = a2\})$   
**proof** –  
**assume**  $a1: is-Lam\ a1$   
**have**  $Ide\ (un-Lam\ a1)$   
**using**  $a\ a1$  *is-Lam-def* **by force**  
**have**  $Lam\ \text{‘ } \{t1. Arr\ t1 \wedge Src\ t1 = un-Lam\ a1\} = \{t. Arr\ t \wedge Src\ t = a1\}$   
**proof**  
**show**  $Lam\ \text{‘ } \{t1. Arr\ t1 \wedge Src\ t1 = un-Lam\ a1\} \subseteq \{t. Arr\ t \wedge Src\ t = a1\}$   
**using**  $a1$  **by fastforce**  
**show**  $\{t. Arr\ t \wedge Src\ t = a1\} \subseteq Lam\ \text{‘ } \{t1. Arr\ t1 \wedge Src\ t1 = un-Lam\ a1\}$   
**proof**  
**fix**  $t$

```

assume  $t \in \{t. \text{Arr } t \wedge \text{Src } t = a1\}$ 
have  $\text{is-Lam } t$ 
  using  $a1$  t by auto
hence  $\text{un-Lam } t \in \{t1. \text{Arr } t1 \wedge \text{Src } t1 = \text{un-Lam } a1\}$ 
  using  $\text{is-Lam-def } t$  by force
thus  $t \in \text{Lam } \langle \{t1. \text{Arr } t1 \wedge \text{Src } t1 = \text{un-Lam } a1\}$ 
  by (metis  $\langle \text{is-Lam } t \rangle \text{lambda.collapse}(2) \text{rev-image-eqI}$ )
qed
qed
moreover have  $\text{inj Lam}$ 
  using  $\text{inj-on-def}$  by blast
ultimately have  $\text{finite } \{t1. \text{Arr } t1 \wedge \text{Src } t1 = \text{un-Lam } a1\}$ 
  by (metis (mono-tags, lifting) Ide.simps(4) a finite-imageD ind1 injD inj-onI)
moreover have  $\text{finite } \{t2. \text{Arr } t2 \wedge \text{Src } t2 = a2\}$ 
  using Ide.simps(4) a ind2 by blast
ultimately
show  $\text{finite } (\{t1. \text{Arr } t1 \wedge \text{Src } t1 = \text{un-Lam } a1\} \times \{t2. \text{Arr } t2 \wedge \text{Src } t2 = a2\})$ 
  by blast
qed
ultimately show  $\text{finite } \{t. \text{Arr } t \wedge \text{Src } t = a1 \circ a2\}$ 
  using  $a$  ind1 ind2 by simp
qed

```

### 3.2.3 Residuation and Substitution

We now develop a series of lemmas that involve the interaction of residuation and substitution.

**lemma** *Raise-resid*:

**shows**  $t \frown u \implies \text{Raise } k \ n \ (t \setminus u) = \text{Raise } k \ n \ t \setminus \text{Raise } k \ n \ u$

**proof** –

**let**  $?P = \lambda(t, u). \forall k \ n. t \frown u \longrightarrow \text{Raise } k \ n \ (t \setminus u) = \text{Raise } k \ n \ t \setminus \text{Raise } k \ n \ u$

**have**  $\bigwedge t \ u.$

$\forall t' \ u'. ((t', u'), (t, u)) \in \text{subterm-pair-rel} \longrightarrow$

$(\forall k \ n. t' \frown u' \longrightarrow$

$\text{Raise } k \ n \ (t' \setminus u') = \text{Raise } k \ n \ t' \setminus \text{Raise } k \ n \ u') \implies$

$(\bigwedge k \ n. t \frown u \implies \text{Raise } k \ n \ (t \setminus u) = \text{Raise } k \ n \ t \setminus \text{Raise } k \ n \ u)$

**using** *subterm-lemmas Coinitial-iff-Con Coinitial-Raise-Raise Raise-subst* **by** *auto*

**thus**  $t \frown u \implies \text{Raise } k \ n \ (t \setminus u) = \text{Raise } k \ n \ t \setminus \text{Raise } k \ n \ u$

**using** *wf-subterm-pair-rel wf-induct [of subterm-pair-rel ?P]* **by** *blast*

**qed**

**lemma** *Con-Raise*:

**shows**  $t \frown u \implies \text{Raise } d \ n \ t \frown \text{Raise } d \ n \ u$

**by** (*metis* *Raise-not-Nil Raise-resid*)

The following is Huet’s Commutation Theorem [7]: “substitution commutes with residuation”.

**lemma** *resid-Subst*:

**assumes**  $t \frown t'$  **and**  $u \frown u'$   
**shows**  $\text{Subst } n \ t \ u \setminus \text{Subst } n \ t' \ u' = \text{Subst } n \ (t \setminus t') \ (u \setminus u')$   
**proof** –  
**let**  $?P = \lambda(u, u'). \forall n \ t \ t'. t \frown t' \wedge u \frown u' \longrightarrow$   
 $\text{Subst } n \ t \ u \setminus \text{Subst } n \ t' \ u' = \text{Subst } n \ (t \setminus t') \ (u \setminus u')$   
**have**  $\bigwedge u \ u'. \forall w \ w'. ((w, w'), (u, u')) \in \text{subterm-pair-rel} \longrightarrow$   
 $(\forall n \ v \ v'. v \frown v' \wedge w \frown w' \longrightarrow$   
 $\text{Subst } n \ v \ w \setminus \text{Subst } n \ v' \ w' = \text{Subst } n \ (v \setminus v') \ (w \setminus w')) \implies$   
 $\forall n \ t \ t'. t \frown t' \wedge u \frown u' \longrightarrow$   
 $\text{Subst } n \ t \ u \setminus \text{Subst } n \ t' \ u' = \text{Subst } n \ (t \setminus t') \ (u \setminus u')$   
**using** *subterm-lemmas Raise-resid Subst-not-Nil Con-Raise Raise-subst substitution-lemma*  
**by** *auto*  
**thus** *?thesis*  
**using** *assms wf-subterm-pair-rel wf-induct [of subterm-pair-rel ?P] by auto*  
**qed**

**lemma** *Trg-Subst [simp]*:  
**shows**  $\llbracket \text{Arr } t; \text{Arr } u \rrbracket \implies \text{Trg} (\text{Subst } d \ t \ u) = \text{Subst } d \ (\text{Trg } t) \ (\text{Trg } u)$   
**by** (*metis Arr-Subst Arr-Trg Arr-not-Nil resid-Arr-self resid-Subst*)

**lemma** *Src-resid*:  
**shows**  $t \frown u \implies \text{Src} (t \setminus u) = \text{Trg } u$   
**proof** (*induct u arbitrary: t, auto simp add: Arr-resid*)  
**fix**  $t \ t1'$   
**show**  $\bigwedge t2'. \llbracket \bigwedge t1. t1 \frown t1' \implies \text{Src} (t1 \setminus t1') = \text{Trg } t1';$   
 $\bigwedge t2. t2 \frown t2' \implies \text{Src} (t2 \setminus t2') = \text{Trg } t2';$   
 $t \frown t1' \circ t2' \rrbracket$   
 $\implies \text{Src} (t \setminus (t1' \circ t2')) = \text{Trg } t1' \circ \text{Trg } t2'$   
**apply** (*cases t; cases t1'*)  
**apply** *auto*  
**by** (*metis Src.simps(3) lambda.distinct(3) lambda.sel(2) resid.simps(2)*)  
**qed**

**lemma** *Coinitial-resid-resid*:  
**assumes**  $t \frown v$  **and**  $u \frown v$   
**shows**  $\text{Coinitial} (t \setminus v) \ (u \setminus v)$   
**using** *assms Src-resid Arr-resid Coinitial-iff-Con by presburger*

**lemma** *Con-implies-is-Lam-iff-is-Lam*:  
**assumes**  $t \frown u$   
**shows**  $\text{is-Lam } t \longleftrightarrow \text{is-Lam } u$   
**using** *assms by auto*

**lemma** *Con-implies-Coinitial3*:  
**assumes**  $t \setminus v \frown u \setminus v$   
**shows**  $\text{Coinitial } v \ u$  **and**  $\text{Coinitial } v \ t$  **and**  $\text{Coinitial } u \ t$   
**using** *assms*  
**by** (*metis Coinitial-iff-Con resid.simps(7)*)**+**

We can now prove Lévy’s “Cube Lemma” [8], which is the key axiom for a residuated

transition system.

**lemma** *Cube*:

**shows**  $v \setminus t \frown u \setminus t \implies (v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$

**proof** –

**let**  $?P = \lambda(t, u, v). v \setminus t \frown u \setminus t \longrightarrow (v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$

**have**  $\bigwedge t u v.$

$\forall t' u' v'.$

$((t', u', v'), (t, u, v)) \in \text{subterm-triple-rel} \longrightarrow ?P(t', u', v') \implies$   
 $v \setminus t \frown u \setminus t \longrightarrow (v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$

**proof** –

**fix**  $t u v$

**assume**  $\text{ind}: \forall t' u' v'.$

$((t', u', v'), (t, u, v)) \in \text{subterm-triple-rel} \longrightarrow ?P(t', u', v')$

**show**  $v \setminus t \frown u \setminus t \longrightarrow (v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$

**proof** (*intro impI*)

**assume**  $\text{con}: v \setminus t \frown u \setminus t$

**have**  $\text{Con } v t$

**using**  $\text{con}$  **by** *auto*

**moreover have**  $\text{Con } u t$

**using**  $\text{con}$  **by** *auto*

**ultimately show**  $(v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$

**using** *subterm-lemmas ind Coinitial-iff-Con Coinitial-resid-resid resid-Subst*

**apply** (*elim ConE [of v t] ConE [of u t]*)

**apply** *simp-all*

**apply** *metis*

**apply** *metis*

**apply** (*cases un-App1 t; cases un-App1 v, simp-all*)

**apply** *metis*

**apply** *metis*

**apply** *metis*

**apply** *metis*

**apply** *metis*

**apply** (*cases un-App1 u, simp-all*)

**apply** *metis*

**by** *metis*

**qed**

**qed**

**hence**  $?P(t, u, v)$

**using** *wf-subterm-triple-rel wf-induct [of subterm-triple-rel ?P]* **by** *blast*

**thus**  $v \setminus t \frown u \setminus t \implies (v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$

**by** *simp*

**qed**

### 3.2.4 Residuation Determines an RTS

We are now in a position to verify that the residuation operation that we have defined satisfies the axioms for a residuated transition system, and that various notions which we have defined syntactically above (*e.g.* arrow, source, target) agree with the versions derived abstractly from residuation.

**sublocale** *partial-magma resid*  
**apply** *unfold-locales*  
**by** (*metis Arr.simps(1) Cointial-iff-Con*)

**lemma** *null-char [simp]*:  
**shows**  $null = \#$   
**using** *null-def*  
**by** (*metis null-is-zero(2) resid.simps(7)*)

**sublocale** *residuation resid*  
**using** *null-char Arr-resid Cointial-iff-Con Cube*  
**apply** (*unfold-locales, auto*)  
**by** *metis+*

**notation** *resid* (**infix**  $\langle \backslash \rangle$  70)

**lemma** *resid-is-residuation*:  
**shows** *residuation resid*  
**..**

**lemma** *arr-char [iff]*:  
**shows**  $arr\ t \longleftrightarrow Arr\ t$   
**using** *Cointial-iff-Con arr-def con-def null-char* **by** *auto*

**lemma** *ide-char [iff]*:  
**shows**  $ide\ t \longleftrightarrow Ide\ t$   
**by** (*metis Ide-iff-Trg-self Ide-implies-Arr arr-char arr-resid-iff-con ide-def resid-Arr-self*)

**lemma** *resid-Arr-Ide*:  
**shows**  $\llbracket Ide\ a; Cointial\ t\ a \rrbracket \Longrightarrow t \backslash a = t$   
**using** *Ide-iff-Src-self*  
**by** (*induct t arbitrary: a, auto*)

**lemma** *resid-Ide-Arr*:  
**shows**  $\llbracket Ide\ a; Cointial\ a\ t \rrbracket \Longrightarrow Ide\ (a \backslash t)$   
**by** (*metis Cointial-resid-resid ConI Ide-iff-Trg-self cube resid-Arr-Ide resid-Arr-self*)

**lemma** *resid-Arr-Src [simp]*:  
**assumes** *Arr t*  
**shows**  $t \backslash Src\ t = t$   
**using** *assms Ide-Src*  
**by** (*simp add: Ide-implies-Arr resid-Arr-Ide*)

**lemma** *resid-Src-Arr [simp]*:  
**assumes** *Arr t*  
**shows**  $Src\ t \backslash t = Trg\ t$   
**using** *assms*

by (metis (full-types) Con-Arr-Src(2) Con-implies-Arr1 Src-Src Src-resid cube  
resid-Arr-Src resid-Arr-self)

**sublocale** *rts resid*

**proof**

show  $\bigwedge a t. \llbracket \text{ide } a; \text{ con } t \ a \rrbracket \implies t \setminus a = t$   
 using *ide-char resid-Arr-Ide*  
 by (metis *Coinitial-iff-Con con-def null-char*)  
 show  $\bigwedge t. \text{arr } t \implies \text{ide } (\text{trg } t)$   
 by (*simp add: Ide-Trg resid-Arr-self trg-def*)  
 show  $\bigwedge a t. \llbracket \text{ide } a; \text{ con } a \ t \rrbracket \implies \text{ide } (\text{resid } a \ t)$   
 using *ide-char null-char resid-Ide-Arr Coinitial-iff-Con con-def* by *force*  
 show  $\bigwedge t u. \text{con } t \ u \implies \exists a. \text{ide } a \ \wedge \ \text{con } a \ t \ \wedge \ \text{con } a \ u$   
 by (metis *Coinitial-iff-Con Ide-Src Ide-iff-Src-self Ide-implies-Arr con-def*  
*ide-char null-char*)  
 show  $\bigwedge t u v. \llbracket \text{ide } (\text{resid } t \ u); \text{ con } u \ v \rrbracket \implies \text{con } (\text{resid } t \ u) \ (\text{resid } v \ u)$   
 by (metis *Coinitial-resid-resid ide-char not-arr-null null-char resid-Ide-Arr*  
*con-def con-sym ide-implies-arr*)

**qed**

**lemma** *is-rts*:

**shows** *rts resid*

..

**lemma** *sources-char<sub>Λ</sub>*:

**shows** *sources t = (if Arr t then {Src t} else {})*

**proof** (*cases Arr t*)

show  $\neg \text{Arr } t \implies ?thesis$   
 using *arr-char arr-iff-has-source* by *auto*  
 assume *t: Arr t*  
 have  $1: \{\text{Src } t\} \subseteq \text{sources } t$   
 using *t Ide-Src* by *force*  
 moreover have  $\text{sources } t \subseteq \{\text{Src } t\}$   
 by (metis *Coinitial-iff-Con Ide-iff-Src-self ide-char in-sourcesE null-char*  
*con-def singleton-iff subsetI*)  
 ultimately show *?thesis*  
 using *t* by *auto*

**qed**

**lemma** *sources-simp [simp]*:

**assumes** *Arr t*

**shows** *sources t = {Src t}*

using *assms sources-char<sub>Λ</sub>* by *auto*

**lemma** *sources-simps [simp]*:

**shows** *sources ‡ = {}*

**and** *sources «x» = {«x»}*

**and** *arr t  $\implies$  sources λ[t] = {λ[Src t]}*

**and**  $\llbracket \text{arr } t; \text{ arr } u \rrbracket \implies \text{sources } (t \circ u) = \{\text{Src } t \circ \text{Src } u\}$

**and**  $\llbracket arr\ t; arr\ u \rrbracket \implies sources\ (\lambda[t] \bullet u) = \{\lambda[Src\ t] \circ Src\ u\}$   
**using** *sources-char $_{\Lambda}$*  **by** *auto*

**lemma** *targets-char $_{\Lambda}$* :  
**shows** *targets*  $t = (if\ Arr\ t\ then\ \{Trg\ t\}\ else\ \{\})$   
**proof** (*cases* *Arr*  $t$ )  
**show**  $\neg Arr\ t \implies ?thesis$   
**by** (*meson* *arr-char* *arr-iff-has-target*)  
**assume**  $t: Arr\ t$   
**have**  $1: \{Trg\ t\} \subseteq targets\ t$   
**using** *t resid-Arr-self* *trg-def* *trg-in-targets* **by** *force*  
**moreover** **have** *targets*  $t \subseteq \{Trg\ t\}$   
**by** (*metis*  $1\ Ide-iff-Src-self\ arr-char\ ide-char\ ide-implies-arr$   
*in-targetsE* *insert-subset* *prfx-implies-con* *resid-Arr-self*  
*sources-resid* *sources-simp*  $t$ )  
**ultimately** **show**  $?thesis$   
**using**  $t$  **by** *auto*  
**qed**

**lemma** *targets-simp* [*simp*]:  
**assumes** *Arr*  $t$   
**shows** *targets*  $t = \{Trg\ t\}$   
**using** *assms* *targets-char $_{\Lambda}$*  **by** *auto*

**lemma** *targets-simps* [*simp*]:  
**shows** *targets*  $\# = \{\}$   
**and** *targets*  $\langle\langle x \rangle\rangle = \{\langle\langle x \rangle\rangle\}$   
**and** *arr*  $t \implies targets\ \lambda[t] = \{\lambda[Trg\ t]\}$   
**and**  $\llbracket arr\ t; arr\ u \rrbracket \implies targets\ (t \circ u) = \{Trg\ t \circ Trg\ u\}$   
**and**  $\llbracket arr\ t; arr\ u \rrbracket \implies targets\ (\lambda[t] \bullet u) = \{subst\ (Trg\ u)\ (Trg\ t)\}$   
**using** *targets-char $_{\Lambda}$*  **by** *auto*

**lemma** *seq-char*:  
**shows** *seq*  $t\ u \iff Arr\ t \wedge Arr\ u \wedge Trg\ t = Src\ u$   
**using** *seq-def* *arr-char* *sources-char $_{\Lambda}$*  *targets-char $_{\Lambda}$*  **by** *force*

**lemma** *seqI $_{\Lambda}$*  [*intro*, *simp*]:  
**assumes** *Arr*  $t$  **and** *Arr*  $u$  **and** *Trg*  $t = Src\ u$   
**shows** *seq*  $t\ u$   
**using** *assms* *seq-char* **by** *simp*

**lemma** *seqE $_{\Lambda}$*  [*elim*]:  
**assumes** *seq*  $t\ u$   
**and**  $\llbracket Arr\ t; Arr\ u; Trg\ t = Src\ u \rrbracket \implies T$   
**shows**  $T$   
**using** *assms* *seq-char* **by** *blast*

The following classifies the ways that transitions can be sequential. It is useful for later proofs by case analysis.

**lemma** *seq-cases*:

**assumes** *seq t u*

**shows**  $(is-Var\ t \wedge is-Var\ u) \vee$   
 $(is-Lam\ t \wedge is-Lam\ u) \vee$   
 $(is-App\ t \wedge is-App\ u) \vee$   
 $(is-App\ t \wedge is-Beta\ u \wedge is-Lam\ (un-App1\ t)) \vee$   
 $(is-App\ t \wedge is-Beta\ u \wedge is-Beta\ (un-App1\ t)) \vee$   
 $is-Beta\ t$

**using** *assms seq-char*

**by** (*cases t; cases u*) *auto*

**sublocale** *confluent-rts resid*

**by** (*unfold-locales*) *fastforce*

**lemma** *is-confluent-rts*:

**shows** *confluent-rts resid*

..

**lemma** *con-char [iff]*:

**shows**  $con\ t\ u \longleftrightarrow Con\ t\ u$

**by** *fastforce*

**lemma** *coinitial-char [iff]*:

**shows**  $coinitial\ t\ u \longleftrightarrow Coinitial\ t\ u$

**by** *fastforce*

**lemma** *sources-Raise*:

**assumes** *Arr t*

**shows**  $sources\ (Raise\ d\ n\ t) = \{Raise\ d\ n\ (Src\ t)\}$

**using** *assms*

**by** (*simp add: Coinitial-Raise-Raise Src-Raise*)

**lemma** *targets-Raise*:

**assumes** *Arr t*

**shows**  $targets\ (Raise\ d\ n\ t) = \{Raise\ d\ n\ (Trg\ t)\}$

**using** *assms*

**by** (*metis Arr-Raise ConI Raise-resid resid-Arr-self targets-char<sub>Λ</sub>*)

**lemma** *sources-subst [simp]*:

**assumes** *Arr t and Arr u*

**shows**  $sources\ (subst\ t\ u) = \{subst\ (Src\ t)\ (Src\ u)\}$

**using** *assms sources-char<sub>Λ</sub> Arr-Subst arr-char by simp*

**lemma** *targets-subst [simp]*:

**assumes** *Arr t and Arr u*

**shows**  $targets\ (subst\ t\ u) = \{subst\ (Trg\ t)\ (Trg\ u)\}$

**using** *assms targets-char<sub>Λ</sub> Arr-Subst arr-char by simp*

**notation** *prfx* (**infix**  $\langle \lesssim \rangle$  50)

**notation** *cong* (infix  $\langle \sim \rangle$  50)

**lemma** *prfx-char* [iff]:  
**shows**  $t \lesssim u \longleftrightarrow \text{Ide } (t \setminus u)$   
**using** *ide-char* **by** *simp*

**lemma** *prfx-Var-iff*:  
**shows**  $u \lesssim \langle i \rangle \longleftrightarrow u = \langle i \rangle$   
**by** (*metis* *Arr.simps*(2) *Coinitial-iff-Con* *Ide.simps*(1) *Ide-iff-Src-self* *Src.simps*(2)  
*ide-char* *resid-Arr-Ide*)

**lemma** *prfx-Lam-iff*:  
**shows**  $u \lesssim \text{Lam } t \longleftrightarrow \text{is-Lam } u \wedge \text{un-Lam } u \lesssim t$   
**using** *ide-char* *Arr-not-Nil* *Con-implies-is-Lam-iff-is-Lam* *Ide-implies-Arr* *is-Lam-def*  
**by** *fastforce*

**lemma** *prfx-App-iff*:  
**shows**  $u \lesssim t1 \circ t2 \longleftrightarrow \text{is-App } u \wedge \text{un-App1 } u \lesssim t1 \wedge \text{un-App2 } u \lesssim t2$   
**using** *ide-char*  
**by** (*cases* *u*; *cases* *t1*) *auto*

**lemma** *prfx-Beta-iff*:  
**shows**  $u \lesssim \lambda[t1] \bullet t2 \longleftrightarrow$   
 $(\text{is-App } u \wedge \text{un-App1 } u \lesssim \lambda[t1] \wedge \text{un-App2 } u \frown t2 \wedge$   
 $(0 \in \text{FV } (\text{un-Lam } (\text{un-App1 } u) \setminus t1) \longrightarrow \text{un-App2 } u \lesssim t2)) \vee$   
 $(\text{is-Beta } u \wedge \text{un-Beta1 } u \lesssim t1 \wedge \text{un-Beta2 } u \frown t2 \wedge$   
 $(0 \in \text{FV } (\text{un-Beta1 } u \setminus t1) \longrightarrow \text{un-Beta2 } u \lesssim t2))$   
**using** *ide-char* *Ide-Subst-iff*  
**by** (*cases* *u*; *cases* *un-App1* *u*) *auto*

**lemma** *cong-Ide-are-eq*:  
**assumes**  $t \sim u$  **and** *Ide*  $t$  **and** *Ide*  $u$   
**shows**  $t = u$   
**using** *assms*  
**by** (*metis* *Coinitial-iff-Con* *Ide-iff-Src-self* *con-char* *prfx-implies-con*)

**lemma** *eq-Ide-are-cong*:  
**assumes**  $t = u$  **and** *Ide*  $t$   
**shows**  $t \sim u$   
**using** *assms* *Ide-implies-Arr* *resid-Ide-Arr* **by** *blast*

**sublocale** *weakly-extensional-rts* *resid*  
**apply** *unfold-locales*  
**by** (*metis* *Coinitial-iff-Con* *Ide-iff-Src-self* *Ide-implies-Arr* *ide-char* *ide-def*)

**lemma** *is-weakly-extensional-rts*:  
**shows** *weakly-extensional-rts* *resid*  
**..**

**lemma** *src-char* [*simp*]:  
**shows**  $\text{src } t = (\text{if } \text{Arr } t \text{ then } \text{Src } t \text{ else } \#)$   
**using** *src-def* **by** *force*

**lemma** *trg-char* [*simp*]:  
**shows**  $\text{trg } t = (\text{if } \text{Arr } t \text{ then } \text{Trg } t \text{ else } \#)$   
**by** (*metis Coinitial-iff-Con resid-Arr-self trg-def*)

We “almost” have an extensional RTS. The case that fails is  $\lambda[t1] \bullet t2 \sim u \implies \lambda[t1] \bullet t2 = u$ . This is because  $t1$  might ignore its argument, so that  $\text{subst } t2 \ t1 = \text{subst } t2' \ t1$ , with both sides being identities, even if  $t2 \neq t2'$ .

The following gives a concrete example of such a situation.

**abbreviation** *non-extensional-ex1*  
**where**  $\text{non-extensional-ex1} \equiv \lambda[\lambda[\langle 0 \rangle] \circ \lambda[\langle 0 \rangle]] \bullet (\lambda[\langle 0 \rangle] \bullet \lambda[\langle 0 \rangle])$

**abbreviation** *non-extensional-ex2*  
**where**  $\text{non-extensional-ex2} \equiv \lambda[\lambda[\langle 0 \rangle] \circ \lambda[\langle 0 \rangle]] \bullet (\lambda[\langle 0 \rangle] \circ \lambda[\langle 0 \rangle])$

**lemma** *non-extensional*:  
**shows**  $\lambda[\langle 1 \rangle] \bullet \text{non-extensional-ex1} \sim \lambda[\langle 1 \rangle] \bullet \text{non-extensional-ex2}$   
**and**  $\lambda[\langle 1 \rangle] \bullet \text{non-extensional-ex1} \neq \lambda[\langle 1 \rangle] \bullet \text{non-extensional-ex2}$   
**by** *auto*

The following gives an example of two terms that are both coinitial and coterminial, but which are not congruent.

**abbreviation** *cong-nontrivial-ex1*  
**where**  $\text{cong-nontrivial-ex1} \equiv \lambda[\langle 0 \rangle \circ \langle 0 \rangle] \circ \lambda[\langle 0 \rangle \circ \langle 0 \rangle] \circ (\lambda[\langle 0 \rangle \circ \langle 0 \rangle] \bullet \lambda[\langle 0 \rangle \circ \langle 0 \rangle])$

**abbreviation** *cong-nontrivial-ex2*  
**where**  $\text{cong-nontrivial-ex2} \equiv \lambda[\langle 0 \rangle \circ \langle 0 \rangle] \bullet \lambda[\langle 0 \rangle \circ \langle 0 \rangle] \circ (\lambda[\langle 0 \rangle \circ \langle 0 \rangle] \circ \lambda[\langle 0 \rangle \circ \langle 0 \rangle])$

**lemma** *cong-nontrivial*:  
**shows** *coinitial cong-nontrivial-ex1 cong-nontrivial-ex2*  
**and** *coterminial cong-nontrivial-ex1 cong-nontrivial-ex2*  
**and**  $\neg \text{cong } \text{cong-nontrivial-ex1 } \text{cong-nontrivial-ex2}$   
**by** *auto*

Every two coinitial transitions have a join, obtained structurally by unioning the sets of marked redexes.

**fun** *Join* (**infix**  $\sqcup$  52)  
**where**  $\langle x \rangle \sqcup \langle x' \rangle = (\text{if } x = x' \text{ then } \langle x \rangle \text{ else } \#)$   
 $\lambda[t] \sqcup \lambda[t'] = \lambda[t \sqcup t']$   
 $\lambda[t] \circ u \sqcup \lambda[t'] \bullet u' = \lambda[(t \sqcup t')] \bullet (u \sqcup u')$   
 $\lambda[t] \bullet u \sqcup \lambda[t'] \circ u' = \lambda[(t \sqcup t')] \bullet (u \sqcup u')$   
 $t \circ u \sqcup t' \circ u' = (t \sqcup t') \circ (u \sqcup u')$   
 $\lambda[t] \bullet u \sqcup \lambda[t'] \bullet u' = \lambda[(t \sqcup t')] \bullet (u \sqcup u')$   
 $- \sqcup - = \#$

**lemma** *Join-sym*:  
**shows**  $t \sqcup u = u \sqcup t$   
**using** *Join.induct* [of  $\lambda t u. t \sqcup u = u \sqcup t$ ] **by** *auto*

**lemma** *Src-Join*:  
**shows**  $\text{Coinitial } t \ u \implies \text{Src } (t \sqcup u) = \text{Src } t$   
**proof** (*induct t arbitrary: u*)  
**show**  $\bigwedge u. \text{Coinitial } \# \ u \implies \text{Src } (\# \sqcup u) = \text{Src } \#$   
**by** *simp*  
**show**  $\bigwedge x u. \text{Coinitial } \langle\langle x \rangle\rangle \ u \implies \text{Src } (\langle\langle x \rangle\rangle \sqcup u) = \text{Src } \langle\langle x \rangle\rangle$   
**by** *auto*  
**fix**  $t \ u$   
**assume**  $\text{ind}: \bigwedge u. \text{Coinitial } t \ u \implies \text{Src } (t \sqcup u) = \text{Src } t$   
**assume**  $\text{tu}: \text{Coinitial } \lambda[t] \ u$   
**show**  $\text{Src } (\lambda[t] \sqcup u) = \text{Src } \lambda[t]$   
**using**  $\text{tu ind}$   
**by** (*cases u*) *auto*  
**next**  
**fix**  $t1 \ t2 \ u$   
**assume**  $\text{ind1}: \bigwedge u1. \text{Coinitial } t1 \ u1 \implies \text{Src } (t1 \sqcup u1) = \text{Src } t1$   
**assume**  $\text{ind2}: \bigwedge u2. \text{Coinitial } t2 \ u2 \implies \text{Src } (t2 \sqcup u2) = \text{Src } t2$   
**assume**  $\text{tu}: \text{Coinitial } (t1 \circ t2) \ u$   
**show**  $\text{Src } (t1 \circ t2 \sqcup u) = \text{Src } (t1 \circ t2)$   
**using**  $\text{tu ind1 ind2}$   
**apply** (*cases u, simp-all*)  
**apply** (*cases t1, simp-all*)  
**by** (*metis Arr.simps(3) Join.simps(2) Src.simps(3) lambda.sel(2)*)  
**next**  
**fix**  $t1 \ t2 \ u$   
**assume**  $\text{ind1}: \bigwedge u1. \text{Coinitial } t1 \ u1 \implies \text{Src } (t1 \sqcup u1) = \text{Src } t1$   
**assume**  $\text{ind2}: \bigwedge u2. \text{Coinitial } t2 \ u2 \implies \text{Src } (t2 \sqcup u2) = \text{Src } t2$   
**assume**  $\text{tu}: \text{Coinitial } (\lambda[t1] \bullet t2) \ u$   
**show**  $\text{Src } ((\lambda[t1] \bullet t2) \sqcup u) = \text{Src } (\lambda[t1] \bullet t2)$   
**using**  $\text{tu ind1 ind2}$   
**apply** (*cases u, simp-all*)  
**by** (*cases un-App1 u*) *auto*

**qed**

**lemma** *resid-Join*:  
**shows**  $\text{Coinitial } t \ u \implies (t \sqcup u) \setminus u = t \setminus u$   
**proof** (*induct t arbitrary: u*)  
**show**  $\bigwedge u. \text{Coinitial } \# \ u \implies (\# \sqcup u) \setminus u = \# \setminus u$   
**by** *auto*  
**show**  $\bigwedge x u. \text{Coinitial } \langle\langle x \rangle\rangle \ u \implies (\langle\langle x \rangle\rangle \sqcup u) \setminus u = \langle\langle x \rangle\rangle \setminus u$   
**by** *auto*  
**fix**  $t \ u$   
**assume**  $\text{ind}: \bigwedge u. \text{Coinitial } t \ u \implies (t \sqcup u) \setminus u = t \setminus u$   
**assume**  $\text{tu}: \text{Coinitial } \lambda[t] \ u$

```

show  $(\lambda[t] \sqcup u) \setminus u = \lambda[t] \setminus u$ 
  using tu ind
  by (cases u) auto
next
fix t1 t2 u
assume ind1:  $\bigwedge u1. \text{Coinitial } t1 \ u1 \implies (t1 \sqcup u1) \setminus u1 = t1 \setminus u1$ 
assume ind2:  $\bigwedge u2. \text{Coinitial } t2 \ u2 \implies (t2 \sqcup u2) \setminus u2 = t2 \setminus u2$ 
assume tu: Coinitial  $(t1 \circ t2) \ u$ 
show  $(t1 \circ t2 \sqcup u) \setminus u = (t1 \circ t2) \setminus u$ 
  using tu ind1 ind2 Coinitial-iff-Con
  apply (cases u, simp-all)
proof -
  fix u1 u2
  assume u:  $u = \lambda[u1] \bullet u2$ 
  have t2u2:  $t2 \frown u2$ 
    using Arr-not-Nil Arr-resid tu u by simp
  have t1u1: Coinitial  $(un-Lam \ t1 \sqcup u1) \ u1$ 
  proof -
    have Arr  $(un-Lam \ t1 \sqcup u1)$ 
      by (metis Arr.simps(3-5) Coinitial-iff-Con Con-implies-is-Lam-iff-is-Lam
        Join.simps(2) Src.simps(3-5) ind1 lambda.collapse(2) lambda.disc(8)
        lambda.sel(3) tu u)
    thus ?thesis
      using Src-Join
      by (metis Arr.simps(3-5) Coinitial-iff-Con Con-implies-is-Lam-iff-is-Lam
        Src.simps(3-5) lambda.collapse(2) lambda.disc(8) lambda.sel(2-3) tu u)
  qed
  have un-Lam  $t1 \frown u1$ 
    using t1u1
    by (metis Coinitial-iff-Con Con-implies-is-Lam-iff-is-Lam ConD(4) lambda.collapse(2)
      lambda.disc(8) resid.simps(2) tu u)
  thus  $(t1 \circ t2 \sqcup \lambda[u1] \bullet u2) \setminus (\lambda[u1] \bullet u2) = (t1 \circ t2) \setminus (\lambda[u1] \bullet u2)$ 
    using u tu t1u1 t2u2 ind1 ind2
    apply (cases t1, auto)
  proof -
    fix v
    assume v:  $t1 = \lambda[v]$ 
    show subst  $(t2 \setminus u2) ((v \sqcup u1) \setminus u1) = \text{subst } (t2 \setminus u2) (v \setminus u1)$ 
    proof -
      have subst  $(t2 \setminus u2) ((v \sqcup u1) \setminus u1) = (t1 \circ t2 \sqcup \lambda[u1] \bullet u2) \setminus (\lambda[u1] \bullet u2)$ 
        by (simp add: Coinitial-iff-Con ind2 t2u2 v)
      also have  $\dots = (t1 \circ t2) \setminus (\lambda[u1] \bullet u2)$ 
      proof -
        have  $(t1 \circ t2 \sqcup \lambda[u1] \bullet u2) \setminus (\lambda[u1] \bullet u2) =$ 
           $(\lambda[(v \sqcup u1)] \bullet (t2 \sqcup u2)) \setminus (\lambda[u1] \bullet u2)$ 
          using v by simp
        also have  $\dots = \text{subst } (t2 \setminus u2) ((v \sqcup u1) \setminus u1)$ 
          by (simp add: Coinitial-iff-Con ind2 t2u2)
        also have  $\dots = \text{subst } (t2 \setminus u2) (v \setminus u1)$ 

```

```

proof –
  have  $(t1 \sqcup \lambda[u1]) \setminus \lambda[u1] = t1 \setminus \lambda[u1]$ 
    using  $u \ tu \ ind1$  by  $simp$ 
  thus  $?thesis$ 
    using  $\langle un-Lam \ t1 \setminus u1 \neq \# \rangle \ t1u1 \ v$  by  $force$ 
qed
also have  $\dots = (t1 \circ t2) \setminus (\lambda[u1] \bullet u2)$ 
  using  $tu \ u \ v$  by  $force$ 
finally show  $?thesis$  by  $blast$ 
qed
also have  $\dots = subst \ (t2 \setminus u2) \ (v \setminus u1)$ 
  by  $(simp \ add: \ t2u2 \ v)$ 
finally show  $?thesis$  by  $auto$ 
qed
qed
next
fix  $t1 \ t2 \ u$ 
assume  $ind1: \bigwedge u1. \ Coinitial \ t1 \ u1 \implies (t1 \sqcup u1) \setminus u1 = t1 \setminus u1$ 
assume  $ind2: \bigwedge u2. \ Coinitial \ t2 \ u2 \implies (t2 \sqcup u2) \setminus u2 = t2 \setminus u2$ 
assume  $tu: \ Coinitial \ (\lambda[t1] \bullet t2) \ u$ 
show  $(\lambda[t1] \bullet t2) \sqcup u \setminus u = (\lambda[t1] \bullet t2) \setminus u$ 
  using  $tu \ ind1 \ ind2 \ Coinitial-iff-Con$ 
  apply  $(cases \ u, \ simp-all)$ 
proof –
  fix  $u1 \ u2$ 
assume  $u: u = u1 \circ u2$ 
show  $(\lambda[t1] \bullet t2) \sqcup u1 \circ u2 \setminus (u1 \circ u2) = (\lambda[t1] \bullet t2) \setminus (u1 \circ u2)$ 
  using  $ind1 \ ind2 \ tu \ u$ 
  by  $(cases \ u1) \ auto$ 
qed
qed

lemma  $prfx-Join$ :
shows  $Coinitial \ t \ u \implies u \lesssim t \sqcup u$ 
proof  $(induct \ t \ arbitrary: \ u)$ 
  show  $\bigwedge u. \ Coinitial \ \# \ u \implies u \lesssim \# \sqcup u$ 
    by  $simp$ 
  show  $\bigwedge x \ u. \ Coinitial \ \langle x \rangle \ u \implies u \lesssim \langle x \rangle \sqcup u$ 
    by  $auto$ 
  fix  $t \ u$ 
assume  $ind: \bigwedge u. \ Coinitial \ t \ u \implies u \lesssim t \sqcup u$ 
assume  $tu: \ Coinitial \ \lambda[t] \ u$ 
show  $u \lesssim \lambda[t] \sqcup u$ 
  using  $tu \ ind$ 
  apply  $(cases \ u, \ auto)$ 
  by  $force$ 
next
fix  $t1 \ t2 \ u$ 

```

```

assume  $ind1: \bigwedge u1. \text{Coinitial } t1 \ u1 \implies u1 \lesssim t1 \sqcup u1$ 
assume  $ind2: \bigwedge u2. \text{Coinitial } t2 \ u2 \implies u2 \lesssim t2 \sqcup u2$ 
assume  $tu: \text{Coinitial } (t1 \circ t2) \ u$ 
show  $u \lesssim t1 \circ t2 \sqcup u$ 
  using  $tu \ ind1 \ ind2 \ \text{Coinitial-iff-Con}$ 
  apply ( $\text{cases } u, \text{simp-all}$ )
  apply ( $\text{metis Ide.simps}(1)$ )
proof –
  fix  $u1 \ u2$ 
  assume  $u: u = \lambda[u1] \bullet u2$ 
  assume  $1: \text{Arr } t1 \wedge \text{Arr } t2 \wedge \text{Arr } u1 \wedge \text{Arr } u2 \wedge \text{Src } t1 = \lambda[\text{Src } u1] \wedge \text{Src } t2 = \text{Src } u2$ 
  have  $2: u1 \frown \text{un-Lam } t1 \sqcup u1$ 
    by ( $\text{metis } 1 \ \text{Coinitial-iff-Con} \ \text{Con-implies-is-Lam-iff-is-Lam} \ \text{Con-Arr-Src}(2)$ )
     $\text{lambda.collapse}(2) \ \text{lambda.disc}(8) \ \text{resid.simps}(2) \ \text{resid-Join}$ 
  have  $3: u2 \frown t2 \sqcup u2$ 
    by ( $\text{metis } 1 \ \text{conE} \ ind2 \ \text{null-char} \ \text{prfx-implies-con}$ )
  show  $\text{Ide } ((\lambda[u1] \bullet u2) \setminus (t1 \circ t2 \sqcup \lambda[u1] \bullet u2))$ 
  using  $u \ tu \ 1 \ 2 \ 3 \ ind1 \ ind2$ 
  apply ( $\text{cases } t1, \text{simp-all}$ )
by ( $\text{metis Arr.simps}(3) \ \text{Ide.simps}(3) \ \text{Ide-Subst Join.simps}(2) \ \text{Src.simps}(3) \ \text{resid.simps}(2)$ )
qed
next
fix  $t1 \ t2 \ u$ 
assume  $ind1: \bigwedge u1. \text{Coinitial } t1 \ u1 \implies u1 \lesssim t1 \sqcup u1$ 
assume  $ind2: \bigwedge u2. \text{Coinitial } t2 \ u2 \implies u2 \lesssim t2 \sqcup u2$ 
assume  $tu: \text{Coinitial } (\lambda[t1] \bullet t2) \ u$ 
show  $u \lesssim (\lambda[t1] \bullet t2) \sqcup u$ 
  using  $tu \ ind1 \ ind2 \ \text{Coinitial-iff-Con}$ 
  apply ( $\text{cases } u, \text{simp-all}$ )
  apply ( $\text{cases } \text{un-App1 } u, \text{simp-all}$ )
  by ( $\text{metis Ide.simps}(1) \ \text{Ide-Subst}$ ) $+$ 
qed

```

**lemma** *Ide-resid-Join*:  
**shows**  $\text{Coinitial } t \ u \implies \text{Ide } (u \setminus (t \sqcup u))$   
**using** *ide-char prfx-Join* **by** *blast*

**lemma** *join-of-Join*:  
**assumes**  $\text{Coinitial } t \ u$   
**shows**  $\text{join-of } t \ u \ (t \sqcup u)$   
**proof** (*unfold join-of-def composite-of-def, intro conjI*)  
**show**  $t \lesssim t \sqcup u$   
**using** *assms Join-sym prfx-Join [of u t]* **by** *simp*  
**show**  $u \lesssim t \sqcup u$   
**using** *assms Ide-resid-Join ide-char* **by** *simp*  
**show**  $(t \sqcup u) \setminus t \lesssim u \setminus t$   
**by** ( $\text{metis } \langle \text{prfx } u \ (\text{Join } t \ u) \rangle \ \text{arr-char} \ \text{assms} \ \text{cong-subst-right}(2) \ \text{prfx-implies-con}$   
 $\text{prfx-reflexive} \ \text{resid-Join} \ \text{con-sym} \ \text{cube}$ )  
**show**  $u \setminus t \lesssim (t \sqcup u) \setminus t$

```

    by (metis Coinitial-resid-resid ⟨prfx t (Join t u)⟩ ⟨prfx u (Join t u)⟩ conE ide-char
        null-char prfx-implies-con resid-Ide-Arr cube)
  show (t ⊔ u) \ u ≲ t \ u
    using ⟨(t ⊔ u) \ t ≲ u \ t⟩ cube by auto
  show t \ u ≲ (t ⊔ u) \ u
    by (metis ⟨(t ⊔ u) \ t ≲ u \ t⟩ assms cube resid-Join)
qed

```

```

sublocale rts-with-joins resid
  using join-of-Join
  apply unfold-locales
  by (metis Coinitial-iff-Con conE joinable-def null-char)

```

```

lemma is-rts-with-joins:
shows rts-with-joins resid
..

```

### 3.2.5 Simulations from Syntactic Constructors

Here we show that the syntactic constructors *Lam* and *App*, as well as the substitution operation *subst*, determine simulations. In addition, we show that *Beta* determines a transformation from  $App \circ (Lam \times Id)$  to *subst*.

```

abbreviation Lamext
where Lamext t ≡ if arr t then λ[t] else ‡

```

```

lemma Lam-is-simulation:
shows simulation resid resid Lamext
  using Arr-resid Coinitial-iff-Con
  by unfold-locales auto

```

```

interpretation Lam: simulation resid resid Lamext
  using Lam-is-simulation by simp

```

```

interpretation ΛxΛ: product-of-weakly-extensional-rts resid resid
..

```

```

abbreviation Appext
where Appext t ≡ if ΛxΛ.arr t then fst t ∘ snd t else ‡

```

```

lemma App-is-binary-simulation:
shows binary-simulation resid resid resid Appext
proof

```

```

  show Λt. ¬ ΛxΛ.arr t ⇒ Appext t = null
    by auto

```

```

  show Λt u. ΛxΛ.con t u ⇒ con (Appext t) (Appext u)
    using ΛxΛ.con-char Coinitial-iff-Con by auto

```

```

  show Λt u. ΛxΛ.con t u ⇒ Appext (ΛxΛ.resid t u) = Appext t \ Appext u
    using ΛxΛ.arr-char ΛxΛ.resid-def

```

```

  apply simp

```

by (*metis Arr-resid Con-implies-Arr1 Con-implies-Arr2*)  
qed

**interpretation** *App*: *binary-simulation resid resid resid App<sub>ext</sub>*  
using *App-is-binary-simulation* by *simp*

**abbreviation** *subst<sub>ext</sub>*  
**where** *subst<sub>ext</sub>*  $\equiv \lambda t.$  if  $\Lambda x \Lambda. arr\ t$  then *subst* (*snd*  $t$ ) (*fst*  $t$ ) else  $\#$

**lemma** *subst-is-binary-simulation*:  
**shows** *binary-simulation resid resid resid subst<sub>ext</sub>*  
**proof**

show  $\bigwedge t. \neg \Lambda x \Lambda. arr\ t \implies subst_{ext}\ t = null$   
by *auto*  
show  $\bigwedge t\ u. \Lambda x \Lambda. con\ t\ u \implies con\ (subst_{ext}\ t)\ (subst_{ext}\ u)$   
using  $\Lambda x \Lambda. con-char\ con-char\ Subst-not-Nil\ resid-Subst\ \Lambda x \Lambda. coinitalE$   
 $\Lambda x \Lambda. con-imp-coinitial$   
**apply** *simp*  
by *metis*  
show  $\bigwedge t\ u. \Lambda x \Lambda. con\ t\ u \implies subst_{ext}\ (\Lambda x \Lambda. resid\ t\ u) = subst_{ext}\ t \setminus subst_{ext}\ u$   
using  $\Lambda x \Lambda. arr-char\ \Lambda x \Lambda. resid-def$   
**apply** *simp*  
by (*metis Arr-resid Con-implies-Arr1 Con-implies-Arr2 resid-Subst*)

qed

**interpretation** *subst*: *binary-simulation resid resid resid subst<sub>ext</sub>*  
using *subst-is-binary-simulation* by *simp*

**interpretation** *Id*: *identity-simulation resid*

..

**interpretation** *Lam-Id*: *product-simulation resid resid resid resid Lam<sub>ext</sub> Id.map*

..

**interpretation** *App-o-Lam-Id*: *composite-simulation  $\Lambda x \Lambda. resid\ \Lambda x \Lambda. resid\ resid\ Lam-Id.map$*   
*App<sub>ext</sub>*

..

**abbreviation** *Beta<sub>ext</sub>*  
**where** *Beta<sub>ext</sub>*  $t \equiv$  if  $\Lambda x \Lambda. arr\ t$  then  $\lambda [fst\ t] \bullet snd\ t$  else  $\#$

**lemma** *Beta-is-transformation*:  
**shows** *transformation  $\Lambda x \Lambda. resid\ resid\ App-o-Lam-Id.map\ subst_{ext}\ Beta_{ext}$*   
**proof**

show  $\bigwedge a\ a'. \llbracket \Lambda x \Lambda. ide\ a; \Lambda x \Lambda. cong\ a\ a' \rrbracket \implies Beta_{ext}\ a = Beta_{ext}\ a'$   
using  $\Lambda x \Lambda. ide-backward-stable\ \Lambda x \Lambda. weak-extensionality$  by *blast*  
show  $\bigwedge f. \neg \Lambda x \Lambda. arr\ f \implies Beta_{ext}\ f = null$   
by *simp*  
show  $\bigwedge f. \Lambda x \Lambda. ide\ f \implies src\ (Beta_{ext}\ f) = App-o-Lam-Id.map\ f$   
using  $\Lambda x \Lambda. src-char\ \Lambda x \Lambda. src-ide\ Lam-Id.map-def$  by *force*  
show  $\bigwedge f. \Lambda x \Lambda. ide\ f \implies trg\ (Beta_{ext}\ f) = subst_{ext}\ f$

```

  using  $\Lambda x \Lambda. \text{trg-char } \Lambda x \Lambda. \text{trg-ide}$  by force
show  $\bigwedge a f. a \in \Lambda x \Lambda. \text{sources } f \implies$ 
       $\text{Beta}_{\text{ext}} a \setminus \text{App-o-Lam-Id.map } f = \text{Beta}_{\text{ext}} (\Lambda x \Lambda. \text{resid } a f)$ 
proof -
  fix  $a f$ 
  assume  $a \in \Lambda x \Lambda. \text{sources } f$ 
  hence  $f: \Lambda x \Lambda. \text{arr } f \wedge a = \Lambda x \Lambda. \text{src } f$ 
  using  $\Lambda x \Lambda. \text{sources-char}_{WE}$  by simp
  have  $\text{Beta}_{\text{ext}} (\Lambda x \Lambda. \text{src } f) \setminus \text{App-o-Lam-Id.map } f = \text{Beta}_{\text{ext}} (\Lambda x \Lambda. \text{trg } f)$ 
  using  $f \Lambda x \Lambda. \text{src-char } \Lambda x \Lambda. \text{trg-char } \text{Arr-Trg } \text{Arr-not-Nil } \text{Lam-Id.map-def}$  by simp
  thus  $\text{Beta}_{\text{ext}} a \setminus \text{App-o-Lam-Id.map } f = \text{Beta}_{\text{ext}} (\Lambda x \Lambda. \text{resid } a f)$ 
  using  $f$  by auto
qed
show  $\bigwedge a f. a \in \Lambda x \Lambda. \text{sources } f \implies \text{App-o-Lam-Id.map } f \setminus \text{Beta}_{\text{ext}} a = \text{subst}_{\text{ext}} f$ 
  using  $\Lambda x \Lambda. \text{src-char } \Lambda x \Lambda. \text{trg-char } \text{Lam-Id.map-def } \Lambda x \Lambda. \text{sources-char}$  by auto
show  $\bigwedge a f. a \in \Lambda x \Lambda. \text{sources } f \implies \text{join-of } (\text{Beta}_{\text{ext}} a) (\text{App-o-Lam-Id.map } f) (\text{Beta}_{\text{ext}} f)$ 
proof -
  fix  $a f$ 
  assume  $a \in \Lambda x \Lambda. \text{sources } f$ 
  hence  $f: \Lambda x \Lambda. \text{arr } f \wedge a = \Lambda x \Lambda. \text{src } f$ 
  using  $\Lambda x \Lambda. \text{sources-char}_{WE}$  by auto
  show  $\text{join-of } (\text{Beta}_{\text{ext}} a) (\text{App-o-Lam-Id.map } f) (\text{Beta}_{\text{ext}} f)$ 
  proof (intro join-ofI composite-ofI)
    show  $\text{App-o-Lam-Id.map } f \lesssim \text{Beta}_{\text{ext}} f$ 
    using  $f \text{Lam-Id.map-def } \text{Ide-Subst arr-char } \text{prfx-char } \text{prfx-reflexive}$ 
    by auto
    show  $\text{Beta}_{\text{ext}} f \setminus \text{Beta}_{\text{ext}} a \sim \text{App-o-Lam-Id.map } f \setminus \text{Beta}_{\text{ext}} a$ 
    using  $f \text{Lam-Id.map-def } \Lambda x \Lambda. \text{src-char } \text{trg-char } \text{trg-def } \Lambda x \Lambda. \text{sources-char}$ 
    apply auto
    by (metis Arr-Subst Ide-Trg)
    show  $1: \text{Beta}_{\text{ext}} f \setminus \text{App-o-Lam-Id.map } f \sim \text{Beta}_{\text{ext}} a \setminus \text{App-o-Lam-Id.map } f$ 
    using  $f \text{Lam-Id.map-def } \text{Ide-Subst } \Lambda x \Lambda. \text{src-char } \text{Ide-Trg } \text{Arr-resid } \text{Coinitial-iff-Con}$ 
      resid-Arr-self
    apply simp
    by metis
    show  $\text{Beta}_{\text{ext}} a \lesssim \text{Beta}_{\text{ext}} f$ 
    using  $f 1 \text{Lam-Id.map-def } \text{Ide-Subst } \Lambda x \Lambda. \text{src-char } \text{resid-Arr-self } \Lambda x \Lambda. \text{sources-char}$ 
    by auto
  qed
qed
qed

```

The next two results are used to show that mapping `App` over lists of transitions preserves paths.

```

lemma App-is-simulation1:
  assumes ide a
  shows simulation resid resid ( $\lambda t. \text{if arr } t \text{ then } t \circ a \text{ else } \#$ )
  proof -
    have ( $\lambda t. \text{if } \Lambda x \Lambda. \text{arr } (t, a) \text{ then } \text{fst } (t, a) \circ \text{snd } (t, a) \text{ else } \#$ ) =

```

```

      ( $\lambda t.$  if arr  $t$  then  $t \circ a$  else  $\#$ )
    using assms ide-implies-arr by force
  thus ?thesis
    using assms App.fixing-ide-gives-simulation-0 [of  $a$ ] by auto
qed

```

```

lemma App-is-simulation2:
  assumes ide a
  shows simulation resid resid ( $\lambda t.$  if arr  $t$  then  $a \circ t$  else  $\#$ )
  proof -
    have ( $\lambda t.$  if  $\Lambda x \Lambda .arr (a, t)$  then  $fst (a, t) \circ snd (a, t)$  else  $\#$ ) =
      ( $\lambda t.$  if arr  $t$  then  $a \circ t$  else  $\#$ )
    using assms ide-implies-arr by force
  thus ?thesis
    using assms App.fixing-ide-gives-simulation-1 [of  $a$ ] by auto
qed

```

### 3.2.6 Reduction and Conversion

Here we define the usual relations of reduction and conversion. Reduction is the least transitive relation that relates  $a$  to  $b$  if there exists an arrow  $t$  having  $a$  as its source and  $b$  as its target. Conversion is the least transitive relation that relates  $a$  to  $b$  if there exists an arrow  $t$  in either direction between  $a$  and  $b$ .

```

inductive red
  where Arr t  $\implies red (Src\ t) (Trg\ t)$ 
    | [red a b; red b c]  $\implies red\ a\ c$ 

```

```

inductive cnv
  where Arr t  $\implies cnv (Src\ t) (Trg\ t)$ 
    | Arr t  $\implies cnv (Trg\ t) (Src\ t)$ 
    | [cnv a b; cnv b c]  $\implies cnv\ a\ c$ 

```

```

lemma cnv-refl:
  assumes Ide a
  shows cnv a a
    using assms
    by (metis Ide-iff-Src-self Ide-implies-Arr cnv.simps)

```

```

lemma cnv-sym:
  shows cnv a b  $\implies cnv\ b\ a$ 
    apply (induct rule: cnv.induct)
    using cnv.intros(1-2)
    apply auto[2]
    using cnv.intros(3) by blast

```

```

lemma red-imp-cnvc:
  shows red a b  $\implies cnv\ a\ b$ 
    using cnv.intros(1,3) red.inducts by blast

```

**end**

We now define a locale that extends the residuation operation defined above to paths, using general results that have already been shown for paths in an RTS. In particular, we are taking advantage of the general proof of the Cube Lemma for residuation on paths.

Our immediate goal is to prove the Church-Rosser theorem, so we first prove a lemma that connects the reduction relation to paths. Later, we will prove many more facts in this locale, thereby developing a general framework for reasoning about reduction paths in the  $\lambda$ -calculus.

**locale** *reduction-paths* =

$\Lambda$ : *lambda-calculus*

**begin**

**sublocale**  $\Lambda$ : *rts*  $\Lambda$ .*resid*

**by** (*simp add*:  $\Lambda$ .*is-rts-with-joins* *rts-with-joins.axioms*(1))

**sublocale** *paths-in-weakly-extensional-rts*  $\Lambda$ .*resid*

..

**sublocale** *paths-in-confluent-rts*  $\Lambda$ .*resid*

**using** *confluent-rts.axioms*(1)  $\Lambda$ .*is-confluent-rts* *paths-in-rts-def*

*paths-in-confluent-rts.intro*

**by** *blast*

**notation**  $\Lambda$ .*resid* (**infix**  $\langle \backslash \rangle$  70)

**notation**  $\Lambda$ .*con* (**infix**  $\langle \frown \rangle$  50)

**notation**  $\Lambda$ .*prfx* (**infix**  $\langle \lesssim \rangle$  50)

**notation**  $\Lambda$ .*cong* (**infix**  $\langle \sim \rangle$  50)

**notation** *Resid* (**infix**  $\langle * \backslash * \rangle$  70)

**notation** *Resid1x* (**infix**  $\langle ^1 \backslash * \rangle$  70)

**notation** *Residx1* (**infix**  $\langle * \backslash ^1 \rangle$  70)

**notation** *con* (**infix**  $\langle * \frown * \rangle$  50)

**notation** *prfx* (**infix**  $\langle * \lesssim * \rangle$  50)

**notation** *cong* (**infix**  $\langle * \sim * \rangle$  50)

**lemma** *red-iff*:

**shows**  $\Lambda$ .*red*  $a$   $b \iff (\exists T. \text{Arr } T \wedge \text{Src } T = a \wedge \text{Trg } T = b)$

**proof**

**show**  $\Lambda$ .*red*  $a$   $b \implies \exists T. \text{Arr } T \wedge \text{Src } T = a \wedge \text{Trg } T = b$

**proof** (*induct rule*:  $\Lambda$ .*red.induct*)

**show**  $\bigwedge t. \Lambda$ .*Arr*  $t \implies \exists T. \text{Arr } T \wedge \text{Src } T = \Lambda$ .*Src*  $t \wedge \text{Trg } T = \Lambda$ .*Trg*  $t$

**by** (*metis* *Arr.simps*(2) *Srcs.simps*(2) *Srcs-simp*<sub>PWE</sub> *Trg.simps*(2)  $\Lambda$ .*trg-def*  $\Lambda$ .*arr-char*  $\Lambda$ .*resid-Arr-self*  $\Lambda$ .*sources-char* <sub>$\Lambda$</sub>  *singleton-insert-inj-eq'*)

**show**  $\bigwedge a$   $b$   $c. \llbracket \exists T. \text{Arr } T \wedge \text{Src } T = a \wedge \text{Trg } T = b;$

$\exists T. \text{Arr } T \wedge \text{Src } T = b \wedge \text{Trg } T = c \rrbracket$

$\implies \exists T. \text{Arr } T \wedge \text{Src } T = a \wedge \text{Trg } T = c$

**by** (*metis* *Arr.simps*(1) *Arr-append*<sub>PWE</sub> *Srcs-append* *Srcs-simp*<sub>PWE</sub> *Trgs-append* *Trgs-simp*<sub>PWE</sub> *singleton-insert-inj-eq'*)

**qed**

**show**  $\exists T. \text{Arr } T \wedge \text{Src } T = a \wedge \text{Trg } T = b \implies \Lambda$ .*red*  $a$   $b$

```

proof –
  have  $Arr\ T \implies \Lambda.red\ (Src\ T)\ (Trg\ T)$  for  $T$ 
  proof (induct  $T$ )
    show  $Arr\ [] \implies \Lambda.red\ (Src\ [])\ (Trg\ [])$ 
      by auto
    fix  $t\ T$ 
    assume  $ind: Arr\ T \implies \Lambda.red\ (Src\ T)\ (Trg\ T)$ 
    assume  $Arr: Arr\ (t\ \# \ T)$ 
    show  $\Lambda.red\ (Src\ (t\ \# \ T))\ (Trg\ (t\ \# \ T))$ 
    proof (cases  $T = []$ )
      show  $T = [] \implies ?thesis$ 
        using  $Arr\ arr-char\ \Lambda.red.intros(1)$  by simp
      assume  $T: T \neq []$ 
      have  $\Lambda.red\ (Src\ (t\ \# \ T))\ (\Lambda.Trg\ t)$ 
        apply simp
        by (meson  $Arr\ Arr.simps(2)\ Con-Arr-self\ Con-implies-Arr(1)\ Con-initial-left$ 
           $\Lambda.arr-char\ \Lambda.red.intros(1)$ )
      moreover have  $\Lambda.Trg\ t = Src\ T$ 
        using  $Arr$ 
        by (metis  $Arr.elims(2)\ Srcs-simp_{PWE}\ T\ \Lambda.arr-iff-has-target\ insert-subset$ 
           $\Lambda.targets-char_{\Lambda}\ list.sel(1)\ list.sel(3)\ singleton-iff$ )
      ultimately show ?thesis
        using  $ind$ 
        by (metis (no-types, opaque-lifting)  $Arr\ Con-Arr-self\ Con-implies-Arr(2)$ 
           $Resid-cons(2)\ T\ Trg.simps(3)\ \Lambda.red.intros(2)\ neq-Nil-conw$ )
    qed
  qed
  thus  $\exists T. Arr\ T \wedge Src\ T = a \wedge Trg\ T = b \implies \Lambda.red\ a\ b$ 
    by blast
  qed
qed
end

```

### 3.2.7 The Church-Rosser Theorem

```

context lambda-calculus
begin

```

```

  interpretation  $\Lambda x: reduction-paths$  .

```

```

  theorem church-rosser:

```

```

  shows  $cnv\ a\ b \implies \exists c. red\ a\ c \wedge red\ b\ c$ 

```

```

  proof (induct rule: cnv.induct)

```

```

    show  $\bigwedge t. Arr\ t \implies \exists c. red\ (Src\ t)\ c \wedge red\ (Trg\ t)\ c$ 

```

```

      by (metis  $Ide-Trg\ Ide-iff-Src-self\ Ide-iff-Trg-self\ Ide-implies-Arr\ red.intros(1)$ )

```

```

    thus  $\bigwedge t. Arr\ t \implies \exists c. red\ (Trg\ t)\ c \wedge red\ (Src\ t)\ c$ 

```

```

      by auto

```

```

    show  $\bigwedge a\ b\ c. \llbracket cnv\ a\ b; cnv\ b\ c; \exists x. red\ a\ x \wedge red\ b\ x; \exists y. red\ b\ y \wedge red\ c\ y \rrbracket$ 

```

$\implies \exists z. \text{red } a \ z \wedge \text{red } c \ z$

**proof** –  
**fix**  $a \ b \ c$   
**assume**  $\text{ind1}: \exists x. \text{red } a \ x \wedge \text{red } b \ x$  **and**  $\text{ind2}: \exists y. \text{red } b \ y \wedge \text{red } c \ y$   
**obtain**  $x$  **where**  $x: \text{red } a \ x \wedge \text{red } b \ x$   
  **using**  $\text{ind1}$  **by** *blast*  
**obtain**  $y$  **where**  $y: \text{red } b \ y \wedge \text{red } c \ y$   
  **using**  $\text{ind2}$  **by** *blast*  
**obtain**  $T1 \ U1$  **where**  $1: \Lambda x. \text{Arr } T1 \wedge \Lambda x. \text{Arr } U1 \wedge \Lambda x. \text{Src } T1 = a \wedge \Lambda x. \text{Src } U1 = b \wedge$   
    $\Lambda x. \text{Trgs } T1 = \Lambda x. \text{Trgs } U1$   
  **using**  $x \ \Lambda x. \text{red-iff [of } a \ x] \ \Lambda x. \text{red-iff [of } b \ x]$  **by** *fastforce*  
**obtain**  $T2 \ U2$  **where**  $2: \Lambda x. \text{Arr } T2 \wedge \Lambda x. \text{Arr } U2 \wedge \Lambda x. \text{Src } T2 = b \wedge \Lambda x. \text{Src } U2 = c \wedge$   
    $\Lambda x. \text{Trgs } T2 = \Lambda x. \text{Trgs } U2$   
  **using**  $y \ \Lambda x. \text{red-iff [of } b \ y] \ \Lambda x. \text{red-iff [of } c \ y]$  **by** *fastforce*  
**show**  $\exists e. \text{red } a \ e \wedge \text{red } c \ e$   
**proof** –  
  **let**  $?T = T1 \ @ \ (\Lambda x. \text{Resid } T2 \ U1)$  **and**  $?U = U2 \ @ \ (\Lambda x. \text{Resid } U1 \ T2)$   
  **have**  $3: \Lambda x. \text{Arr } ?T \wedge \Lambda x. \text{Arr } ?U \wedge \Lambda x. \text{Src } ?T = a \wedge \Lambda x. \text{Src } ?U = c$   
  **using**  $1 \ 2$   
  **by**  $(\text{metis } \Lambda x. \text{Arr-append}_{PWE} \ \Lambda x. \text{Arr-has-Trg} \ \Lambda x. \text{Con-imp-Arr-Resid} \ \Lambda x. \text{Src-append}$   
    $\Lambda x. \text{Src-resid} \ \Lambda x. \text{Srcs-simp}_{PWE} \ \Lambda x. \text{Trgs-simps}(1) \ \Lambda x. \text{Trgs-simp}_{PWE} \ \Lambda x. \text{arr}_{IP}$   
    $\Lambda x. \text{arr-append-imp-seq} \ \Lambda x. \text{confluence-ind} \ \text{singleton-insert-inj-eq}')$   
  **moreover have**  $\Lambda x. \text{Trgs } ?T = \Lambda x. \text{Trgs } ?U$   
  **using**  $1 \ 2 \ 3 \ \Lambda x. \text{Srcs-simp}_{PWE} \ \Lambda x. \text{Trgs-Resid-sym} \ \Lambda x. \text{Trgs-append} \ \Lambda x. \text{confluence-ind}$   
  **by** *presburger*  
  **ultimately have**  $\exists T \ U. \Lambda x. \text{Arr } T \wedge \Lambda x. \text{Arr } U \wedge \Lambda x. \text{Src } T = a \wedge \Lambda x. \text{Src } U = c \wedge$   
    $\Lambda x. \text{Trgs } T = \Lambda x. \text{Trgs } U$   
  **by** *blast*  
  **thus** *?thesis*  
  **using**  $\Lambda x. \text{red-iff} \ \Lambda x. \text{Arr-has-Trg}$  **by** *fastforce*  
**qed**  
**qed**  
**qed**

**corollary** *weak-diamond*:  
**assumes**  $\text{red } a \ b$  **and**  $\text{red } a \ b'$   
**obtains**  $c$  **where**  $\text{red } b \ c$  **and**  $\text{red } b' \ c$   
**proof** –  
  **have**  $\text{cnv } b \ b'$   
  **using** *assms*  
  **by**  $(\text{metis } \text{cnv.intros}(1,3) \ \text{cnv-sym} \ \text{red.induct})$   
  **thus** *?thesis*  
  **using** *that church-rosser* **by** *blast*  
**qed**

As a consequence of the Church-Rosser Theorem, the collection of all reduction paths forms a coherent normal sub-RTS of the RTS of reduction paths, and on identities the congruence induced by this normal sub-RTS coincides with convertibility. The quotient of the  $\lambda$ -calculus RTS by this congruence is then obviously discrete: the only transitions

are identities.

**interpretation** *Red*: *normal-sub-rts*  $\Lambda x. \text{Resid} \langle \text{Collect } \Lambda x. \text{Arr} \rangle$

**proof**

**show**  $\bigwedge t. t \in \text{Collect } \Lambda x. \text{Arr} \implies \Lambda x. \text{arr } t$

**by** *blast*

**show**  $\bigwedge a. \Lambda x. \text{ide } a \implies a \in \text{Collect } \Lambda x. \text{Arr}$

**using**  $\Lambda x. \text{Ide-char } \Lambda x. \text{ide-char}$  **by** *blast*

**show**  $\bigwedge u t. \llbracket u \in \text{Collect } \Lambda x. \text{Arr}; \Lambda x. \text{coinitial } t u \rrbracket \implies \Lambda x. \text{Resid } u t \in \text{Collect } \Lambda x. \text{Arr}$

**by** (*metis*  $\Lambda x. \text{Con-imp-Arr-Resid } \Lambda x. \text{Resid.simps}(1) \Lambda x. \text{con-sym } \Lambda x. \text{confluence}_P \Lambda x. \text{ide-def}$   
 $\langle \bigwedge a. \Lambda x. \text{ide } a \implies a \in \text{Collect } \Lambda x. \text{Arr} \rangle$  *mem-Collect-eq*  $\Lambda x. \text{arr-resid-iff-con}$ )

**show**  $\bigwedge u t. \llbracket u \in \text{Collect } \Lambda x. \text{Arr}; \Lambda x. \text{Resid } t u \in \text{Collect } \Lambda x. \text{Arr} \rrbracket \implies t \in \text{Collect } \Lambda x. \text{Arr}$

**by** (*metis*  $\Lambda x. \text{Arr.simps}(1) \Lambda x. \text{Con-implies-Arr}(1)$  *mem-Collect-eq*)

**show**  $\bigwedge u t. \llbracket u \in \text{Collect } \Lambda x. \text{Arr}; \Lambda x. \text{seq } u t \rrbracket \implies \exists v. \Lambda x. \text{composite-of } u t v$

**by** (*meson*  $\Lambda x. \text{obtains-composite-of}$ )

**show**  $\bigwedge u t. \llbracket u \in \text{Collect } \Lambda x. \text{Arr}; \Lambda x. \text{seq } t u \rrbracket \implies \exists v. \Lambda x. \text{composite-of } t u v$

**by** (*meson*  $\Lambda x. \text{obtains-composite-of}$ )

**qed**

**interpretation** *Red*: *coherent-normal-sub-rts*  $\Lambda x. \text{Resid} \langle \text{Collect } \Lambda x. \text{Arr} \rangle$

**apply** *unfold-locales*

**by** (*metis*  $\text{Red.Cong-closure-props}(4) \text{Red.Cong-imp-arr}(2) \Lambda x. \text{Con-imp-Arr-Resid}$   
 $\Lambda x. \text{arr-resid-iff-con } \Lambda x. \text{con-char } \Lambda x. \text{sources-resid mem-Collect-eq}$ )

**lemma** *cnv-iff-Cong*:

**assumes** *ide a* **and** *ide b*

**shows**  $\text{cnv } a b \iff \text{Red.Cong } [a] [b]$

**proof**

**assume** *1*:  $\text{Red.Cong } [a] [b]$

**obtain**  $U V$

**where**  $UV: \Lambda x. \text{Arr } U \wedge \Lambda x. \text{Arr } V \wedge \text{Red.Cong}_0 (\Lambda x. \text{Resid } [a] U) (\Lambda x. \text{Resid } [b] V)$

**using** *1*  $\text{Red.Cong-def } [of [a] [b]]$  **by** *blast*

**have**  $\text{red } a (\Lambda x. \text{Trg } U) \wedge \text{red } b (\Lambda x. \text{Trg } V)$

**by** (*metis*  $UV \Lambda x. \text{Arr.simps}(1) \Lambda x. \text{Con-implies-Arr}(1) \Lambda x. \text{Resid-single-ide}(2) \Lambda x. \text{Src-resid}$   
 $\Lambda x. \text{Trg.simps}(2) \text{assms}(1-2)$  *mem-Collect-eq* *reduction-paths.red-iff trg-ide*)

**moreover have**  $\Lambda x. \text{Trg } U = \Lambda x. \text{Trg } V$

**using**  $UV$

**by** (*metis* (*no-types, lifting*)  $\text{Red.Cong}_0\text{-imp-con } \Lambda x. \text{Arr.simps}(1) \Lambda x. \text{Con-Arr-self}$   
 $\Lambda x. \text{Con-implies-Arr}(1) \Lambda x. \text{Resid-single-ide}(2) \Lambda x. \text{Src-resid } \Lambda x. \text{cube } \Lambda x. \text{ide-def}$   
 $\Lambda x. \text{resid-arr-ide assms}(1)$  *mem-Collect-eq*)

**ultimately show**  $\text{cnv } a b$

**by** (*metis*  $\text{cnv-sym cnv.intros}(3)$  *red-imp-cnv*)

**next**

**assume** *1*:  $\text{cnv } a b$

**obtain**  $c$  **where**  $c: \text{red } a c \wedge \text{red } b c$

**using** *1* *church-rosser* **by** *blast*

**obtain**  $U$  **where**  $U: \Lambda x. \text{Arr } U \wedge \Lambda x. \text{Src } U = a \wedge \Lambda x. \text{Trg } U = c$

**using**  $c$   $\Lambda x. \text{red-iff}$  **by** *blast*

**obtain**  $V$  **where**  $V: \Lambda x. \text{Arr } V \wedge \Lambda x. \text{Src } V = b \wedge \Lambda x. \text{Trg } V = c$

**using**  $c$   $\Lambda x. \text{red-iff}$  **by** *blast*

**have**  $\Lambda x. \text{Resid1x } a \ U = c \wedge \Lambda x. \text{Resid1x } b \ V = c$   
**by** (*metis*  $U \ V \ \Lambda x. \text{Con-single-ide-ind} \ \Lambda x. \text{Ide.simps}(2) \ \Lambda x. \text{Resid1x-as-Resid}$   
 $\Lambda x. \text{Resid-Ide-Arr-ind} \ \Lambda x. \text{Resid-single-ide}(2) \ \Lambda x. \text{Srcs-simp}_{PWE} \ \Lambda x. \text{Trg.simps}(2)$   
 $\Lambda x. \text{Trg-resid-sym} \ \Lambda x. \text{ex-un-Src} \ \text{assms}(1-2) \ \text{singletonD} \ \text{trg-ide}$ )  
**hence**  $\text{Red.Cong}_0 \ (\Lambda x. \text{Resid } [a] \ U) \ (\Lambda x. \text{Resid } [b] \ V)$   
**by** (*metis*  $\text{Red.Cong}_0\text{-reflexive} \ U \ V \ \Lambda x. \text{Con-single-ideI}(1) \ \Lambda x. \text{Resid1x-as-Resid}$   
 $\Lambda x. \text{Srcs-simp}_{PWE} \ \Lambda x. \text{arr-resid} \ \Lambda x. \text{con-char} \ \text{assms}(1-2) \ \text{empty-set}$   
 $\text{list.set-intros}(1) \ \text{list.simps}(15)$ )  
**thus**  $\text{Red.Cong} \ [a] \ [b]$   
**using**  $U \ V \ \text{Red.Cong-def} \ [\text{of } [a] \ [b]]$  **by** *blast*  
**qed**

**interpretation**  $\Lambda q$ : *quotient-by-coherent-normal*  $\Lambda x. \text{Resid} \ \langle \text{Collect } \Lambda x. \text{Arr} \rangle$   
**..**

**lemma** *quotient-by-cnvc-is-discrete*:  
**shows**  $\Lambda q. \text{arr } t \longleftrightarrow \Lambda q. \text{ide } t$   
**by** (*metis*  $\text{Red.Cong-class-memb-is-arr} \ \Lambda q. \text{arr-char} \ \Lambda q. \text{ide-char}' \ \Lambda x. \text{arr-char}$   
 $\text{mem-Collect-eq} \ \text{subsetI}$ )

### 3.2.8 Normalization

A *normal form* is an identity that is not the source of any non-identity arrow.

**definition** *NF*  
**where**  $\text{NF } a \equiv \text{Ide } a \wedge (\forall t. \text{Arr } t \wedge \text{Src } t = a \longrightarrow \text{Ide } t)$

**lemma** (*in reduction-paths*) *path-from-NF-is-Ide*:  
**assumes**  $\Lambda. \text{NF } a$   
**shows**  $\llbracket \text{Arr } U; \text{Src } U = a \rrbracket \Longrightarrow \text{Ide } U$   
**proof** (*induct*  $U$ , *simp*)  
**fix**  $u \ U$   
**assume** *ind*:  $\llbracket \text{Arr } U; \text{Src } U = a \rrbracket \Longrightarrow \text{Ide } U$   
**assume**  $uU$ :  $\text{Arr } (u \ \# \ U)$  **and**  $a$ :  $\text{Src } (u \ \# \ U) = a$   
**have**  $\Lambda. \text{Ide } u$   
**using**  $\text{assms } a \ \Lambda. \text{NF-def } uU$  **by** *force*  
**thus**  $\text{Ide } (u \ \# \ U)$   
**using**  $a \ uU \ \text{ind}$   
**by** (*metis*  $\text{Arr-consE} \ \text{Con-Arr-self} \ \text{Con-imp-eq-Srcs} \ \text{Con-initial-right} \ \text{Ide.simps}(2)$   
 $\text{Ide-consI} \ \text{Srcs.simps}(2) \ \text{Srcs-simp}_{PWE} \ \Lambda. \text{Ide-iff-Src-self} \ \Lambda. \text{Ide-implies-Arr}$   
 $\Lambda. \text{sources-char}_\Lambda \ \Lambda. \text{trg-ide} \ \Lambda. \text{ide-char}$   
 $\text{singleton-insert-inj-eq}$ )

**qed**

**lemma** *NF-reduct-is-trivial*:  
**assumes**  $\text{NF } a$  **and**  $\text{red } a \ b$   
**shows**  $a = b$   
**proof** –  
**interpret**  $\Lambda x$ : *reduction-paths* .  
**have**  $\bigwedge U. \llbracket \Lambda x. \text{Arr } U; a \in \Lambda x. \text{Srcs } U \rrbracket \Longrightarrow \Lambda x. \text{Ide } U$

**using** *assms*  $\Lambda x.$ *path-from-NF-is-Ide*  
**by** (*simp add:*  $\Lambda x.$ *Srcs-simp<sub>PWE</sub>*)  
**thus** *?thesis*  
**using** *assms*  $\Lambda x.$ *red-iff*  
**by** (*metis*  $\Lambda x.$ *Con-Arr-self*  $\Lambda x.$ *Resid-Arr-Ide-ind*  $\Lambda x.$ *Src-resid*  $\Lambda x.$ *path-from-NF-is-Ide*)  
**qed**

**lemma** *NF-unique*:  
**assumes** *red*  $t$   $u$  **and** *red*  $t$   $u'$  **and** *NF*  $u$  **and** *NF*  $u'$   
**shows**  $u = u'$   
**using** *assms* *weak-diamond* *NF-reduct-is-trivial* **by** *metis*

A term is *normalizable* if it is an identity that is reducible to a normal form.

**definition** *normalizable*  
**where** *normalizable*  $a \equiv \text{Ide } a \wedge (\exists b. \text{red } a \ b \wedge \text{NF } b)$

**end**

### 3.3 Reduction Paths

In this section we develop further facts about reduction paths for the  $\lambda$ -calculus.

**context** *reduction-paths*  
**begin**

#### 3.3.1 Sources and Targets

**lemma** *Srcs-simp <sub>$\Lambda$ P</sub>*:  
**shows** *Arr*  $t \implies \text{Srcs } t = \{\Lambda.\text{Src } (\text{hd } t)\}$   
**by** (*metis* *Arr-has-Src* *Srcs.elims* *list.sel(1)*  $\Lambda.$ *sources-char <sub>$\Lambda$</sub>* )

**lemma** *Trgs-simp <sub>$\Lambda$ P</sub>*:  
**shows** *Arr*  $t \implies \text{Trgs } t = \{\Lambda.\text{Trg } (\text{last } t)\}$   
**by** (*metis* *Arr.simps(1)* *Arr-has-Trg* *Trgs.simps(2)* *Trgs-append* *append-butlast-last-id* *not-Cons-self2*  $\Lambda.$ *targets-char <sub>$\Lambda$</sub>* )

**lemma** *sources-single-Src* [*simp*]:  
**assumes**  $\Lambda.$ *Arr*  $t$   
**shows** *sources*  $[\Lambda.\text{Src } t] = \text{sources } [t]$   
**using** *assms*  
**by** (*metis*  $\Lambda.$ *Con-Arr-Src(1)*  $\Lambda.$ *Ide-Src* *Ide.simps(2)* *Resid.simps(3)* *con-char* *ideE* *ide-char* *sources-resid*  $\Lambda.$ *con-char*  $\Lambda.$ *ide-char* *list.discI*  $\Lambda.$ *resid-Arr-Src*)

**lemma** *targets-single-Trg* [*simp*]:  
**assumes**  $\Lambda.$ *Arr*  $t$   
**shows** *targets*  $[\Lambda.\text{Trg } t] = \text{targets } [t]$   
**using** *assms*  
**by** (*metis* (*full-types*) *Resid.simps(3)* *conI<sub>P</sub>*  $\Lambda.$ *Arr-Trg*  $\Lambda.$ *arr-char*  $\Lambda.$ *resid-Arr-Src*  $\Lambda.$ *resid-Src-Arr*  $\Lambda.$ *arr-resid-iff-con* *targets-resid-sym*)

**lemma** *sources-single-Trg* [*simp*]:  
**assumes**  $\Lambda.Arr\ t$   
**shows**  $sources\ [\Lambda.Trq\ t] = targets\ [t]$   
**using** *assms*  
**by** (*metis*  $\Lambda.Ide-Trq\ Ide.simps(2)\ ideE\ ide-char\ sources-resid\ \Lambda.ide-char\ targets-single-Trq$ )

**lemma** *targets-single-Src* [*simp*]:  
**assumes**  $\Lambda.Arr\ t$   
**shows**  $targets\ [\Lambda.Src\ t] = sources\ [t]$   
**using** *assms*  
**by** (*metis*  $\Lambda.Arr-Src\ \Lambda.Trq-Src\ sources-single-Src\ sources-single-Trq$ )

**lemma** *single-Src-hd-in-sources*:  
**assumes**  $Arr\ T$   
**shows**  $[\Lambda.Src\ (hd\ T)] \in sources\ T$   
**using** *assms*  
**by** (*metis*  $Arr.simps(1)\ Arr-has-Src\ Ide.simps(2)\ Resid-Arr-Src\ Srcs-simp_P\ \Lambda.source-is-ide\ conI_P\ empty-set\ ide-char\ in-sourcesI\ \Lambda.sources-char_\Lambda\ list.set-intros(1)\ list.simps(15)$ )

**lemma** *single-Trg-last-in-targets*:  
**assumes**  $Arr\ T$   
**shows**  $[\Lambda.Trq\ (last\ T)] \in targets\ T$   
**using** *assms* *targets-char\_P* *Arr-imp-arr-last* *Trgs-simp\_{\Lambda P}*  $\Lambda.Ide-Trq$  **by** *fastforce*

**lemma** *in-sources-iff*:  
**assumes**  $Arr\ T$   
**shows**  $A \in sources\ T \longleftrightarrow A\ *\sim*\ [\Lambda.Src\ (hd\ T)]$   
**using** *assms*  
**by** (*meson* *single-Src-hd-in-sources* *sources-are-cong* *sources-cong-closed*)

**lemma** *in-targets-iff*:  
**assumes**  $Arr\ T$   
**shows**  $B \in targets\ T \longleftrightarrow B\ *\sim*\ [\Lambda.Trq\ (last\ T)]$   
**using** *assms*  
**by** (*meson* *single-Trg-last-in-targets* *targets-are-cong* *targets-cong-closed*)

**lemma** *seq-imp-cong-Trg-last-Src-hd*:  
**assumes**  $seq\ T\ U$   
**shows**  $\Lambda.Trq\ (last\ T) \sim \Lambda.Src\ (hd\ U)$   
**using** *assms* *Arr-imp-arr-hd* *Arr-imp-arr-last* *Srcs-simp\_{PWE}* *Trgs-simp\_{PWE}*  
 $\Lambda.cong-reflexive\ seq-char$   
**by** (*metis* *Srcs-simp\_{\Lambda P}* *Trgs-simp\_{\Lambda P}*  $\Lambda.Arr-Trq$   $\Lambda.arr-char$  *singleton-inject*)

**lemma** *sources-char\_{\Lambda P}*:  
**shows**  $sources\ T = \{A.\ Arr\ T \wedge A\ *\sim*\ [\Lambda.Src\ (hd\ T)]\}$   
**using** *in-sources-iff* *arr-char* *sources-char\_P* **by** *auto*

**lemma** *targets-char* <sub>$\Lambda P$</sub> :

**shows**  $targets\ T = \{B.\ Arr\ T \wedge B\ *\sim*\ [\Lambda.Trg\ (last\ T)]\}$   
**using** *in-targets-iff arr-char targets-char by auto*

**lemma** *Src-hd-eqI*:

**assumes**  $T\ *\sim*\ U$

**shows**  $\Lambda.Src\ (hd\ T) = \Lambda.Src\ (hd\ U)$

**using** *assms*

**by** (*metis Con-imp-eq-Srcs Con-implies-Arr(1) Ide.simps(1) Srcs-simp <sub>$\Lambda P$</sub>  ide-char singleton-insert-inj-eq'*)

**lemma** *Trg-last-eqI*:

**assumes**  $T\ *\sim*\ U$

**shows**  $\Lambda.Trg\ (last\ T) = \Lambda.Trg\ (last\ U)$

**proof** –

**have**  $1: [\Lambda.Trg\ (last\ T)] \in targets\ T \wedge [\Lambda.Trg\ (last\ U)] \in targets\ U$

**using** *assms*

**by** (*metis Con-implies-Arr(1) Ide.simps(1) ide-char single-Trg-last-in-targets*)

**have**  $\Lambda.cong\ (\Lambda.Trg\ (last\ T))\ (\Lambda.Trg\ (last\ U))$

**by** (*metis 1 Ide.simps(2) Resid.simps(3) assms con-char cong-implies-coterminal coterminal-iff ide-char prfx-implies-con targets-are-cong*)

**moreover have**  $\Lambda.Ide\ (\Lambda.Trg\ (last\ T)) \wedge \Lambda.Ide\ (\Lambda.Trg\ (last\ U))$

**using** *1 Ide.simps(2) ide-char by blast*

**ultimately show** *?thesis*

**using**  $\Lambda.weak-extensionality$  **by** *blast*

**qed**

**lemma** *Trg-last-Src-hd-eqI*:

**assumes**  $seq\ T\ U$

**shows**  $\Lambda.Trg\ (last\ T) = \Lambda.Src\ (hd\ U)$

**using** *assms Arr-imp-arr-hd Arr-imp-arr-last  $\Lambda.Ide-Src$   $\Lambda.weak-extensionality$   $\Lambda.Ide-Trg$  seq-char seq-imp-cong-Trg-last-Src-hd*

**by** *force*

**lemma** *seqI <sub>$\Lambda P$</sub>  [intro]*:

**assumes**  $Arr\ T$  **and**  $Arr\ U$  **and**  $\Lambda.Trg\ (last\ T) = \Lambda.Src\ (hd\ U)$

**shows**  $seq\ T\ U$

**by** (*metis assms Arr-imp-arr-last Srcs-simp <sub>$\Lambda P$</sub>   $\Lambda.arr-char$   $\Lambda.targets-char_{\Lambda}$  Trgs-simp <sub>$P$</sub>  seq-char*)

**lemma** *conI <sub>$\Lambda P$</sub>  [intro]*:

**assumes**  $arr\ T$  **and**  $arr\ U$  **and**  $\Lambda.Src\ (hd\ T) = \Lambda.Src\ (hd\ U)$

**shows**  $T\ *\frown*\ U$

**using** *assms*

**by** (*simp add: Srcs-simp <sub>$\Lambda P$</sub>  arr-char con-char confluence-ind*)

### 3.3.2 Mapping Constructors over Paths

**lemma** *Arr-map-Lam*:

**assumes** *Arr T*

**shows** *Arr (map  $\Lambda.Lam T$ )*

**proof** –

**interpret** *Lam*: *simulation  $\Lambda.resid \Lambda.resid \langle \lambda t. \text{if } \Lambda.arr t \text{ then } \lambda[t] \text{ else } \# \rangle$*

**using**  *$\Lambda.Lam$ -is-simulation* **by** *simp*

**interpret** *simulation Resid Resid*

$\langle \lambda T. \text{if } Arr T \text{ then } map (\lambda t. \text{if } \Lambda.arr t \text{ then } \lambda[t] \text{ else } \#) T \text{ else } [] \rangle$

**using** *assms Lam.lifts-to-paths* **by** *blast*

**have** *map ( $\lambda t. \text{if } \Lambda.Arr t \text{ then } \lambda[t] \text{ else } \#) T = map \Lambda.Lam T$*

**using** *assms set-Arr-subset-arr* **by** *fastforce*

**thus** *?thesis*

**using** *assms preserves-reflects-arr [of T] arr-char*

**by** (*simp add:  $\langle map (\lambda t. \text{if } \Lambda.Arr t \text{ then } \lambda[t] \text{ else } \#) T = map \Lambda.Lam T \rangle$* )

**qed**

**lemma** *Arr-map-App1*:

**assumes**  *$\Lambda.Ide b$  and Arr T*

**shows** *Arr (map ( $\lambda t. t \circ b$ ) T)*

**proof** –

**interpret** *App1*: *simulation  $\Lambda.resid \Lambda.resid \langle \lambda t. \text{if } \Lambda.arr t \text{ then } t \circ b \text{ else } \# \rangle$*

**using** *assms  $\Lambda.App$ -is-simulation1 [of b]* **by** *simp*

**interpret** *simulation Resid Resid*

$\langle \lambda T. \text{if } Arr T \text{ then } map (\lambda t. \text{if } \Lambda.arr t \text{ then } t \circ b \text{ else } \#) T \text{ else } [] \rangle$

**using** *assms App1.lifts-to-paths* **by** *blast*

**have** *map ( $\lambda t. \text{if } \Lambda.arr t \text{ then } t \circ b \text{ else } \#) T = map (\lambda t. t \circ b) T$*

**using** *assms set-Arr-subset-arr* **by** *auto*

**thus** *?thesis*

**using** *assms preserves-reflects-arr arr-char*

**by** (*metis (mono-tags, lifting)*)

**qed**

**lemma** *Arr-map-App2*:

**assumes**  *$\Lambda.Ide a$  and Arr T*

**shows** *Arr (map ( $\Lambda.App a$ ) T)*

**proof** –

**interpret** *App2*: *simulation  $\Lambda.resid \Lambda.resid \langle \lambda u. \text{if } \Lambda.arr u \text{ then } a \circ u \text{ else } \# \rangle$*

**using** *assms  $\Lambda.App$ -is-simulation2* **by** *simp*

**interpret** *simulation Resid Resid*

$\langle \lambda T. \text{if } Arr T \text{ then } map (\lambda u. \text{if } \Lambda.arr u \text{ then } a \circ u \text{ else } \#) T \text{ else } [] \rangle$

**using** *assms App2.lifts-to-paths* **by** *blast*

**have** *map ( $\lambda u. \text{if } \Lambda.arr u \text{ then } a \circ u \text{ else } \#) T = map (\lambda u. a \circ u) T$*

**using** *assms set-Arr-subset-arr* **by** *auto*

**thus** *?thesis*

**using** *assms preserves-reflects-arr arr-char*

**by** (*metis (mono-tags, lifting)*)

**qed**

**interpretation**  $\Lambda_{Lam}$ : *source-replete-sub-rts*  $\Lambda.resid \langle \lambda t. \Lambda.Arr t \wedge \Lambda.is-Lam t \rangle$

**proof**

**show**  $\bigwedge t. \Lambda.Arr t \wedge \Lambda.is-Lam t \implies \Lambda.arr t$

**by** *blast*

**show**  $\bigwedge t. \Lambda.Arr t \wedge \Lambda.is-Lam t \implies \Lambda.sources t \subseteq \{t. \Lambda.Arr t \wedge \Lambda.is-Lam t\}$

**by** *auto*

**show**  $\llbracket \Lambda.Arr t \wedge \Lambda.is-Lam t; \Lambda.Arr u \wedge \Lambda.is-Lam u; \Lambda.con t u \rrbracket$   
 $\implies \Lambda.Arr (t \setminus u) \wedge \Lambda.is-Lam (t \setminus u)$

**for**  $t u$

**apply** (*cases t; cases u*)

**apply** *simp-all*

**using**  $\Lambda.Coinitial-resid-resid$

**by** *presburger*

**qed**

**interpretation** *un-Lam*: *simulation*  $\Lambda_{Lam}.resid \Lambda.resid$

$\langle \lambda t. \text{if } \Lambda_{Lam}.arr t \text{ then } \Lambda.un-Lam t \text{ else } \sharp \rangle$

**proof**

**let**  $?un-Lam = \lambda t. \text{if } \Lambda_{Lam}.arr t \text{ then } \Lambda.un-Lam t \text{ else } \sharp$

**show**  $\bigwedge t. \neg \Lambda_{Lam}.arr t \implies ?un-Lam t = \Lambda.null$

**by** *auto*

**show**  $\bigwedge t u. \Lambda_{Lam}.con t u \implies \Lambda.con (?un-Lam t) (?un-Lam u)$

**by** (*auto simp add:  $\Lambda_{Lam}.con-char$* )

**show**  $\bigwedge t u. \Lambda_{Lam}.con t u \implies ?un-Lam (\Lambda_{Lam}.resid t u) = ?un-Lam t \setminus ?un-Lam u$

**using**  $\Lambda_{Lam}.resid-closed \Lambda_{Lam}.resid-def$  **by** (*auto simp add:  $\Lambda_{Lam}.con-char$* )

**qed**

**lemma** *Arr-map-un-Lam*:

**assumes** *Arr T and set  $T \subseteq Collect \Lambda.is-Lam$*

**shows** *Arr (map  $\Lambda.un-Lam T$ )*

**proof** –

**have** *map ( $\lambda t. \text{if } \Lambda_{Lam}.arr t \text{ then } \Lambda.un-Lam t \text{ else } \sharp) T = map \Lambda.un-Lam T$*

**using** *assms set-Arr-subset-arr by (auto simp add:  $\Lambda_{Lam}.arr-char$ )*

**thus** *?thesis*

**using** *assms*

**by** (*metis (no-types, lifting)  $\Lambda_{Lam}.path-reflection \Lambda.arr-char mem-Collect-eq$  set-Arr-subset-arr subset-code(1) un-Lam.preserves-paths*)

**qed**

**interpretation**  $\Lambda_{App}$ : *source-replete-sub-rts*  $\Lambda.resid \langle \lambda t. \Lambda.Arr t \wedge \Lambda.is-App t \rangle$

**proof**

**show**  $\bigwedge t. \Lambda.Arr t \wedge \Lambda.is-App t \implies \Lambda.arr t$

**by** *blast*

**show**  $\bigwedge t. \Lambda.Arr t \wedge \Lambda.is-App t \implies \Lambda.sources t \subseteq \{t. \Lambda.Arr t \wedge \Lambda.is-App t\}$

**by** *auto*

**show**  $\llbracket \Lambda.Arr t \wedge \Lambda.is-App t; \Lambda.Arr u \wedge \Lambda.is-App u; \Lambda.con t u \rrbracket$

$\implies \Lambda.Arr (t \setminus u) \wedge \Lambda.is-App (t \setminus u)$

**for**  $t u$

**using**  $\Lambda.Arr-resid$

by (cases t; cases u) auto  
qed

**interpretation** *un-App1: simulation*  $\Lambda_{App}.resid \Lambda.resid$   
 $\langle \lambda t. \text{if } \Lambda_{App}.arr \ t \ \text{then } \Lambda.un-App1 \ t \ \text{else } \# \rangle$

**proof**  
 let  $?un-App1 = \lambda t. \text{if } \Lambda_{App}.arr \ t \ \text{then } \Lambda.un-App1 \ t \ \text{else } \#$   
 show  $\bigwedge t. \neg \Lambda_{App}.arr \ t \implies ?un-App1 \ t = \Lambda.null$   
 by auto  
 show  $\bigwedge t \ u. \Lambda_{App}.con \ t \ u \implies \Lambda.con \ (?un-App1 \ t) \ (?un-App1 \ u)$   
 by (auto simp add:  $\Lambda_{App}.con-char$ )  
 show  $\Lambda_{App}.con \ t \ u \implies ?un-App1 \ (\Lambda_{App}.resid \ t \ u) = ?un-App1 \ t \ \setminus \ ?un-App1 \ u$   
 for t u  
 using  $\Lambda_{App}.resid-def \ \Lambda.Arr-resid$   
 by (cases t; cases u) (auto simp add:  $\Lambda_{App}.con-char$ )  
 qed

**interpretation** *un-App2: simulation*  $\Lambda_{App}.resid \Lambda.resid$   
 $\langle \lambda t. \text{if } \Lambda_{App}.arr \ t \ \text{then } \Lambda.un-App2 \ t \ \text{else } \# \rangle$

**proof**  
 let  $?un-App2 = \lambda t. \text{if } \Lambda_{App}.arr \ t \ \text{then } \Lambda.un-App2 \ t \ \text{else } \#$   
 show  $\bigwedge t. \neg \Lambda_{App}.arr \ t \implies ?un-App2 \ t = \Lambda.null$   
 by auto  
 show  $\bigwedge t \ u. \Lambda_{App}.con \ t \ u \implies \Lambda.con \ (?un-App2 \ t) \ (?un-App2 \ u)$   
 by (auto simp add:  $\Lambda_{App}.con-char$ )  
 show  $\Lambda_{App}.con \ t \ u \implies ?un-App2 \ (\Lambda_{App}.resid \ t \ u) = ?un-App2 \ t \ \setminus \ ?un-App2 \ u$   
 for t u  
 using  $\Lambda_{App}.resid-def \ \Lambda.Arr-resid$   
 by (cases t; cases u) (auto simp add:  $\Lambda_{App}.con-char$ )  
 qed

**lemma** *Arr-map-un-App1:*  
**assumes** *Arr T and set*  $T \subseteq Collect \ \Lambda.is-App$   
**shows** *Arr* (map  $\Lambda.un-App1 \ T$ )

**proof** –  
**interpret**  $P_{App}: paths-in-rts \ \Lambda_{App}.resid$   
 ..  
**interpret** *un-App1: simulation*  $P_{App}.Resid \ Resid$   
 $\langle \lambda T. \text{if } P_{App}.Arr \ T \ \text{then}$   
 $\quad \text{map } (\lambda t. \text{if } \Lambda_{App}.arr \ t \ \text{then } \Lambda.un-App1 \ t \ \text{else } \#) \ T$   
 $\quad \text{else } [] \rangle$   
 using *un-App1.lifts-to-paths* by simp  
 have 1:  $\text{map } (\lambda t. \text{if } \Lambda_{App}.arr \ t \ \text{then } \Lambda.un-App1 \ t \ \text{else } \#) \ T = \text{map } \Lambda.un-App1 \ T$   
 using *assms set-Arr-subset-arr* by (auto simp add:  $\Lambda_{App}.arr-char$ )  
 have 2:  $P_{App}.Arr \ T$   
 using *assms set-Arr-subset-arr*  $\Lambda_{App}.path-reflection$  [of T] by blast  
 hence *arr* (if  $P_{App}.Arr \ T$  then map  $(\lambda t. \text{if } \Lambda_{App}.arr \ t \ \text{then } \Lambda.un-App1 \ t \ \text{else } \#) \ T$  else [])  
 using *un-App1.preserves-reflects-arr* [of T] by blast  
 hence *Arr* (if  $P_{App}.Arr \ T$  then map  $(\lambda t. \text{if } \Lambda_{App}.arr \ t \ \text{then } \Lambda.un-App1 \ t \ \text{else } \#) \ T$  else [])

**using** *arr-char* **by** *auto*  
**hence** *Arr* (if  $P_{App}.Arr\ T$  then  $map\ \Lambda.un-App1\ T$  else  $\square$ )  
**using** *1* **by** *metis*  
**thus** *?thesis*  
**using** *2* **by** *simp*  
**qed**

**lemma** *Arr-map-un-App2*:  
**assumes** *Arr*  $T$  **and**  $set\ T \subseteq Collect\ \Lambda.is-App$   
**shows** *Arr* ( $map\ \Lambda.un-App2\ T$ )  
**proof** –  
**interpret**  $P_{App}$ : *paths-in-rts*  $\Lambda_{App}.resid$   
**..**  
**interpret** *un-App2*: *simulation*  $P_{App}.Resid\ Resid$   
 $\langle \lambda T. if\ P_{App}.Arr\ T\ then$   
 $map\ (\lambda t. if\ \Lambda_{App}.arr\ t\ then\ \Lambda.un-App2\ t\ else\ \#)\ T$   
 $else\ \square \rangle$   
**using** *un-App2.lifts-to-paths* **by** *simp*  
**have** *1*:  $map\ (\lambda t. if\ \Lambda_{App}.arr\ t\ then\ \Lambda.un-App2\ t\ else\ \#)\ T = map\ \Lambda.un-App2\ T$   
**using** *assms set-Arr-subset-arr* **by** (*auto simp add: \Lambda\_{App}.arr-char*)  
**have** *2*:  $P_{App}.Arr\ T$   
**using** *assms set-Arr-subset-arr \Lambda\_{App}.path-reflection [of T]* **by** *blast*  
**hence** *arr* (if  $P_{App}.Arr\ T$  then  $map\ (\lambda t. if\ \Lambda_{App}.arr\ t\ then\ \Lambda.un-App2\ t\ else\ \#)\ T$  else  $\square$ )  
**using** *un-App2.preserves-reflects-arr [of T]* **by** *blast*  
**hence** *Arr* (if  $P_{App}.Arr\ T$  then  $map\ (\lambda t. if\ \Lambda_{App}.arr\ t\ then\ \Lambda.un-App2\ t\ else\ \#)\ T$  else  $\square$ )  
**using** *arr-char* **by** *blast*  
**hence** *Arr* (if  $P_{App}.Arr\ T$  then  $map\ \Lambda.un-App2\ T$  else  $\square$ )  
**using** *1* **by** *metis*  
**thus** *?thesis*  
**using** *2* **by** *simp*  
**qed**

**lemma** *map-App-map-un-App1*:  
**shows**  $\llbracket Arr\ U; set\ U \subseteq Collect\ \Lambda.is-App; \Lambda.Ide\ b; \Lambda.un-App2\ 'set\ U \subseteq \{b\} \rrbracket \implies$   
 $map\ (\lambda t. \Lambda.App\ t\ b)\ (map\ \Lambda.un-App1\ U) = U$   
**by** (*induct U*) *auto*

**lemma** *map-App-map-un-App2*:  
**shows**  $\llbracket Arr\ U; set\ U \subseteq Collect\ \Lambda.is-App; \Lambda.Ide\ a; \Lambda.un-App1\ 'set\ U \subseteq \{a\} \rrbracket \implies$   
 $map\ (\Lambda.App\ a)\ (map\ \Lambda.un-App2\ U) = U$   
**by** (*induct U*) *auto*

**lemma** *map-Lam-Resid*:  
**assumes** *coinitial*  $T\ U$   
**shows**  $map\ \Lambda.Lam\ (T\ *\ \backslash\ * \ U) = map\ \Lambda.Lam\ T\ *\ \backslash\ * \ map\ \Lambda.Lam\ U$   
**proof** –  
**interpret** *Lam*: *simulation*  $\Lambda.resid\ \Lambda.resid\ \langle \lambda t. if\ \Lambda.arr\ t\ then\ \lambda[t]\ else\ \# \rangle$   
**using**  $\Lambda.Lam-is-simulation$  **by** *simp*  
**interpret** *Lamx*: *simulation*  $Resid\ Resid$

$\langle \lambda T. \text{ if Arr } T \text{ then}$   
 $\quad \text{map } (\lambda t. \text{ if } \Lambda.\text{arr } t \text{ then } \lambda[t] \text{ else } \#) T$   
 $\quad \text{else } [] \rangle$   
**using** *Lam.lifts-to-paths* **by** *simp*  
**have**  $\bigwedge T. \text{ Arr } T \implies \text{map } (\lambda t. \text{ if } \Lambda.\text{arr } t \text{ then } \lambda[t] \text{ else } \#) T = \text{map } \Lambda.\text{Lam } T$   
**using** *set-Arr-subset-arr* **by** *auto*  
**moreover have**  $\text{Arr } (T \text{ }^* \setminus^* U)$   
**using** *assms confluence<sub>P</sub> Con-imp-Arr-Resid con-char* **by** *force*  
**moreover have**  $T \text{ }^* \frown^* U$   
**using** *assms confluence* **by** *simp*  
**moreover have**  $\text{Arr } T \wedge \text{Arr } U$   
**using** *assms arr-char* **by** *auto*  
**ultimately show** *?thesis*  
**using** *assms Lam.x.preserves-resid [of T U]* **by** *presburger*  
**qed**

**lemma** *map-App1-Resid*:

**assumes**  $\Lambda.\text{Ide } x$  **and** *coinitial*  $T U$

**shows**  $\text{map } (\Lambda.\text{App } x) (T \text{ }^* \setminus^* U) = \text{map } (\Lambda.\text{App } x) T \text{ }^* \setminus^* \text{map } (\Lambda.\text{App } x) U$

**proof** –

**interpret** *App*: *simulation*  $\Lambda.\text{resid } \Lambda.\text{resid}$   $\langle \lambda t. \text{ if } \Lambda.\text{arr } t \text{ then } x \circ t \text{ else } \# \rangle$

**using** *assms*  $\Lambda.\text{App-is-simulation2}$  **by** *simp*

**interpret** *Appx*: *simulation* *Resid* *Resid*

$\langle \lambda T. \text{ if Arr } T \text{ then map } (\lambda t. \text{ if } \Lambda.\text{arr } t \text{ then } x \circ t \text{ else } \#) T \text{ else } [] \rangle$

**using** *App.lifts-to-paths* **by** *simp*

**have**  $\bigwedge T. \text{ Arr } T \implies \text{map } (\lambda t. \text{ if } \Lambda.\text{arr } t \text{ then } x \circ t \text{ else } \#) T = \text{map } (\Lambda.\text{App } x) T$

**using** *set-Arr-subset-arr* **by** *auto*

**moreover have**  $\text{Arr } (T \text{ }^* \setminus^* U)$

**using** *assms confluence<sub>P</sub> Con-imp-Arr-Resid con-char* **by** *force*

**moreover have**  $T \text{ }^* \frown^* U$

**using** *assms confluence* **by** *simp*

**moreover have**  $\text{Arr } T \wedge \text{Arr } U$

**using** *assms arr-char* **by** *auto*

**ultimately show** *?thesis*

**using** *assms Appx.preserves-resid [of T U]* **by** *presburger*

**qed**

**lemma** *map-App2-Resid*:

**assumes**  $\Lambda.\text{Ide } x$  **and** *coinitial*  $T U$

**shows**  $\text{map } (\lambda t. t \circ x) (T \text{ }^* \setminus^* U) = \text{map } (\lambda t. t \circ x) T \text{ }^* \setminus^* \text{map } (\lambda t. t \circ x) U$

**proof** –

**interpret** *App*: *simulation*  $\Lambda.\text{resid } \Lambda.\text{resid}$   $\langle \lambda t. \text{ if } \Lambda.\text{arr } t \text{ then } t \circ x \text{ else } \# \rangle$

**using** *assms*  $\Lambda.\text{App-is-simulation1}$  **by** *simp*

**interpret** *Appx*: *simulation* *Resid* *Resid*

$\langle \lambda T. \text{ if Arr } T \text{ then map } (\lambda t. \text{ if } \Lambda.\text{arr } t \text{ then } t \circ x \text{ else } \#) T \text{ else } [] \rangle$

**using** *App.lifts-to-paths* **by** *simp*

**have**  $\bigwedge T. \text{ Arr } T \implies \text{map } (\lambda t. \text{ if } \Lambda.\text{arr } t \text{ then } t \circ x \text{ else } \#) T = \text{map } (\lambda t. t \circ x) T$

**using** *set-Arr-subset-arr* **by** *auto*

**moreover have**  $\text{Arr } (T \text{ }^* \setminus^* U)$

**using** *assms confluence<sub>P</sub> Con-imp-Arr-Resid con-char* **by** *force*  
**moreover have**  $T \sim^* U$   
**using** *assms confluence* **by** *simp*  
**moreover have**  $\text{Arr } T \wedge \text{Arr } U$   
**using** *assms arr-char* **by** *auto*  
**ultimately show** *?thesis*  
**using** *assms Appx.preserves-resid [of T U]* **by** *presburger*  
**qed**

**lemma** *cong-map-Lam*:  
**shows**  $T \sim^* U \implies \text{map } \Lambda.\text{Lam } T \sim^* \text{map } \Lambda.\text{Lam } U$   
**apply** (*induct U arbitrary: T*)  
**apply** (*simp add: ide-char*)  
**by** (*metis map-Lam-Resid cong-implies-coinitial cong-reflexive ideE*  
*map-is-Nil-conv Con-imp-Arr-Resid arr-char*)

**lemma** *cong-map-App1*:  
**shows**  $[\Lambda.\text{Ide } x; T \sim^* U] \implies \text{map } (\Lambda.\text{App } x) T \sim^* \text{map } (\Lambda.\text{App } x) U$   
**apply** (*induct U arbitrary: x T*)  
**apply** (*simp add: ide-char*)  
**apply** (*intro conjI*)  
**by** (*metis Nil-is-map-conv arr-resid-iff-con con-char con-imp-coinitial*  
*cong-reflexive ideE map-App1-Resid*)+

**lemma** *cong-map-App2*:  
**shows**  $[\Lambda.\text{Ide } x; T \sim^* U] \implies \text{map } (\lambda X. X \circ x) T \sim^* \text{map } (\lambda X. X \circ x) U$   
**apply** (*induct U arbitrary: x T*)  
**apply** (*simp add: ide-char*)  
**apply** (*intro conjI*)  
**by** (*metis Nil-is-map-conv arr-resid-iff-con con-char cong-implies-coinitial*  
*cong-reflexive ide-def arr-char ideE map-App2-Resid*)+

### 3.3.3 Decomposition of ‘App Paths’

The following series of results is aimed at showing that a reduction path, all of whose transitions have *App* as their top-level constructor, can be factored up to congruence into a reduction path in which only the “rator” components are reduced, followed by a reduction path in which only the “rand” components are reduced.

**lemma** *orthogonal-App-single-single*:  
**assumes**  $\Lambda.\text{Arr } t$  **and**  $\Lambda.\text{Arr } u$   
**shows**  $[\Lambda.\text{Src } t \circ u] \sim^* [t \circ \Lambda.\text{Src } u] = [\Lambda.\text{Trg } t \circ u]$   
**and**  $[t \circ \Lambda.\text{Src } u] \sim^* [\Lambda.\text{Src } t \circ u] = [t \circ \Lambda.\text{Trg } u]$   
**using** *assms arr-char  $\Lambda.\text{Arr-not-Nil}$*  **by** *auto*

**lemma** *orthogonal-App-single-Arr*:  
**shows**  $[\text{Arr } [t]; \text{Arr } U] \implies$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U \sim^* [t \circ \Lambda.\text{Src } (\text{hd } U)] = \text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } t)) U \wedge$   
 $[t \circ \Lambda.\text{Src } (\text{hd } U)] \sim^* \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U = [t \circ \Lambda.\text{Trg } (\text{last } U)]$   
**proof** (*induct U arbitrary: t*)

**show**  $\bigwedge t. \llbracket \text{Arr } [t]; \text{Arr } [] \rrbracket \implies$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) [] \text{ }^*\backslash^* [t \circ \Lambda.\text{Src } (\text{hd } [])] = \text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } t)) [] \wedge$   
 $[t \circ \Lambda.\text{Src } (\text{hd } [])] \text{ }^*\backslash^* \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) [] = [t \circ \Lambda.\text{Trg } (\text{last } [])]$   
**by** *fastforce*  
**fix**  $t \ u \ U$   
**assume**  $\text{ind}: \bigwedge t. \llbracket \text{Arr } [t]; \text{Arr } U \rrbracket \implies$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U \text{ }^*\backslash^* [t \circ \Lambda.\text{Src } (\text{hd } U)] =$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } t)) U \wedge$   
 $[t \circ \Lambda.\text{Src } (\text{hd } U)] \text{ }^*\backslash^* \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U = [t \circ \Lambda.\text{Trg } (\text{last } U)]$   
**assume**  $t: \text{Arr } [t]$   
**assume**  $uU: \text{Arr } (u \# U)$   
**show**  $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) (u \# U) \text{ }^*\backslash^* [t \circ \Lambda.\text{Src } (\text{hd } (u \# U))] =$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } t)) (u \# U) \wedge$   
 $[t \circ \Lambda.\text{Src } (\text{hd } (u \# U))] \text{ }^*\backslash^* \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) (u \# U) =$   
 $[t \circ \Lambda.\text{Trg } (\text{last } (u \# U))]$   
**proof** (*cases*  $U = []$ )  
**show**  $U = [] \implies ?thesis$   
**using**  $t \ uU$  *orthogonal-App-single-single* **by** *simp*  
**assume**  $U: U \neq []$   
**have** 2: *coinitial*  $([\Lambda.\text{Src } t \circ u] @ \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U) [t \circ \Lambda.\text{Src } u]$   
**proof** (*intro coinitialI'*)  
**show** 3: *arr*  $([\Lambda.\text{Src } t \circ u] @ \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U)$   
**using**  $t \ uU$   
**by** (*metis Arr-iff-Con-self Arr-map-App2 Con-rec(1) append-Cons append-Nil*  
 $\Lambda.\text{Con-implies-Arr2 } \Lambda.\text{Ide-Src } \Lambda.\text{con-char list.simps(9) arr-char}$ )  
**show** *src*  $([\Lambda.\text{Src } t \circ u] @ \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U) \sim \text{src } [t \circ \Lambda.\text{Src } u]$   
**proof** –  
**have** *seq*  $([\Lambda.\text{Src } t \circ u] (\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U))$   
**using**  $U$  3 *arr-append-imp-seq* **by** *force*  
**hence** *sources*  $([\Lambda.\text{Src } t \circ u] @ \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U) = \text{sources } [t \circ \Lambda.\text{Src } u]$   
**using** *sources-append* [*of*  $([\Lambda.\text{Src } t \circ u] \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U)$   
 $\text{sources-single-Src } [\text{of } \Lambda.\text{Src } t \circ u]$   
 $\text{sources-single-Src } [\text{of } t \circ \Lambda.\text{Src } u]$ ]  
**using** *arr-char t*  
**by** (*simp add: seq-char*)  
**thus** *?thesis*  
**using** 3 *coinitial-iff'* **by** *blast*  
**qed**  
**qed**  
**show** *?thesis*  
**proof**  
**show** 4:  $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) (u \# U) \text{ }^*\backslash^* [t \circ \Lambda.\text{Src } (\text{hd } (u \# U))] =$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } t)) (u \# U)$   
**proof** –  
**have**  $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) (u \# U) \text{ }^*\backslash^* [t \circ \Lambda.\text{Src } (\text{hd } (u \# U))] =$   
 $([\Lambda.\text{Src } t \circ u] @ \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U) \text{ }^*\backslash^* [t \circ \Lambda.\text{Src } u]$   
**by** *simp*  
**also have**  $\dots = [\Lambda.\text{Src } t \circ u] \text{ }^*\backslash^* [t \circ \Lambda.\text{Src } u] @$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U \text{ }^*\backslash^* ([t \circ \Lambda.\text{Src } u] \text{ }^*\backslash^* [\Lambda.\text{Src } t \circ u])$

```

    by (meson 2 Resid-append(1) con-char confluence not-Cons-self2)
  also have ... = [Λ.Trig t ∘ u] @ map (Λ.App (Λ.Src t)) U * \ * [t ∘ Λ.Trig u]
    using t Λ.Arr-not-Nil
    by (metis Arr-imp-arr-hd Λ.arr-char list.sel(1) orthogonal-App-single-single(1)
        orthogonal-App-single-single(2) uU)
  also have ... = [Λ.Trig t ∘ u] @ map (Λ.App (Λ.Trig t)) U
  proof -
    have Λ.Src (hd U) = Λ.Trig u
      using U uU Arr.elims(2) Srcs-simpΛP by force
    thus ?thesis
      using t uU ind Arr.elims(2) by fastforce
  qed
  also have ... = map (Λ.App (Λ.Trig t)) (u # U)
    by auto
  finally show ?thesis by blast
  qed
  show [t ∘ Λ.Src (hd (u # U))] * \ * map (Λ.App (Λ.Src t)) (u # U) =
    [t ∘ Λ.Trig (last (u # U))]
  proof -
    have [t ∘ Λ.Src (hd (u # U))] * \ * map (Λ.App (Λ.Src t)) (u # U) =
      ([t ∘ Λ.Src (hd (u # U))] * \ * [Λ.Src t ∘ u]) * \ * map (Λ.App (Λ.Src t)) U
    by (metis U 4 Con-sym Resid-cons(2) list.distinct(1) list.simps(9) map-is-Nil-conv)
    also have ... = [t ∘ Λ.Trig u] * \ * map (Λ.App (Λ.Src t)) U
    by (metis Arr-imp-arr-hd lambda-calculus.arr-char list.sel(1)
        orthogonal-App-single-single(2) t uU)
    also have ... = [t ∘ Λ.Trig (last (u # U))]
    by (metis 2 t U uU Con-Arr-self Con-cons(1) Con-implies-Arr(1) Trig-last-Src-hd-eq1
        arr-append-imp-seq coinitalE ind Λ.Src.simps(4) Λ.Trig.simps(3)
        Λ.lambda.inject(3) last.simps list.distinct(1) list.map-sel(1) map-is-Nil-conv)
    finally show ?thesis by blast
  qed
  qed
  qed
  qed

```

**lemma** *orthogonal-App-Arr-Arr*:

**shows**  $\llbracket \text{Arr } T; \text{Arr } U \rrbracket \implies$

$$\begin{aligned}
 & \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } T))) U * \ \ * \ \text{map } (\lambda X. \Lambda.\text{App } X (\Lambda.\text{Src } (\text{hd } U))) T = \\
 & \text{map } (\Lambda.\text{App } (\Lambda.\text{Trig } (\text{last } T))) U \wedge \\
 & \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) T * \ \ * \ \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } T))) U = \\
 & \text{map } (\lambda X. X \circ \Lambda.\text{Trig } (\text{last } U)) T
 \end{aligned}$$

**proof** (*induct T arbitrary: U*)

**show**  $\wedge U. \llbracket \text{Arr } \square; \text{Arr } U \rrbracket$

$$\begin{aligned}
 & \implies \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } \square))) U * \ \ * \ \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) \square = \\
 & \text{map } (\Lambda.\text{App } (\Lambda.\text{Trig } (\text{last } \square))) U \wedge \\
 & \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) \square * \ \ * \ \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } \square))) U = \\
 & \text{map } (\lambda X. X \circ \Lambda.\text{Trig } (\text{last } U)) \square
 \end{aligned}$$

by *simp*

**fix**  $t T U$

```

assume ind:  $\bigwedge U. \llbracket \text{Arr } T; \text{Arr } U \rrbracket$ 
   $\implies \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } T))) U \text{ }^*\backslash^*$ 
     $\text{map } (\lambda X. \Lambda.\text{App } X (\Lambda.\text{Src } (\text{hd } U))) T =$ 
     $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\text{last } T))) U \wedge$ 
     $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) T \text{ }^*\backslash^* \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } T))) U =$ 
     $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U)) T$ 
assume tT:  $\text{Arr } (t \# T)$ 
assume U:  $\text{Arr } U$ 
show  $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } (t \# T)))) U \text{ }^*\backslash^* \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) (t \# T) =$ 
   $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\text{last } (t \# T)))) U \wedge$ 
   $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) (t \# T) \text{ }^*\backslash^* \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } (t \# T)))) U =$ 
   $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U)) (t \# T)$ 
proof (cases  $T = []$ )
  show  $T = [] \implies ?thesis$ 
    using tT U
    by (simp add: orthogonal-App-single-Arr)
  assume  $T: T \neq []$ 
  have 1:  $\text{Arr } T$ 
    using T tT Arr-imp-Arr-tl by fastforce
  have 2:  $\Lambda.\text{Src } (\text{hd } T) = \Lambda.\text{Trg } t$ 
    using tT T Arr.elims(2) Srcs-simp $\Lambda_P$  by force
  show ?thesis
proof
  show 3:  $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } (t \# T)))) U \text{ }^*\backslash^*$ 
     $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) (t \# T) =$ 
     $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\text{last } (t \# T)))) U$ 
proof –
  have  $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } (t \# T)))) U \text{ }^*\backslash^* \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) (t \# T)$ 
  =
     $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U \text{ }^*\backslash^*$ 
     $([\Lambda.\text{App } t (\Lambda.\text{Src } (\text{hd } U))] @ \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) T)$ 
    using tT U by simp
  also have  $\dots = (\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) U \text{ }^*\backslash^* [t \circ \Lambda.\text{Src } (\text{hd } U)]) \text{ }^*\backslash^*$ 
     $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) T$ 
    using tT U Resid-append(2)
    by (metis Con-appendI(2) Resid.simps(1) T map-is-Nil-conv not-Cons-self2)
  also have  $\dots = \text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } t)) U \text{ }^*\backslash^* \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) T$ 
    using tT U orthogonal-App-single-Arr Arr-imp-arr-hd by fastforce
  also have  $\dots = \text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\text{last } (t \# T)))) U$ 
    using tT U 1 2 ind by auto
  finally show ?thesis by blast
qed
show  $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) (t \# T) \text{ }^*\backslash^*$ 
   $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } (t \# T)))) U =$ 
   $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U)) (t \# T)$ 
proof –
  have  $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) (t \# T) \text{ }^*\backslash^*$ 
     $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } (t \# T)))) U =$ 
     $([t \circ \Lambda.\text{Src } (\text{hd } U)] @ \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U)) T) \text{ }^*\backslash^*$ 

```

```

      map (Λ.App (Λ.Src t)) U
    using tT U by simp
  also have ... = ([t ∘ Λ.Src (hd U)] * \ * map (Λ.App (Λ.Src t)) U) @
    (map (λX. X ∘ Λ.Src (hd U)) T * \ *
      (map (Λ.App (Λ.Src t)) U * \ * [t ∘ Λ.Src (hd U)]))
    using tT U 3 Con-sym
      Resid-append(1)
      [of [t ∘ Λ.Src (hd U)] map (λX. X ∘ Λ.Src (hd U)) T
        map (Λ.App (Λ.Src t)) U]
    by fastforce
  also have ... = [t ∘ Λ.Trq (last U)] @
    map (λX. X ∘ Λ.Src (hd U)) T * \ * map (Λ.App (Λ.Trq t)) U
    using tT U Arr-imp-arr-hd orthogonal-App-single-Arr by fastforce
  also have ... = [t ∘ Λ.Trq (last U)] @ map (λX. X ∘ Λ.Trq (last U)) T
    using tT U 1 2 ind by presburger
  also have ... = map (λX. X ∘ Λ.Trq (last U)) (t # T)
    by simp
  finally show ?thesis by blast
qed
qed
qed
qed

```

**lemma** *orthogonal-App-cong*:

**assumes** *Arr T* and *Arr U*

**shows**  $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (hd U)) T @ \text{map } (\Lambda.\text{App } (\Lambda.\text{Trq } (last T))) U \sim^*$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (hd T))) U @ \text{map } (\lambda X. X \circ \Lambda.\text{Trq } (last U)) T$

**proof**

```

  have 1: Arr (map (λX. X ∘ Λ.Src (hd U)) T)
    using assms Arr-imp-arr-hd Arr-map-App1 Λ.Ide-Src by force
  have 2: Arr (map (Λ.App (Λ.Trq (last T))) U)
    using assms Arr-imp-arr-last Arr-map-App2 Λ.Ide-Trq by force
  have 3: Arr (map (Λ.App (Λ.Src (hd T))) U)
    using assms Arr-imp-arr-hd Arr-map-App2 Λ.Ide-Src by force
  have 4: Arr (map (λX. X ∘ Λ.Trq (last U)) T)
    using assms Arr-imp-arr-last Arr-map-App1 Λ.Ide-Trq by force
  have 5: Arr (map (λX. X ∘ Λ.Src (hd U)) T @ map (Λ.App (Λ.Trq (last T))) U)
    using assms
    by (metis (no-types, lifting) 1 2 Arr.simps(2) Arr-has-Src Arr-imp-arr-last
      Srcs.simps(1) Srcs-Resid-Arr-single Trqs-simpP arr-append arr-char last-map
      orthogonal-App-single-Arr seq-char)
  have 6: Arr (map (Λ.App (Λ.Src (hd T))) U @ map (λX. X ∘ Λ.Trq (last U)) T)
    using assms
    by (metis (no-types, lifting) 3 4 Arr.simps(2) Arr-has-Src Arr-imp-arr-hd
      Srcs.simps(1) Srcs.simps(2) Srcs-Resid Srcs-simpP arr-append arr-char hd-map
      orthogonal-App-single-Arr seq-char)
  have 7: Con (map (λX. X ∘ Λ.Src (hd U)) T @ map ((∘) (Λ.Trq (last T))) U)
    (map ((∘) (Λ.Src (hd T))) U @ map (λX. X ∘ Λ.Trq (last U)) T)

```

**using** *assms orthogonal-App-Arr-Arr* [of  $T\ U$ ]  
**by** (*metis 1 2 5 6 Con-imp-eq-Srcs Resid.simps(1) Srcs-append confluence-ind*)  
**have** 8: *Con* ( $\text{map } ((\circ) (\Lambda.\text{Src } (\text{hd } T)))\ U\ @\ \text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T$ )  
 $(\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U))\ T\ @\ \text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U)$   
**using** 7 *Con-sym* **by** *simp*  
**show**  $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U))\ T\ @\ \text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U\ \overset{*}{\lesssim}\$   
 $\text{map } ((\circ) (\Lambda.\text{Src } (\text{hd } T)))\ U\ @\ \text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T$   
**proof** –  
**have**  $(\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U))\ T\ @\ \text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U)\ \overset{*}{\setminus}\$   
 $(\text{map } ((\circ) (\Lambda.\text{Src } (\text{hd } T)))\ U\ @\ \text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T) =$   
 $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T\ \overset{*}{\setminus}\ \text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T\ @$   
 $(\text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U)\ \overset{*}{\setminus}\ \text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U)\ \overset{*}{\setminus}$   
 $(\text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T)\ \overset{*}{\setminus}\ \text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T)$   
**using** *assms 7 orthogonal-App-Arr-Arr*  
*Resid-append2*  
 $[\text{of } \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U))\ T\ \text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\text{last } T)))\ U$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } T)))\ U\ \text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T]$   
**by** *fastforce*  
**moreover** **have** *Ide ...*  
**using** *assms 1 2 3 4 5 6 7 Resid-Arr-self*  
**by** (*metis Arr-append-iff<sub>P</sub> Con-Arr-self Con-imp-Arr-Resid Ide-appendI<sub>P</sub>*  
*Resid-Ide-Arr-ind append-Nil2 calculation*)  
**ultimately show** *?thesis*  
**using** *ide-char* **by** *presburger*  
**qed**  
**show**  $\text{map } ((\circ) (\Lambda.\text{Src } (\text{hd } T)))\ U\ @\ \text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T\ \overset{*}{\lesssim}\$   
 $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U))\ T\ @\ \text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U$   
**proof** –  
**have**  $\text{map } ((\circ) (\Lambda.\text{Src } (\text{hd } T)))\ U\ \overset{*}{\setminus}\ \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U))\ T =$   
 $\text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U$   
**by** (*simp add: assms orthogonal-App-Arr-Arr*)  
**have**  $(\text{map } ((\circ) (\Lambda.\text{Src } (\text{hd } T)))\ U\ @\ \text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T)\ \overset{*}{\setminus}\$   
 $(\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U))\ T\ @\ \text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U) =$   
 $(\text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U)\ \overset{*}{\setminus}\ \text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U\ @$   
 $(\text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T)\ \overset{*}{\setminus}\ \text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T)\ \overset{*}{\setminus}$   
 $(\text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U)\ \overset{*}{\setminus}\ \text{map } ((\circ) (\Lambda.\text{Trg } (\text{last } T)))\ U)$   
**using** *assms 8 orthogonal-App-Arr-Arr* [of  $T\ U$ ]  
*Resid-append2*  
 $[\text{of } \text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } T)))\ U\ \text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } U))\ T$   
 $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } U))\ T\ \text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\text{last } T)))\ U]$   
**by** *fastforce*  
**moreover** **have** *Ide ...*  
**using** *assms 1 2 3 4 5 6 8 Resid-Arr-self Arr-append-iff<sub>P</sub> Con-sym*  
**by** (*metis Con-Arr-self Con-imp-Arr-Resid Ide-appendI<sub>P</sub> Resid-Ide-Arr-ind*  
*append-Nil2 calculation*)  
**ultimately show** *?thesis*  
**using** *ide-char* **by** *presburger*  
**qed**  
**qed**

We arrive at the final objective of this section: factorization, up to congruence, of a path whose transitions all have *App* as the top-level constructor, into the composite of a path that reduces only the “rators” and a path that reduces only the “rands”.

**lemma** *map-App-decomp*:

**shows**  $\llbracket \text{Arr } U; \text{ set } U \subseteq \text{Collect } \Lambda.\text{is-App} \rrbracket \implies$

$$\begin{aligned} & \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\Lambda.\text{un-App2 } (\text{hd } U))) (\text{map } \Lambda.\text{un-App1 } U) @ \\ & \text{map } (\lambda X. \Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } U)) \circ X) (\text{map } \Lambda.\text{un-App2 } U) * \sim^* \\ & U \end{aligned}$$

**proof** (*induct*  $U$ )

**show**  $\text{Arr } [] \implies \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\Lambda.\text{un-App2 } (\text{hd } []))) (\text{map } \Lambda.\text{un-App1 } []) @$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } [])))) (\text{map } \Lambda.\text{un-App2 } []) * \sim^*$   
 $[]$

**by** *simp*

**fix**  $u U$

**assume** *ind*:  $\llbracket \text{Arr } U; \text{ set } U \subseteq \text{Collect } \Lambda.\text{is-App} \rrbracket \implies$

$$\begin{aligned} & \text{map } (\lambda X. \Lambda.\text{App } X (\Lambda.\text{Src } (\Lambda.\text{un-App2 } (\text{hd } U)))) (\text{map } \Lambda.\text{un-App1 } U) @ \\ & \text{map } (\lambda X. \Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } U)) \circ X) (\text{map } \Lambda.\text{un-App2 } U) * \sim^* \\ & U \end{aligned}$$

**assume**  $uU: \text{Arr } (u \# U)$

**assume** *set*:  $\text{set } (u \# U) \subseteq \text{Collect } \Lambda.\text{is-App}$

**have**  $u: \Lambda.\text{Arr } u \wedge \Lambda.\text{is-App } u$

**using** *set set-Arr-subset-arr uU by fastforce*

**show**  $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\Lambda.\text{un-App2 } (\text{hd } (u \# U)))) (\text{map } \Lambda.\text{un-App1 } (u \# U)) @$   
 $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } (u \# U)))) (\text{map } \Lambda.\text{un-App2 } (u \# U)) * \sim^*$   
 $u \# U$

**proof** (*cases*  $U = []$ )

**assume**  $U: U = []$

**show** *?thesis*

**using**  $u U \Lambda.\text{Con-sym } \Lambda.\text{Ide-iff-Src-self } \Lambda.\text{resid-Arr-self } \Lambda.\text{resid-Src-Arr}$   
 $\Lambda.\text{resid-Arr-Src } \Lambda.\text{Src-resid } \Lambda.\text{Arr-resid ide-char } \Lambda.\text{Arr-not-Nil}$

**by** (*cases*  $u$ , *simp-all*)

**next**

**assume**  $U: U \neq []$

**have**  $1: \text{Arr } (\text{map } \Lambda.\text{un-App1 } U)$

**using**  $U \text{ set Arr-map-un-App1 } uU$

**by** (*metis Arr-imp-Arr-tl list.distinct(1) list.map-disc-iff list.map-sel(2) list.sel(3)*)

**have**  $2: \text{Arr } [\Lambda.\text{un-App2 } u]$

**using**  $U uU \text{ set}$

**by** (*metis Arr.simps(2) Arr-imp-arr-hd Arr-map-un-App2 hd-map list.discI list.sel(1)*)

**have**  $3: \Lambda.\text{Arr } (\Lambda.\text{un-App1 } u) \wedge \Lambda.\text{Arr } (\Lambda.\text{un-App2 } u)$

**using**  $uU \text{ set}$

**by** (*metis Arr-imp-arr-hd Arr-map-un-App1 Arr-map-un-App2 \Lambda.arr-char*  
 $\text{list.distinct(1) list.map-sel(1) list.sel(1)$ )

**have**  $4: \text{map } (\lambda X. X \circ \Lambda.\text{Src } (\Lambda.\text{un-App2 } u)) (\text{map } \Lambda.\text{un-App1 } U) @$

$[\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } U)) \circ \Lambda.\text{un-App2 } u] * \sim^*$

$[\Lambda.\text{Src } (\text{hd } (\text{map } \Lambda.\text{un-App1 } U)) \circ \Lambda.\text{un-App2 } u] @$

$\text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } [\Lambda.\text{un-App2 } u])) (\text{map } \Lambda.\text{un-App1 } U)$

**proof** –

**have**  $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } [\Lambda.\text{un-App2 } u])) (\text{map } \Lambda.\text{un-App1 } U) =$

$\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\Lambda.\text{un-App2 } u)) (\text{map } \Lambda.\text{un-App1 } U)$   
**using**  $U \text{ } uU \text{ set by simp}$   
**moreover have**  $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\text{last } (\text{map } \Lambda.\text{un-App1 } U)))) [\Lambda.\text{un-App2 } u] =$   
 $[\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } U)) \circ \Lambda.\text{un-App2 } u]$   
**by** (*simp add: U last-map*)  
**moreover have**  $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } (\text{map } \Lambda.\text{un-App1 } U)))) [\Lambda.\text{un-App2 } u] =$   
 $[\Lambda.\text{Src } (\text{hd } (\text{map } \Lambda.\text{un-App1 } U)) \circ \Lambda.\text{un-App2 } u]$   
**by simp**  
**moreover have**  $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } [\Lambda.\text{un-App2 } u])) (\text{map } \Lambda.\text{un-App1 } U) =$   
 $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } [\Lambda.\text{un-App2 } u])) (\text{map } \Lambda.\text{un-App1 } U)$   
**using**  $U \text{ } uU \text{ set by blast}$   
**ultimately show** *?thesis*  
**using**  $U \text{ } uU \text{ set last-map hd-map 1 2 3}$   
 $\text{orthogonal-App-cong [of map } \Lambda.\text{un-App1 } U [\Lambda.\text{un-App2 } u]]$   
**by presburger**  
**qed**  
**have** 5:  $\Lambda.\text{Arr } (\Lambda.\text{un-App1 } u \circ \Lambda.\text{Src } (\Lambda.\text{un-App2 } u))$   
**by** (*simp add: 3*)  
**have** 6:  $\text{Arr } (\text{map } (\lambda X. \Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } U)) \circ X) (\text{map } \Lambda.\text{un-App2 } U))$   
**by** (*metis 1 Arr-imp-arr-last Arr-map-App2 Arr-map-un-App2 Con-implies-Arr(2)*  
 $\text{Ide.simps(1) Resid-Arr-self Resid-cons(2) U insert-subset}$   
 $\Lambda.\text{Ide-Trg } \Lambda.\text{arr-char last-map list.simps(15) set } uU$ )  
**have** 7:  $\Lambda.\text{Arr } (\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } U)))$   
**by** (*metis 4 Arr.simps(2) Arr-append-iff<sub>P</sub> Con-implies-Arr(2) Ide.simps(1)*  
 $U \text{ ide-char } \Lambda.\text{Arr.simps(4) } \Lambda.\text{arr-char list.map-disc-iff not-Cons-self2}$ )  
**have** 8:  $\Lambda.\text{Src } (\text{hd } (\text{map } \Lambda.\text{un-App1 } U)) = \Lambda.\text{Trg } (\Lambda.\text{un-App1 } u)$   
**proof** –  
**have**  $\Lambda.\text{Src } (\text{hd } U) = \Lambda.\text{Trg } u$   
**using**  $u \text{ } uU \text{ } U$  **by fastforce**  
**thus** *?thesis*  
**using**  $u \text{ } uU \text{ } U \text{ set}$   
**apply** (*cases u; cases hd U*)  
**apply** (*simp-all add: list.map-sel(1)*)  
**using** *list.set-sel(1)*  
**by fastforce**  
**qed**  
**have** 9:  $\Lambda.\text{Src } (\Lambda.\text{un-App2 } (\text{hd } U)) = \Lambda.\text{Trg } (\Lambda.\text{un-App2 } u)$   
**proof** –  
**have**  $\Lambda.\text{Src } (\text{hd } U) = \Lambda.\text{Trg } u$   
**using**  $u \text{ } uU \text{ } U$  **by fastforce**  
**thus** *?thesis*  
**using**  $u \text{ } uU \text{ } U \text{ set}$   
**apply** (*cases u; cases hd U*)  
**apply** *simp-all*  
**by** (*metis lambda-calculus.lambda.disc(15) list.set-sel(1) mem-Collect-eq*  
 $\text{subset-code(1)}$ )  
**qed**  
**have**  $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\Lambda.\text{un-App2 } (\text{hd } (u \# U)))) (\text{map } \Lambda.\text{un-App1 } (u \# U)) @$   
 $\text{map } ((\circ) (\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } (u \# U)))) (\text{map } \Lambda.\text{un-App2 } (u \# U))) =$

```

    [Λ.un-App1 u ◦ Λ.Src (Λ.un-App2 u)] @
      (map (λX. X ◦ Λ.Src (Λ.un-App2 u))
        (map Λ.un-App1 U) @ [Λ.Trig (Λ.un-App1 (last U)) ◦ Λ.un-App2 u]) @
        map ((◦) (Λ.Trig (Λ.un-App1 (last U)))) (map Λ.un-App2 U)
  using uU U by simp
also have 12: cong ... ([Λ.un-App1 u ◦ Λ.Src (Λ.un-App2 u)] @
  ([Λ.Src (hd (map Λ.un-App1 U)) ◦ Λ.un-App2 u] @
    map (λX. X ◦ Λ.Trig (last [Λ.un-App2 u])) (map Λ.un-App1 U)) @
    map ((◦) (Λ.Trig (Λ.un-App1 (last U)))) (map Λ.un-App2 U))
proof (intro cong-append [of [Λ.un-App1 u ◦ Λ.Src (Λ.un-App2 u)]]
  cong-append [where U = map (λX. Λ.Trig (Λ.un-App1 (last U)) ◦ X)
    (map Λ.un-App2 U)])
  show [Λ.un-App1 u ◦ Λ.Src (Λ.un-App2 u)] *~* [Λ.un-App1 u ◦ Λ.Src (Λ.un-App2 u)]
    using 5 arr-char cong-reflexive Arr.simps(2) Λ.arr-char by presburger
  show map (λX. Λ.Trig (Λ.un-App1 (last U)) ◦ X) (map Λ.un-App2 U) *~*
    map (λX. Λ.Trig (Λ.un-App1 (last U)) ◦ X) (map Λ.un-App2 U)
    using 6 cong-reflexive by auto
  show map (λX. X ◦ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
    [Λ.Trig (Λ.un-App1 (last U)) ◦ Λ.un-App2 u] *~*
    [Λ.Src (hd (map Λ.un-App1 U)) ◦ Λ.un-App2 u] @
    map (λX. X ◦ Λ.Trig (last [Λ.un-App2 u])) (map Λ.un-App1 U)
    using 4 by simp
  show 10: seq [Λ.un-App1 u ◦ Λ.Src (Λ.un-App2 u)]
    ((map (λX. X ◦ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
      [Λ.Trig (Λ.un-App1 (last U)) ◦ Λ.un-App2 u]) @
      map (λX. Λ.Trig (Λ.un-App1 (last U)) ◦ X) (map Λ.un-App2 U))
proof
  show Arr [Λ.un-App1 u ◦ Λ.Src (Λ.un-App2 u)]
    using 5 Arr.simps(2) by blast
  show Arr ((map (λX. X ◦ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
    [Λ.Trig (Λ.un-App1 (last U)) ◦ Λ.un-App2 u]) @
    map (λX. Λ.Trig (Λ.un-App1 (last U)) ◦ X) (map Λ.un-App2 U))
proof (intro Arr-appendIPWE)
  show Arr (map (λX. X ◦ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U))
    using 1 3 Arr-map-App1 lambda-calculus.Ide-Src by blast
  show Arr [Λ.Trig (Λ.un-App1 (last U)) ◦ Λ.un-App2 u]
    by (simp add: 3 7)
  show Trig (map (λX. X ◦ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U)) =
    Src [Λ.Trig (Λ.un-App1 (last U)) ◦ Λ.un-App2 u]
    by (metis 4 Arr-appendEPWE Con-implies-Arr(2) Ide.simps(1) U ide-char
      list.map-disc-iff not-Cons-self2)
  show Arr (map (λX. Λ.Trig (Λ.un-App1 (last U)) ◦ X) (map Λ.un-App2 U))
    using 6 by simp
  show Trig (map (λX. X ◦ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
    [Λ.Trig (Λ.un-App1 (last U)) ◦ Λ.un-App2 u]) =
    Src (map (λX. Λ.Trig (Λ.un-App1 (last U)) ◦ X) (map Λ.un-App2 U))
    using U uU set 1 3 6 7 9 Srcs-simpPWE Arr-imp-arr-hd Arr-imp-arr-last
  apply auto
  by (metis Nil-is-map-conv hd-map Λ.Src.simps(4) Λ.Src-Trig Λ.Trig-Trig

```

```

      last-map list.map-comp)
qed
show  $\Lambda.Trg (last [\Lambda.un-App1 u \circ \Lambda.Src (\Lambda.un-App2 u)]) =$ 
 $\Lambda.Src (hd ((map (\lambda X. X \circ \Lambda.Src (\Lambda.un-App2 u)) (map \Lambda.un-App1 U) @$ 
 $[\Lambda.Trg (\Lambda.un-App1 (last U)) \circ \Lambda.un-App2 u] @$ 
 $map (\lambda X. \Lambda.Trg (\Lambda.un-App1 (last U)) \circ X) (map \Lambda.un-App2 U))))$ 
  using 8 9
  by (simp add: 3 U hd-map)
qed
show seq (map ( $\lambda X. X \circ \Lambda.Src (\Lambda.un-App2 u)$ ) (map  $\Lambda.un-App1 U$ ) @
 $[\Lambda.Trg (\Lambda.un-App1 (last U)) \circ \Lambda.un-App2 u]$ )
  (map ( $\lambda X. \Lambda.Trg (\Lambda.un-App1 (last U)) \circ X$ ) (map  $\Lambda.un-App2 U$ ))
  by (metis Nil-is-map-conv U 10 append-is-Nil-conv arr-append-imp-seq seqE)
qed
also have 11:  $[\Lambda.un-App1 u \circ \Lambda.Src (\Lambda.un-App2 u)] @$ 
 $[\Lambda.Src (hd (map \Lambda.un-App1 U)) \circ \Lambda.un-App2 u] @$ 
 $map (\lambda X. X \circ \Lambda.Trg (last [\Lambda.un-App2 u])) (map \Lambda.un-App1 U) @$ 
 $map ((\circ) (\Lambda.Trg (\Lambda.un-App1 (last U)))) (map \Lambda.un-App2 U) =$ 
 $[\Lambda.un-App1 u \circ \Lambda.Src (\Lambda.un-App2 u)] @$ 
 $[\Lambda.Src (hd (map \Lambda.un-App1 U)) \circ \Lambda.un-App2 u] @$ 
 $map (\lambda X. X \circ \Lambda.Trg (last [\Lambda.un-App2 u])) (map \Lambda.un-App1 U) @$ 
 $map ((\circ) (\Lambda.Trg (\Lambda.un-App1 (last U)))) (map \Lambda.un-App2 U)$ 
  by simp
also have cong ... ([u] @ U)
proof (intro cong-append)
  show seq ( $[\Lambda.un-App1 u \circ \Lambda.Src (\Lambda.un-App2 u)] @$ 
 $[\Lambda.Src (hd (map \Lambda.un-App1 U)) \circ \Lambda.un-App2 u]$ )
    (map ( $\lambda X. X \circ \Lambda.Trg (last [\Lambda.un-App2 u])) (map \Lambda.un-App1 U) @$ 
 $map ((\circ) (\Lambda.Trg (\Lambda.un-App1 (last U)))) (map \Lambda.un-App2 U)$ )
  by (metis 5 11 12 U Arr.simps(1-2) Con-implies-Arr(2) Ide.simps(1) Nil-is-map-conv
    append-is-Nil-conv arr-append-imp-seq arr-char ide-char  $\Lambda.arr-char$ )
  show  $[\Lambda.un-App1 u \circ \Lambda.Src (\Lambda.un-App2 u)] @$ 
 $[\Lambda.Src (hd (map \Lambda.un-App1 U)) \circ \Lambda.un-App2 u] * \sim^*$ 
 $[u]$ 
  proof -
    have  $[\Lambda.un-App1 u \circ \Lambda.Src (\Lambda.un-App2 u)] @$ 
 $[\Lambda.Trg (\Lambda.un-App1 u) \circ \Lambda.un-App2 u] * \sim^*$ 
 $[u]$ 
      using u uU U  $\Lambda.Arr-Trg$   $\Lambda.Arr-not-Nil$   $\Lambda.resid-Arr-self$ 
      apply (cases u)
      apply auto
      by force+
    thus ?thesis using 8 by simp
  qed
show map ( $\lambda X. X \circ \Lambda.Trg (last [\Lambda.un-App2 u])$ ) (map  $\Lambda.un-App1 U$ ) @
 $map ((\circ) (\Lambda.Trg (\Lambda.un-App1 (last U)))) (map \Lambda.un-App2 U) * \sim^*$ 
  U
  using ind set 9
  apply simp

```

using  $U \ uU$  by *blast*  
 qed  
 also have  $[u] @ U = u \# U$   
 by *simp*  
 finally show *?thesis* by *blast*  
 qed  
 qed

### 3.3.4 Miscellaneous

**lemma** *Resid-parallel*:  
**assumes** *cong t t'* and *coinitial t u*  
**shows**  $u \text{ ** } t = u \text{ ** } t'$   
**proof** –  
 have  $u \text{ ** } t = (u \text{ ** } t) \text{ ** } (t' \text{ ** } t)$   
 using *assms*  
 by (*metis con-target conIP con-sym resid-arr-ide*)  
 also have  $\dots = (u \text{ ** } t') \text{ ** } (t \text{ ** } t')$   
 using *cube* by *auto*  
 also have  $\dots = u \text{ ** } t'$   
 using *assms*  
 by (*metis con-target conIP con-sym resid-arr-ide*)  
 finally show *?thesis* by *blast*  
 qed

**lemma** *set-Ide-subset-single-hd*:  
**shows**  $\text{Ide } T \implies \text{set } T \subseteq \{\text{hd } T\}$   
**apply** (*induct T, auto*)  
**using**  $\Lambda.\text{coinitial-ide-are-cong}$   
**by** (*metis Arr-imp-arr-hd Ide-consE Ide-imp-Ide-hd Ide-implies-Arr Srcs-simpPWE Srcs-simp $\Lambda$ P*  
 $\Lambda.\text{trg-ide equals0D } \Lambda.\text{Ide-iff-Src-self } \Lambda.\text{arr-char } \Lambda.\text{ide-char set-empty singletonD}$   
 $\text{subset-code}(1)$ )

A single parallel reduction with *Beta* as the top-level operator factors, up to congruence, either as a path in which the top-level redex is contracted first, or as a path in which the top-level redex is contracted last.

**lemma** *Beta-decomp*:  
**assumes**  $\Lambda.\text{Arr } t$  and  $\Lambda.\text{Arr } u$   
**shows**  $[\lambda[\Lambda.\text{Src } t] \bullet \Lambda.\text{Src } u] @ [\Lambda.\text{subst } u t] \text{ ** } [\lambda[t] \bullet u]$   
**and**  $[\lambda[t] \circ u] @ [\lambda[\Lambda.\text{Trg } t] \bullet \Lambda.\text{Trg } u] \text{ ** } [\lambda[t] \bullet u]$   
**using** *assms*  $\Lambda.\text{Arr-not-Nil } \Lambda.\text{Subst-not-Nil ide-char } \Lambda.\text{Ide-Subst } \Lambda.\text{Ide-Trg}$   
 $\Lambda.\text{Arr-Subst } \Lambda.\text{resid-Arr-self}$   
**by** *auto*

If a reduction path follows an initial reduction whose top-level constructor is *Lam*, then all the terms in the path have *Lam* as their top-level constructor.

**lemma** *seq-Lam-Arr-implies*:  
**shows**  $\llbracket \text{seq } [t] \ U; \Lambda.\text{is-Lam } t \rrbracket \implies \text{set } U \subseteq \text{Collect } \Lambda.\text{is-Lam}$   
**proof** (*induct U arbitrary: t*)

```

show  $\bigwedge t. \llbracket \text{seq } [t] \rrbracket; \Lambda.\text{is-Lam } t \rrbracket \implies \text{set } [] \subseteq \text{Collect } \Lambda.\text{is-Lam}$ 
  by simp
fix  $u \ U \ t$ 
assume  $\text{ind}: \bigwedge t. \llbracket \text{seq } [t] \ U; \Lambda.\text{is-Lam } t \rrbracket \implies \text{set } U \subseteq \text{Collect } \Lambda.\text{is-Lam}$ 
assume  $uU: \text{seq } [t] \ (u \# \ U)$ 
assume  $t: \Lambda.\text{is-Lam } t$ 
show  $\text{set } (u \# \ U) \subseteq \text{Collect } \Lambda.\text{is-Lam}$ 
proof –
  have  $\Lambda.\text{is-Lam } u$ 
    by (metis Trg-last-Src-hd-eqI  $\Lambda.\text{Src.simps}(1-2,4-5)$   $\Lambda.\text{Trg.simps}(2)$   $\Lambda.\text{is-App-def}$ 
       $\Lambda.\text{is-Beta-def}$   $\Lambda.\text{is-Lam-def}$   $\Lambda.\text{is-Var-def}$   $\Lambda.\text{lambda.disc}(9)$   $\Lambda.\text{lambda.exhaust-disc}$ 
      last-ConsL list.sel(1)  $t \ uU$ )
  moreover have  $\text{set } U \subseteq \text{Collect } \Lambda.\text{is-Lam}$ 
  proof (cases  $U = []$ )
    show  $U = [] \implies ?thesis$ 
      by simp
    assume  $U: U \neq []$ 
    have  $\text{seq } [u] \ U$ 
      by (metis U append-Cons arr-append-imp-seq not-Cons-self2 self-append-conv2
        seqE  $uU$ )
    thus ?thesis
      using ind calculation by simp
  qed
  ultimately show ?thesis by auto
qed
qed
qed

lemma seq-map-un-Lam:
assumes  $\text{seq } [\lambda[t]] \ U$ 
shows  $\text{seq } [t] \ (\text{map } \Lambda.\text{un-Lam } U)$ 
proof –
  have  $\text{Arr } (\lambda[t] \# \ U)$ 
    using assms
    by (simp add: seq-char)
  hence  $\text{Arr } (\text{map } \Lambda.\text{un-Lam } (\lambda[t] \# \ U)) \wedge \text{Arr } U$ 
    using seq-Lam-Arr-implies
    by (metis Arr-map-un-Lam  $\langle \text{seq } [\lambda[t]] \ U \rangle \Lambda.\text{lambda.discI}(2)$  mem-Collect-eq
      seq-char set-ConsD subset-code(1))
  hence  $\text{Arr } (\Lambda.\text{un-Lam } \lambda[t] \# \ \text{map } \Lambda.\text{un-Lam } U) \wedge \text{Arr } U$ 
    by simp
  thus ?thesis
    using seq-char
    by (metis (no-types, lifting) Arr.simps(1) Con-imp-eq-Srcs Con-implies-Arr(2)
      Con-initial-right Resid-rec(1) Resid-rec(3) Srcs-Resid  $\Lambda.\text{lambda.sel}(2)$ 
      map-is-Nil-conv confluence-ind)
qed
end

```

### 3.4 Developments

A *development* is a reduction path from a term in which at each step exactly one redex is contracted, and the only redexes that are contracted are those that are residuals of redexes present in the original term. That is, no redexes are contracted that were newly created as a result of the previous reductions. The main theorem about developments is the Finite Developments Theorem, which states that all developments are finite. A proof of this theorem was published by Hindley [6], who attributes the result to Schroer [9]. Other proofs were published subsequently. Here we follow the paper by de Vrijer [5], which may in some sense be considered the definitive work because de Vrijer's proof gives an exact bound on the number of steps in a development. Since de Vrijer used a classical, named-variable representation of  $\lambda$ -terms, for the formalization given in the present article it was necessary to find the correct way to adapt de Vrijer's proof to the de Bruijn index representation of terms. I found this to be a somewhat delicate matter and to my knowledge it has not been done previously.

**context** *lambda-calculus*  
**begin**

We define an *elementary reduction* defined to be a term with exactly one marked redex. These correspond to the most basic computational steps.

**fun** *elementary-reduction*  
**where** *elementary-reduction*  $\# \longleftrightarrow \text{False}$   
     | *elementary-reduction* ( $\llcorner - \lrcorner$ )  $\longleftrightarrow \text{False}$   
     | *elementary-reduction*  $\lambda[t] \longleftrightarrow \text{elementary-reduction } t$   
     | *elementary-reduction* ( $t \circ u$ )  $\longleftrightarrow$   
         (*elementary-reduction*  $t \wedge \text{Ide } u$ )  $\vee$  (*Ide*  $t \wedge \text{elementary-reduction } u$ )  
     | *elementary-reduction* ( $\lambda[t] \bullet u$ )  $\longleftrightarrow \text{Ide } t \wedge \text{Ide } u$

It is tempting to imagine that elementary reductions would be atoms with respect to the preorder  $\lesssim$ , but this is not necessarily the case. For example, suppose  $t = \lambda[\llcorner 1 \lrcorner] \bullet (\lambda[\llcorner 0 \lrcorner] \circ \llcorner 0 \lrcorner)$  and  $u = \lambda[\llcorner 1 \lrcorner] \bullet (\lambda[\llcorner 0 \lrcorner] \bullet \llcorner 0 \lrcorner)$ . Then  $t$  is an elementary reduction,  $u \lesssim t$  (in fact  $u \sim t$ ) but  $u$  is not an identity, nor is it elementary.

**lemma** *elementary-reduction-is-arr*:  
**shows** *elementary-reduction*  $t \implies \text{arr } t$   
     **using** *Ide-implies-Arr arr-char*  
     **by** (*induct*  $t$ ) *auto*

**lemma** *elementary-reduction-not-ide*:  
**shows** *elementary-reduction*  $t \implies \neg \text{ide } t$   
     **using** *ide-char*  
     **by** (*induct*  $t$ ) *auto*

**lemma** *elementary-reduction-Raise-iff*:  
**shows**  $\bigwedge d n. \text{elementary-reduction } (\text{Raise } d \ n \ t) \longleftrightarrow \text{elementary-reduction } t$   
     **using** *Ide-Raise*  
     **by** (*induct*  $t$ ) *auto*

**lemma** *elementary-reduction-Lam-iff*:  
**shows**  $is\text{-}Lam\ t \implies elementary\text{-}reduction\ t \longleftrightarrow elementary\text{-}reduction\ (un\text{-}Lam\ t)$   
**by** (*metis elementary-reduction.simps(3) lambda.collapse(2)*)

**lemma** *elementary-reduction-App-iff*:  
**shows**  $is\text{-}App\ t \implies elementary\text{-}reduction\ t \longleftrightarrow$   
 $(elementary\text{-}reduction\ (un\text{-}App1\ t) \wedge ide\ (un\text{-}App2\ t)) \vee$   
 $(ide\ (un\text{-}App1\ t) \wedge elementary\text{-}reduction\ (un\text{-}App2\ t))$   
**using** *ide-char*  
**by** (*metis elementary-reduction.simps(4) lambda.collapse(3)*)

**lemma** *elementary-reduction-Beta-iff*:  
**shows**  $is\text{-}Beta\ t \implies elementary\text{-}reduction\ t \longleftrightarrow ide\ (un\text{-}Beta1\ t) \wedge ide\ (un\text{-}Beta2\ t)$   
**using** *ide-char*  
**by** (*metis elementary-reduction.simps(5) lambda.collapse(4)*)

**lemma** *cong-elementary-reductions-are-equal*:  
**shows**  $\llbracket elementary\text{-}reduction\ t; elementary\text{-}reduction\ u; t \sim u \rrbracket \implies t = u$   
**proof** (*induct t arbitrary: u*)  
**show**  $\bigwedge u. \llbracket elementary\text{-}reduction\ \#\; elementary\text{-}reduction\ u; \# \sim u \rrbracket \implies \# = u$   
**by** *simp*  
**show**  $\bigwedge x\ u. \llbracket elementary\text{-}reduction\ \langle x \rangle; elementary\text{-}reduction\ u; \langle x \rangle \sim u \rrbracket \implies \langle x \rangle = u$   
**by** *simp*  
**show**  $\bigwedge t\ u. \llbracket \bigwedge u. \llbracket elementary\text{-}reduction\ t; elementary\text{-}reduction\ u; t \sim u \rrbracket \implies t = u;$   
 $elementary\text{-}reduction\ \lambda[t]; elementary\text{-}reduction\ u; \lambda[t] \sim u \rrbracket$   
 $\implies \lambda[t] = u$   
**by** (*metis elementary-reduction-Lam-iff lambda.collapse(2) lambda.inject(2) prfx-Lam-iff*)  
**show**  $\bigwedge t1\ t2. \llbracket \bigwedge u. \llbracket elementary\text{-}reduction\ t1; elementary\text{-}reduction\ u; t1 \sim u \rrbracket \implies t1 = u;$   
 $\bigwedge u. \llbracket elementary\text{-}reduction\ t2; elementary\text{-}reduction\ u; t2 \sim u \rrbracket \implies t2 = u;$   
 $elementary\text{-}reduction\ (t1 \circ t2); elementary\text{-}reduction\ u; t1 \circ t2 \sim u \rrbracket$   
 $\implies t1 \circ t2 = u$   
**for**  $u$   
**using** *prfx-App-iff*  
**apply** (*cases u*)  
**apply** *auto[3]*  
**apply** (*metis elementary-reduction-App-iff ide-backward-stable lambda.sel(3-4)*  
*weak-extensionality*)  
**by** *auto*  
**show**  $\bigwedge t1\ t2. \llbracket \bigwedge u. \llbracket elementary\text{-}reduction\ t1; elementary\text{-}reduction\ u; t1 \sim u \rrbracket \implies t1 = u;$   
 $\bigwedge u. \llbracket elementary\text{-}reduction\ t2; elementary\text{-}reduction\ u; t2 \sim u \rrbracket \implies t2 = u;$   
 $elementary\text{-}reduction\ (\lambda[t1] \bullet t2); elementary\text{-}reduction\ u; \lambda[t1] \bullet t2 \sim u \rrbracket$   
 $\implies \lambda[t1] \bullet t2 = u$   
**for**  $u$   
**using** *prfx-App-iff*  
**apply** (*cases u, simp-all*)  
**by** (*metis (full-types) Cinitial-iff-Con Ide-iff-Src-self Ide.simps(1)*)

**qed**

An *elementary reduction path* is a path in which each step is an elementary reduction. It will be convenient to regard the empty list as an elementary reduction path, even

though it is not actually a path according to our previous definition of that notion.

**definition** (in *reduction-paths*) *elementary-reduction-path*  
**where** *elementary-reduction-path*  $T \longleftrightarrow$   
 $(T = [] \vee \text{Arr } T \wedge \text{set } T \subseteq \text{Collect } \Lambda.\text{elementary-reduction})$

In the formal definition of “development” given below, we represent a set of redexes simply by a term, in which the occurrences of *Beta* correspond to the redexes in the set. To express the idea that an elementary reduction  $u$  is a member of the set of redexes represented by term  $t$ , it is not adequate to say  $u \lesssim t$ . To see this, consider the developments of a term of the form  $\lambda[t1] \bullet t2$ . Intuitively, such developments should consist of a (possibly empty) initial segment containing only transitions of the form  $t1 \circ t2$ , followed by a transition of the form  $\lambda[u1] \bullet u2'$ , followed by a development of the residual of the original  $\lambda[t1] \bullet t2$  after what has come so far. The requirement  $u \lesssim \lambda[t1] \bullet t2$  is not a strong enough constraint on the transitions in the initial segment, because  $\lambda[u1] \bullet u2 \lesssim \lambda[t1] \bullet t2$  can hold for  $t2$  and  $u2$  cointial, but otherwise without any particular relationship between their sets of marked redexes. In particular, this can occur when  $u2$  and  $t2$  occur as subterms that can be deleted by the contraction of an outer redex. So we need to introduce a notion of containment between terms that is stronger and more “syntactic” than  $\lesssim$ . The notion “subsumed by” defined below serves this purpose. Term  $u$  is subsumed by term  $t$  if both terms are arrows with exactly the same form except that  $t$  may contain  $\lambda[t1] \bullet t2$  (a marked redex) in places where  $u$  contains  $\lambda[t1] \circ t2$ .

**fun** *subs* (infix  $\sqsubseteq$  50)  
**where**  $\llbracket i \rrbracket \sqsubseteq \llbracket i' \rrbracket \longleftrightarrow i = i'$   
 $\lambda[t] \sqsubseteq \lambda[t'] \longleftrightarrow t \sqsubseteq t'$   
 $t \circ u \sqsubseteq t' \circ u' \longleftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u'$   
 $\lambda[t] \circ u \sqsubseteq \lambda[t'] \bullet u' \longleftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u'$   
 $\lambda[t] \bullet u \sqsubseteq \lambda[t'] \bullet u' \longleftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u'$   
 $- \sqsubseteq - \longleftrightarrow \text{False}$

**lemma** *subs-implies-prfx*:

**shows**  $t \sqsubseteq u \implies t \lesssim u$

**apply** (*induct*  $t$  *arbitrary*:  $u$ )

**apply** *auto*[1]

**using** *subs.elims*(2)

**apply** *fastforce*

**proof** –

**show**  $\bigwedge t. [\bigwedge u. t \sqsubseteq u \implies t \lesssim u; \lambda[t] \sqsubseteq u] \implies \lambda[t] \lesssim u$  **for**  $u$

**by** (*cases*  $u$ , *auto*) *fastforce*

**show**  $\bigwedge t2. [\bigwedge u1. t1 \sqsubseteq u1 \implies t1 \lesssim u1;$

$\bigwedge u2. t2 \sqsubseteq u2 \implies t2 \lesssim u2;$

$t1 \circ t2 \sqsubseteq u]$

$\implies t1 \circ t2 \lesssim u$  **for**  $t1$   $u$

**apply** (*cases*  $t1$ ; *cases*  $u$ )

**apply** *simp-all*

**apply** *fastforce+*

**apply** (*metis* *Ide-Subst con-char lambda.sel*(2) *subs.simps*(2) *prfx-Lam-iff prfx-char*)

```

      prfx-implies-con)
    by fastforce+
  show  $\bigwedge t1\ t2. [\bigwedge u1. t1 \sqsubseteq u1 \implies t1 \lesssim u1;$ 
       $\bigwedge u2. t2 \sqsubseteq u2 \implies t2 \lesssim u2;$ 
       $\lambda[t1] \bullet t2 \sqsubseteq u]$ 
       $\implies \lambda[t1] \bullet t2 \lesssim u$  for  $u$ 
    using Ide-Subst
    apply (cases u, simp-all)
    by (metis Ide.simps(1))
qed

```

The following is an example showing that two terms can be related by  $\lesssim$  without being related by  $\sqsubseteq$ .

```

lemma subs-example:
shows  $\lambda[\langle 1 \rangle] \bullet (\lambda[\langle 0 \rangle] \bullet \langle 0 \rangle) \lesssim \lambda[\langle 1 \rangle] \bullet (\lambda[\langle 0 \rangle] \circ \langle 0 \rangle) = True$ 
and  $\lambda[\langle 1 \rangle] \bullet (\lambda[\langle 0 \rangle] \bullet \langle 0 \rangle) \sqsubseteq \lambda[\langle 1 \rangle] \bullet (\lambda[\langle 0 \rangle] \circ \langle 0 \rangle) = False$ 
by auto

```

```

lemma subs-Ide:
shows  $[\text{ide } u; \text{Src } t = \text{Src } u] \implies u \sqsubseteq t$ 
using Ide-Src Ide-implies-Arr Ide-iff-Src-self
by (induct t arbitrary: u, simp-all) force+

```

```

lemma subs-App:
shows  $u \sqsubseteq t1 \circ t2 \iff \text{is-App } u \wedge \text{un-App1 } u \sqsubseteq t1 \wedge \text{un-App2 } u \sqsubseteq t2$ 
by (metis lambda.collapse(3) prfx-App-iff subs.simps(3) subs-implies-prfx)

```

end

```

context reduction-paths
begin

```

We now formally define a *development* of  $t$  to be an elementary reduction path  $U$  that is cointial with  $[t]$  and is such that each transition  $u$  in  $U$  is subsumed by the residual of  $t$  along the prefix of  $U$  coming before  $u$ . Stated another way, each transition in  $U$  corresponds to the contraction of a single redex that is the residual of a redex originally marked in  $t$ .

```

fun development
where development t []  $\longleftrightarrow \Lambda.Arr\ t$ 
  | development t (u # U)  $\longleftrightarrow$ 
     $\Lambda.\text{elementary-reduction } u \wedge u \sqsubseteq t \wedge \text{development } (t \setminus u)\ U$ 

```

```

lemma development-imp-Arr:
assumes development t U
shows  $\Lambda.Arr\ t$ 
using assms
by (metis  $\Lambda.\text{Con-implies-Arr2}$   $\Lambda.\text{Ide.simps}(1)$   $\Lambda.\text{ide-char}$   $\Lambda.\text{subs-implies-prfx}$ 
development.elims(2))

```

**lemma** *development-Ide*:  
**shows**  $\Lambda.Ide\ t \implies development\ t\ U \longleftrightarrow U = []$   
**using**  $\Lambda.Ide\text{-implies}\text{-Arr}$   
**apply** (*induct*  $U$  *arbitrary*:  $t$ )  
**apply** *auto*  
**by** (*meson*  $\Lambda.elementary\text{-reduction}\text{-not}\text{-ide}$   $\Lambda.ide\text{-backward}\text{-stable}$   $\Lambda.ide\text{-char}$   
 $\Lambda.subs\text{-implies}\text{-prfx}$ )

**lemma** *development-implies*:  
**shows**  $development\ t\ U \implies elementary\text{-reduction}\text{-path}\ U \wedge (U \neq [] \longrightarrow U \stackrel{*}{\lesssim} [t])$   
**apply** (*induct*  $U$  *arbitrary*:  $t$ )  
**using** *elementary-reduction-path-def*  
**apply** *simp*  
**proof** –  
**fix**  $t\ u\ U$   
**assume** *ind*:  $\bigwedge t. development\ t\ U \implies elementary\text{-reduction}\text{-path}\ U \wedge (U \neq [] \longrightarrow U \stackrel{*}{\lesssim} [t])$   
**show**  $development\ t\ (u \# U) \implies elementary\text{-reduction}\text{-path}\ (u \# U) \wedge (u \# U \neq [] \longrightarrow u \# U \stackrel{*}{\lesssim} [t])$   
**proof** (*cases*  $U = []$ )  
**assume**  $uU$ :  $development\ t\ (u \# U)$   
**show**  $U = [] \implies ?thesis$   
**using**  $uU\ \Lambda.subs\text{-implies}\text{-prfx}\ ide\text{-char}\ \Lambda.elementary\text{-reduction}\text{-is}\text{-arr}$   
 $elementary\text{-reduction}\text{-path}\text{-def}\ prfx\text{-implies}\text{-con}$   
**by** *force*  
**assume**  $U: U \neq []$   
**have**  $\Lambda.elementary\text{-reduction}\ u \wedge u \sqsubseteq t \wedge development\ (t \setminus u)\ U$   
**using**  $U\ uU\ development.elims(1)$  **by** *blast*  
**hence**  $1: \Lambda.elementary\text{-reduction}\ u \wedge elementary\text{-reduction}\text{-path}\ U \wedge u \sqsubseteq t \wedge (U \neq [] \longrightarrow U \stackrel{*}{\lesssim} [t \setminus u])$   
**using**  $U\ uU\ ind$  **by** *auto*  
**show** *?thesis*  
**proof** (*unfold* *elementary-reduction-path-def*, *intro* *conjI*)  
**show**  $u \# U = [] \vee Arr\ (u \# U) \wedge set\ (u \# U) \subseteq Collect\ \Lambda.elementary\text{-reduction}$   
**using**  $U\ 1$   
**by** (*metis* *Con-implies-Arr(1)* *Con-rec(2)* *con-char* *prfx-implies-con*  
 $elementary\text{-reduction}\text{-path}\text{-def}\ insert\text{-subset}\ list.simps(15)\ mem\text{-Collect}\text{-eq}$   
 $\Lambda.prfx\text{-implies}\text{-con}\ \Lambda.subs\text{-implies}\text{-prfx}$ )  
**show**  $u \# U \neq [] \longrightarrow u \# U \stackrel{*}{\lesssim} [t]$   
**proof** –  
**have**  $u \# U \stackrel{*}{\lesssim} [t] \longleftrightarrow ide\ ([u \setminus t] @ U \setminus [t \setminus u])$   
**using**  $1\ U\ Con\text{-rec}(2)\ Resid\text{-rec}(2)\ con\text{-char}\ prfx\text{-implies}\text{-con}$   
 $\Lambda.prfx\text{-implies}\text{-con}\ \Lambda.subs\text{-implies}\text{-prfx}$   
**by** *simp*  
**also** **have**  $\dots \longleftrightarrow True$   
**using**  $U\ 1\ ide\text{-char}\ Ide\text{-append}\text{-iff}_{PWE}\ [of\ [u \setminus t]\ U \setminus [t \setminus u]]$   
**by** (*metis* *Ide.simps(2)* *Ide-append*<sub>PWE</sub> *Src-resid* *Trg.simps(2)*  
 $\Lambda.apex\text{-sym}\ con\text{-char}\ \Lambda.subs\text{-implies}\text{-prfx}\ prfx\text{-implies}\text{-con}$ )  
**finally** **show** *?thesis* **by** *blast*

qed  
 qed  
 qed  
 qed

The converse of the previous result does not hold, because there could be a stage  $i$  at which  $u_i \lesssim t_i$ , but  $t_i$  deletes the redex contracted in  $u_i$ , so there is nothing forcing that redex to have been originally marked in  $t$ . So  $U$  being a development of  $t$  is a stronger property than  $U$  just being an elementary reduction path such that  $U \stackrel{*}{\lesssim} [t]$ .

**lemma** *development-append*:  
**shows**  $\llbracket \text{development } t \ U; \text{development } (t \ ^1 \setminus^* \ U) \ V \rrbracket \implies \text{development } t \ (U \ @ \ V)$   
**using** *development-imp-Arr null-char*  
**apply** (*induct U arbitrary: t V*)  
**apply** *auto*  
**by** (*metis Resid1x.simps(2-3) append-Nil neq-Nil-conv*)

**lemma** *development-map-Lam*:  
**shows**  $\text{development } t \ T \implies \text{development } \lambda[t] \ (\text{map } \Lambda.\text{Lam } T)$   
**using**  $\Lambda.\text{Arr-not-Nil}$  *development-imp-Arr*  
**by** (*induct T arbitrary: t*) *auto*

**lemma** *development-map-App-1*:  
**shows**  $\llbracket \text{development } t \ T; \Lambda.\text{Arr } u \rrbracket \implies \text{development } (t \circ u) \ (\text{map } (\lambda x. x \circ \Lambda.\text{Src } u) \ T)$   
**apply** (*induct T arbitrary: t*)  
**apply** (*simp add: \Lambda.Ide-implies-Arr*)  
**proof** –  
**fix**  $t \ T \ t'$   
**assume**  $\text{ind}: \bigwedge t. \llbracket \text{development } t \ T; \Lambda.\text{Arr } u \rrbracket$   
 $\implies \text{development } (t \circ u) \ (\text{map } (\lambda x. x \circ \Lambda.\text{Src } u) \ T)$   
**assume**  $t'T: \text{development } t \ (t' \ # \ T)$   
**assume**  $u: \Lambda.\text{Arr } u$   
**show**  $\text{development } (t \circ u) \ (\text{map } (\lambda x. x \circ \Lambda.\text{Src } u) \ (t' \ # \ T))$   
**using**  $u \ t'T \ \text{ind}$   
**apply** *simp*  
**using**  $\Lambda.\text{Arr-not-Nil}$   $\Lambda.\text{Ide-Src}$  *development-imp-Arr*  $\Lambda.\text{subs-Ide}$  **by** *force*  
**qed**

**lemma** *development-map-App-2*:  
**shows**  $\llbracket \Lambda.\text{Arr } t; \text{development } u \ U \rrbracket \implies \text{development } (t \circ u) \ (\text{map } (\lambda x. \Lambda.\text{App } (\Lambda.\text{Src } t) \ x)$   
 $U)$   
**apply** (*induct U arbitrary: u*)  
**apply** (*simp add: \Lambda.Ide-implies-Arr*)  
**proof** –  
**fix**  $u \ U \ u'$   
**assume**  $\text{ind}: \bigwedge u. \llbracket \Lambda.\text{Arr } t; \text{development } u \ U \rrbracket$   
 $\implies \text{development } (t \circ u) \ (\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) \ U)$   
**assume**  $u'U: \text{development } u \ (u' \ # \ U)$   
**assume**  $t: \Lambda.\text{Arr } t$   
**show**  $\text{development } (t \circ u) \ (\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } t)) \ (u' \ # \ U))$

```

using  $t u' U$  ind
apply simp
by (metis  $\Lambda$ .Coinitial-iff-Con  $\Lambda$ .Ide-Src  $\Lambda$ .Ide-iff-Src-self  $\Lambda$ .Ide-implies-Arr
      development-imp-Arr  $\Lambda$ .ide-char  $\Lambda$ .resid-Arr-Ide  $\Lambda$ .subs-Ide)
qed

```

### 3.4.1 Finiteness of Developments

A term  $t$  has the finite developments property if there exists a finite value that bounds the length of all developments of  $t$ . The goal of this section is to prove the Finite Developments Theorem: every term has the finite developments property.

```

definition FD
where FD  $t \equiv \exists n. \forall U. \text{development } t U \longrightarrow \text{length } U \leq n$ 

```

**end**

In [6], Hindley proceeds by using structural induction to establish a bound on the length of a development of a term. The only case that poses any difficulty is the case of a  $\beta$ -redex, which is  $\lambda[t] \bullet u$  in the notation used here. He notes that there is an easy bound on the length of a development of a special form in which all the contractions of residuals of  $t$  occur before the contraction of the top-level redex. The development first takes  $\lambda[t] \bullet u$  to  $\lambda[t'] \bullet u'$ , then to  $\text{subst } u' t'$ , then continues with independent developments of  $u'$ . The number of independent developments of  $u'$  is given by the number of free occurrences of  $\text{Var } \theta$  in  $t'$ . As there can be only finitely many such  $t'$ , we can use the maximum number of free occurrences of  $\text{Var } \theta$  over all such  $t'$  to bound the steps in the independent developments of  $u'$ .

In the general case, the problem is that reductions of residuals of  $t$  can increase the number of free occurrences of  $\text{Var } \theta$ , so we can't readily count them at any particular stage. Hindley shows that developments in which there are reductions of residuals of  $t$  that occur after the contraction of the top-level redex are equivalent to reductions of the special form, by a transformation with a bounded increase in length. This can be considered as a weak form of standardization for developments.

A later paper by de Vrijer [5] obtains an explicit function for the exact number of steps in a development of maximal length. His proof is very straightforward and amenable to formalization, and it is what we follow here. The main issue for us is that de Vrijer uses a classical representation of  $\lambda$ -terms, with variable names and  $\alpha$ -equivalence, whereas here we are using de Bruijn indices. This means that we have to discover the correct modification of de Vrijer's definitions to apply to the present situation.

```

context lambda-calculus
begin

```

Our first definition is that of the “multiplicity” of a free variable in a term. This is a count of the maximum number of times a variable could occur free in a term reachable in a development. The main issue in adjusting to de Bruijn indices is that the same variable will have different indices depending on the depth at which it occurs in the term. So, we need to keep track of how the indices of variables change as we move through the

term. Our modified definitions adjust the parameter to the multiplicity function on each recursive call, to account for the contextual depth (*i.e.* the number of binders on a path from the root of the term).

The definition of this function is readily understandable, except perhaps for the *Beta* case. The multiplicity  $mtp\ x\ (\lambda[t] \bullet u)$  has to be at least as large as  $mtp\ x\ (\lambda[t] \circ u)$ , to account for developments in which the top-level redex is not contracted. However, if the top-level redex  $\lambda[t] \bullet u$  is contracted, then the contractum is  $subst\ u\ t$ , so the multiplicity has to be at least as large as  $mtp\ x\ (subst\ u\ t)$ . This leads to the relation:

$$mtp\ x\ (\lambda[t] \bullet u) = \max (mtp\ x\ (\lambda[t] \circ u)) (mtp\ x\ (subst\ u\ t))$$

This is not directly suitable for use in a definition of the function  $mtp$ , because proving the termination is problematic. Instead, we have to guess the correct expression for  $mtp\ x\ (subst\ u\ t)$  and use that.

Now, each variable  $x$  in  $subst\ u\ t$  other than the variable  $\theta$  that is substituted for still has all the occurrences that it does in  $\lambda[t]$ . In addition, the variable being substituted for (which has index  $\theta$  in the outermost context of  $t$ ) will in general have multiple free occurrences in  $t$ , with a total multiplicity given by  $mtp\ \theta\ t$ . The substitution operation replaces each free occurrence by  $u$ , which has the effect of multiplying the multiplicity of a variable  $x$  in  $t$  by a factor of  $mtp\ \theta\ t$ . These considerations lead to the following:

$$mtp\ x\ (\lambda[t] \bullet u) = \max (mtp\ x\ \lambda[t] + mtp\ x\ u) (mtp\ x\ \lambda[t] + mtp\ x\ u * mtp\ \theta\ t)$$

However, we can simplify this to:

$$mtp\ x\ (\lambda[t] \bullet u) = mtp\ x\ \lambda[t] + mtp\ x\ u * \max\ 1\ (mtp\ \theta\ t)$$

and replace the  $mtp\ x\ \lambda[t]$  by  $mtp\ (Suc\ x)\ t$  to simplify the ordering necessary for the termination proof and allow it to be done automatically.

The final result is perhaps about the first thing one would think to write down, but there are possible ways to go wrong and it is of course still necessary to discover the proper form required for the various induction proofs. I followed a long path of rather more complicated-looking definitions, until I eventually managed to find the proper inductive forms for all the lemmas and eventually arrive back at this definition.

```

fun mtp :: nat ⇒ lambda ⇒ nat
where mtp x # = 0
  | mtp x «z» = (if z = x then 1 else 0)
  | mtp x λ[t] = mtp (Suc x) t
  | mtp x (t ∘ u) = mtp x t + mtp x u
  | mtp x (λ[t] • u) = mtp (Suc x) t + mtp x u * max 1 (mtp θ t)

```

The multiplicity function generalizes the free variable predicate. This is not actually used, but is included for explanatory purposes.

```

lemma mtp-gt-0-iff-in-FV:
shows mtp x t > 0 ⟷ x ∈ FV t
proof (induct t arbitrary: x)
  show ∧x. 0 < mtp x # ⟷ x ∈ FV #

```

```

    by simp
  show  $\bigwedge x z. 0 < mtp\ x\ \langle\langle z \rangle\rangle \longleftrightarrow x \in FV\ \langle\langle z \rangle\rangle$ 
    by auto
  show Lam:  $\bigwedge t x. (\bigwedge x. 0 < mtp\ x\ t \longleftrightarrow x \in FV\ t)$ 
     $\implies 0 < mtp\ x\ \lambda[t] \longleftrightarrow x \in FV\ \lambda[t]$ 
  proof -
    fix t and x :: nat
    assume ind:  $\bigwedge x. 0 < mtp\ x\ t \longleftrightarrow x \in FV\ t$ 
    show  $0 < mtp\ x\ \lambda[t] \longleftrightarrow x \in FV\ \lambda[t]$ 
      using ind
      apply auto
      apply (metis Diff-iff One-nat-def diff-Suc-1 empty-iff imageI insert-iff
        nat.distinct(1))
      by (metis Suc-pred neq0-conv)
  qed
  show  $\bigwedge t u x.$ 
     $\llbracket \bigwedge x. 0 < mtp\ x\ t \longleftrightarrow x \in FV\ t;$ 
     $\bigwedge x. 0 < mtp\ x\ u \longleftrightarrow x \in FV\ u \rrbracket$ 
     $\implies 0 < mtp\ x\ (t \circ u) \longleftrightarrow x \in FV\ (t \circ u)$ 
  by simp
  show  $\bigwedge t u x.$ 
     $\llbracket \bigwedge x. 0 < mtp\ x\ t \longleftrightarrow x \in FV\ t;$ 
     $\bigwedge x. 0 < mtp\ x\ u \longleftrightarrow x \in FV\ u \rrbracket$ 
     $\implies 0 < mtp\ x\ (\lambda[t] \bullet u) \longleftrightarrow x \in FV\ (\lambda[t] \bullet u)$ 
  proof -
    fix t u and x :: nat
    assume ind1:  $\bigwedge x. 0 < mtp\ x\ t \longleftrightarrow x \in FV\ t$ 
    assume ind2:  $\bigwedge x. 0 < mtp\ x\ u \longleftrightarrow x \in FV\ u$ 
    show  $0 < mtp\ x\ (\lambda[t] \bullet u) \longleftrightarrow x \in FV\ (\lambda[t] \bullet u)$ 
      using ind1 ind2
      apply simp
      by force
  qed
  qed

```

We now establish a fact about commutation of multiplicity and Raise that will be needed subsequently.

**lemma** *mtpE-eq-Raise*:

**shows**  $x < d \implies mtp\ x\ (Raise\ d\ k\ t) = mtp\ x\ t$

**by** (*induct t arbitrary: x k d*) *auto*

**lemma** *mtp-Raise-ind*:

**shows**  $\llbracket l \leq d; size\ t \leq s \rrbracket \implies mtp\ (x + d + k)\ (Raise\ l\ k\ t) = mtp\ (x + d)\ t$

**proof** (*induct s arbitrary: d x k l t*)

**show**  $\bigwedge d x k l. \llbracket l \leq d; size\ t \leq 0 \rrbracket \implies mtp\ (x + d + k)\ (Raise\ l\ k\ t) = mtp\ (x + d)\ t$

**for** *t*

**by** (*cases t*) *auto*

**show**  $\bigwedge s d x k l.$

$\llbracket \bigwedge d x k l t. \llbracket l \leq d; size\ t \leq s \rrbracket \implies mtp\ (x + d + k)\ (Raise\ l\ k\ t) = mtp\ (x + d)\ t;$

$l \leq d; \text{size } t \leq \text{Suc } s$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l k t) = \text{mtp } (x + d) t$

**for**  $t$

**proof** (*cases*  $t$ )

**show**  $\bigwedge d x k l s. t = \# \implies \text{mtp } (x + d + k) (\text{Raise } l k t) = \text{mtp } (x + d) t$   
**by** *simp*

**show**  $\bigwedge z d x k l s. \llbracket l \leq d; t = \langle z \rangle \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l k t) = \text{mtp } (x + d) t$

**by** *simp*

**show**  $\bigwedge u d x k l s. \llbracket l \leq d; \text{size } t \leq \text{Suc } s; t = \lambda[u];$   
 $(\bigwedge d x k l u. \llbracket l \leq d; \text{size } u \leq s \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l k u) = \text{mtp } (x + d) u \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l k t) = \text{mtp } (x + d) t$

**proof** –

**fix**  $u d x s$  **and**  $k l :: \text{nat}$

**assume**  $l: l \leq d$  **and**  $s: \text{size } t \leq \text{Suc } s$  **and**  $t: t = \lambda[u]$

**assume** *ind*:  $\bigwedge d x k l u. \llbracket l \leq d; \text{size } u \leq s \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l k u) = \text{mtp } (x + d) u$

**show**  $\text{mtp } (x + d + k) (\text{Raise } l k t) = \text{mtp } (x + d) t$

**proof** –

**have**  $\text{mtp } (x + d + k) (\text{Raise } l k t) = \text{mtp } (\text{Suc } (x + d + k)) (\text{Raise } (\text{Suc } l) k u)$   
**using**  $t$  **by** *simp*

**also have**  $\dots = \text{mtp } (x + \text{Suc } d) u$

**proof** –

**have**  $\text{size } u \leq s$   
**using**  $t$  **by** *force*

**thus** *?thesis*  
**using**  $l s$  *ind* [*of*  $\text{Suc } l \text{ Suc } d$ ] **by** *simp*

**qed**

**also have**  $\dots = \text{mtp } (x + d) t$   
**using**  $t$  **by** *auto*

**finally show** *?thesis* **by** *blast*

**qed**

**qed**

**show**  $\bigwedge t1 t2 d x k l s.$   
 $\llbracket \bigwedge d x k l t1. \llbracket l \leq d; \text{size } t1 \leq s \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l k t1) = \text{mtp } (x + d) t1;$   
 $\bigwedge d x k l t2. \llbracket l \leq d; \text{size } t2 \leq s \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l k t2) = \text{mtp } (x + d) t2;$   
 $l \leq d; \text{size } t \leq \text{Suc } s; t = t1 \circ t2 \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l k t) = \text{mtp } (x + d) t$

**proof** –

**fix**  $t1 t2 s$

**assume**  $s: \text{size } t \leq \text{Suc } s$  **and**  $t: t = t1 \circ t2$

**have**  $\text{size } t1 \leq s \wedge \text{size } t2 \leq s$   
**using**  $s t$  **by** *auto*

**thus**  $\bigwedge d x k l.$   
 $\llbracket \bigwedge d x k l t1. \llbracket l \leq d; \text{size } t1 \leq s \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l k t1) = \text{mtp } (x + d) t1;$

$\wedge d \ x \ k \ l \ t2. \llbracket l \leq d; \text{size } t2 \leq s \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l \ k \ t2) = \text{mtp } (x + d) \ t2;$   
 $l \leq d; \text{size } t \leq \text{Suc } s; t = t1 \circ t2 \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l \ k \ t) = \text{mtp } (x + d) \ t$   
**by** *simp*  
**qed**  
**show**  $\wedge t1 \ t2 \ d \ x \ k \ l \ s.$   
 $\llbracket \wedge d \ x \ k \ l \ t1. \llbracket l \leq d; \text{size } t1 \leq s \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l \ k \ t1) = \text{mtp } (x + d) \ t1;$   
 $\wedge d \ x \ k \ l \ t2. \llbracket l \leq d; \text{size } t2 \leq s \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l \ k \ t2) = \text{mtp } (x + d) \ t2;$   
 $l \leq d; \text{size } t \leq \text{Suc } s; t = \lambda[t1] \bullet t2 \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l \ k \ t) = \text{mtp } (x + d) \ t$   
**proof** –  
**fix**  $t1 \ t2 \ d \ x \ s$  **and**  $k \ l :: \text{nat}$   
**assume**  $l: l \leq d$  **and**  $s: \text{size } t \leq \text{Suc } s$  **and**  $t: t = \lambda[t1] \bullet t2$   
**assume**  $\text{ind}: \wedge d \ x \ k \ l \ N. \llbracket l \leq d; \text{size } N \leq s \rrbracket$   
 $\implies \text{mtp } (x + d + k) (\text{Raise } l \ k \ N) = \text{mtp } (x + d) \ N$   
**show**  $\text{mtp } (x + d + k) (\text{Raise } l \ k \ t) = \text{mtp } (x + d) \ t$   
**proof** –  
**have**  $1: \text{size } t1 \leq s \wedge \text{size } t2 \leq s$   
**using**  $s \ t$  **by** *auto*  
**have**  $\text{mtp } (x + d + k) (\text{Raise } l \ k \ t) =$   
 $\text{mtp } (\text{Suc } (x + d + k)) (\text{Raise } (\text{Suc } l) \ k \ t1) +$   
 $\text{mtp } (x + d + k) (\text{Raise } l \ k \ t2) * \text{max } 1 (\text{mtp } 0 (\text{Raise } (\text{Suc } l) \ k \ t1))$   
**using**  $t \ l$  **by** *simp*  
**also have**  $\dots = \text{mtp } (\text{Suc } (x + d + k)) (\text{Raise } (\text{Suc } l) \ k \ t1) +$   
 $\text{mtp } (x + d) \ t2 * \text{max } 1 (\text{mtp } 0 (\text{Raise } (\text{Suc } l) \ k \ t1))$   
**using**  $l \ 1$  *ind* **by** *auto*  
**also have**  $\dots = \text{mtp } (x + \text{Suc } d) \ t1 + \text{mtp } (x + d) \ t2 * \text{max } 1 (\text{mtp } 0 \ t1)$   
**proof** –  
**have**  $\text{mtp } (x + \text{Suc } d + k) (\text{Raise } (\text{Suc } l) \ k \ t1) = \text{mtp } (x + \text{Suc } d) \ t1$   
**using**  $l \ 1$  *ind* [of  $\text{Suc } l \ \text{Suc } d \ t1$ ] **by** *simp*  
**moreover have**  $\text{mtp } 0 (\text{Raise } (\text{Suc } l) \ k \ t1) = \text{mtp } 0 \ t1$   
  
**using**  $l \ 1$  *ind* [of  $\text{Suc } l \ \text{Suc } d \ t1 \ k$ ] *mtpE-eq-Raise* **by** *simp*  
**ultimately show** *?thesis*  
**by** *simp*  
**qed**  
**also have**  $\dots = \text{mtp } (x + d) \ t$   
**using**  $t$  **by** *auto*  
**finally show** *?thesis* **by** *blast*  
**qed**  
**qed**  
**qed**  
**qed**

**lemma** *mtp-Raise*:  
**assumes**  $l \leq d$

**shows**  $mtp (x + d + k) (Raise\ l\ k\ t) = mtp (x + d) t$   
**using** *assms mtp-Raise-ind* **by** *blast*

**lemma** *mtp-Raise'*:

**shows**  $mtp\ l\ (Raise\ l\ (Suc\ k)\ t) = 0$   
**by** (*induct t arbitrary: k l*) *auto*

**lemma** *mtp-raise*:

**shows**  $mtp (x + Suc\ d) (raise\ d\ t) = mtp (Suc\ x) t$   
**by** (*metis Suc-eq-plus1 add.assoc le-add2 le-add-same-cancel2 mtp-Raise plus-1-eq-Suc*)

**lemma** *mtp-Subst-cancel*:

**shows**  $mtp\ k\ (Subst\ (Suc\ d + k)\ u\ t) = mtp\ k\ t$

**proof** (*induct t arbitrary: k d*)

**show**  $\bigwedge k\ d. mtp\ k\ (Subst\ (Suc\ d + k)\ u\ \#) = mtp\ k\ \#$   
**by** *simp*

**show**  $\bigwedge k\ z\ d. mtp\ k\ (Subst\ (Suc\ d + k)\ u\ \langle z \rangle) = mtp\ k\ \langle z \rangle$

**using** *mtp-Raise'*

**apply** *auto*

**by** (*metis add-Suc-right add-Suc-shift order-refl raise-plus*)

**show**  $\bigwedge t\ k\ d. (\bigwedge k\ d. mtp\ k\ (Subst\ (Suc\ d + k)\ u\ t) = mtp\ k\ t)$

$\implies mtp\ k\ (Subst\ (Suc\ d + k)\ u\ \lambda[t]) = mtp\ k\ \lambda[t]$

**by** (*metis Subst.simps(3) add-Suc-right mtp.simps(3)*)

**show**  $\bigwedge t1\ t2\ k\ d.$

$\llbracket \bigwedge k\ d. mtp\ k\ (Subst\ (Suc\ d + k)\ u\ t1) = mtp\ k\ t1;$

$\bigwedge k\ d. mtp\ k\ (Subst\ (Suc\ d + k)\ u\ t2) = mtp\ k\ t2 \rrbracket$

$\implies mtp\ k\ (Subst\ (Suc\ d + k)\ u\ (t1 \circ t2)) = mtp\ k\ (t1 \circ t2)$

**by** *auto*

**show**  $\bigwedge t1\ t2\ k\ d.$

$\llbracket \bigwedge k\ d. mtp\ k\ (Subst\ (Suc\ d + k)\ u\ t1) = mtp\ k\ t1;$

$\bigwedge k\ d. mtp\ k\ (Subst\ (Suc\ d + k)\ u\ t2) = mtp\ k\ t2 \rrbracket$

$\implies mtp\ k\ (Subst\ (Suc\ d + k)\ u\ (\lambda[t1] \bullet t2)) = mtp\ k\ (\lambda[t1] \bullet t2)$

**using** *mtp-Raise'*

**apply** *auto*

**by** (*metis Nat.add-0-right add-Suc-right*)

**qed**

**lemma** *mtp<sub>0</sub>-Subst-cancel*:

**shows**  $mtp\ 0\ (Subst\ (Suc\ d)\ u\ t) = mtp\ 0\ t$   
**using** *mtp-Subst-cancel [of 0]* **by** *simp*

We can now (!) prove the desired generalization of de Vrijer's formula for the commutation of multiplicity and substitution. This is the main lemma whose form is difficult to find. To get this right, the proper relationships have to exist between the various depth parameters to *Subst* and the arguments to *mtp*.

**lemma** *mtp-Subst'*:

**shows**  $mtp (x + Suc\ d) (Subst\ d\ u\ t) = mtp (x + Suc\ (Suc\ d)) t + mtp (Suc\ x) u * mtp\ d\ t$

**proof** (*induct t arbitrary: d x u*)

**show**  $\bigwedge d\ x\ u. mtp (x + Suc\ d) (Subst\ d\ u\ \#) =$

$$mtp (x + Suc (Suc d)) \# + mtp (Suc x) u * mtp d \#$$
**by** *simp*

**show**  $\bigwedge z d x u. mtp (x + Suc d) (Subst d u \llbracket z \rrbracket) =$   

$$mtp (x + Suc (Suc d)) \llbracket z \rrbracket + mtp (Suc x) u * mtp d \llbracket z \rrbracket$$
**using** *mtp-raise by auto*

**show**  $\bigwedge t d x u.$   

$$(\bigwedge d x u. mtp (x + Suc d) (Subst d u t) =$$
  

$$mtp (x + Suc (Suc d)) t + mtp (Suc x) u * mtp d t)$$
  

$$\implies mtp (x + Suc d) (Subst d u \lambda[t]) =$$
  

$$mtp (x + Suc (Suc d)) \lambda[t] + mtp (Suc x) u * mtp d \lambda[t]$$

**proof** –

**fix**  $t u d x$

**assume** *ind*:  $\bigwedge d x N. mtp (x + Suc d) (Subst d N t) =$   

$$mtp (x + Suc (Suc d)) t + mtp (Suc x) N * mtp d t$$

**have**  $mtp (x + Suc d) (Subst d u \lambda[t]) =$   

$$mtp (Suc x + Suc (Suc d)) t +$$
  

$$mtp (x + Suc (Suc d)) (raise (Suc d) u) * mtp (Suc d) t$$

**using** *ind mtp-raise add-Suc-shift*

**by** (*metis Subst.simps(3) add-Suc-right mtp.simps(3)*)

**also have**  $\dots = mtp (x + Suc (Suc d)) \lambda[t] + mtp (Suc x) u * mtp d \lambda[t]$

**using** *Raise-Suc*

**by** (*metis add-Suc-right add-Suc-shift mtp.simps(3) mtp-raise*)

**finally show**  $mtp (x + Suc d) (Subst d u \lambda[t]) =$   

$$mtp (x + Suc (Suc d)) \lambda[t] + mtp (Suc x) u * mtp d \lambda[t]$$

**by** *blast*

**qed**

**show**  $\bigwedge t1 t2 u d x.$   

$$\llbracket \bigwedge d x u. mtp (x + Suc d) (Subst d u t1) =$$
  

$$mtp (x + Suc (Suc d)) t1 + mtp (Suc x) u * mtp d t1;$$
  

$$\bigwedge d x u. mtp (x + Suc d) (Subst d u t2) =$$
  

$$mtp (x + Suc (Suc d)) t2 + mtp (Suc x) u * mtp d t2 \rrbracket$$
  

$$\implies mtp (x + Suc d) (Subst d u (t1 \circ t2)) =$$
  

$$mtp (x + Suc (Suc d)) (t1 \circ t2) + mtp (Suc x) u * mtp d (t1 \circ t2)$$

**by** (*simp add: add-mult-distrib2*)

**show**  $\bigwedge t1 t2 u d x.$   

$$\llbracket \bigwedge d x N. mtp (x + Suc d) (Subst d N t1) =$$
  

$$mtp (x + Suc (Suc d)) t1 + mtp (Suc x) N * mtp d t1;$$
  

$$\bigwedge d x N. mtp (x + Suc d) (Subst d N t2) =$$
  

$$mtp (x + Suc (Suc d)) t2 + mtp (Suc x) N * mtp d t2 \rrbracket$$
  

$$\implies mtp (x + Suc d) (Subst d u (\lambda[t1] \bullet t2)) =$$
  

$$mtp (x + Suc (Suc d)) (\lambda[t1] \bullet t2) + mtp (Suc x) u * mtp d (\lambda[t1] \bullet t2)$$

**proof** –

**fix**  $t1 t2 u d x$

**assume** *ind1*:  $\bigwedge d x N. mtp (x + Suc d) (Subst d N t1) =$   

$$mtp (x + Suc (Suc d)) t1 + mtp (Suc x) N * mtp d t1$$

**assume** *ind2*:  $\bigwedge d x N. mtp (x + Suc d) (Subst d N t2) =$   

$$mtp (x + Suc (Suc d)) t2 + mtp (Suc x) N * mtp d t2$$

**show**  $mtp (x + Suc d) (Subst d u (\lambda[t1] \bullet t2)) =$   

$$mtp (x + Suc (Suc d)) (\lambda[t1] \bullet t2) + mtp (Suc x) u * mtp d (\lambda[t1] \bullet t2)$$

```

proof –
  let ?A = mtp (Suc x + Suc (Suc d)) t1
  let ?B = mtp (Suc x + Suc d) t2
  let ?M1 = mtp (Suc d) t1
  let ?M2 = mtp d t2
  let ?M10 = mtp 0 (Subst (Suc d) u t1)
  let ?M10' = mtp 0 t1
  let ?N = mtp (Suc x) u
  have mtp (x + Suc d) (Subst d u (λ[t1] • t2)) =
    mtp (x + Suc d) (λ[Subst (Suc d) u t1] • Subst d u t2)
    by simp
  also have ... = mtp (x + Suc (Suc d)) (Subst (Suc d) u t1) +
    mtp (x + Suc d) (Subst d u t2) *
    max 1 (mtp 0 (Subst (Suc d) u t1))
    by simp
  also have ... = (?A + ?N * ?M1) + (?B + ?N * ?M2) * max 1 ?M10
    using ind1 ind2 add-Suc-shift by presburger
  also have ... = ?A + ?N * ?M1 + ?B * max 1 ?M10 + ?N * ?M2 * max 1 ?M10
    by algebra
  also have ... = ?A + ?B * max 1 ?M10' + ?N * ?M1 + ?N * ?M2 * max 1 ?M10'
  proof –
    have ?M10 = ?M10'
      using mtp0-Subst-cancel by blast
      thus ?thesis by auto
    qed
  also have ... = ?A + ?B * max 1 ?M10' + ?N * (?M1 + ?M2 * max 1 ?M10')
    by algebra
  also have ... = mtp (Suc x + Suc d) (λ[t1] • t2) + mtp (Suc x) u * mtp d (λ[t1] • t2)
    by simp
  finally show ?thesis by simp
qed
qed
qed

```

The following lemma provides expansions that apply when the parameter to *mtp* is 0, as opposed to the previous lemma, which only applies for parameters greater than 0.

**lemma** *mtp-Subst*:

**shows**  $mtp\ k\ (Subst\ k\ u\ t) = mtp\ (Suc\ k)\ t + mtp\ k\ (raise\ k\ u) * mtp\ k\ t$

**proof** (*induct t arbitrary: u k*)

**show**  $\bigwedge u\ k. mtp\ k\ (Subst\ k\ u\ \#) = mtp\ (Suc\ k)\ \# + mtp\ k\ (raise\ k\ u) * mtp\ k\ \#$

**by simp**

**show**  $\bigwedge x\ u\ k. mtp\ k\ (Subst\ k\ u\ \langle\langle x \rangle\rangle) =$

$mtp\ (Suc\ k)\ \langle\langle x \rangle\rangle + mtp\ k\ (raise\ k\ u) * mtp\ k\ \langle\langle x \rangle\rangle$

**by auto**

**show**  $\bigwedge t\ u\ k. (\bigwedge u\ k. mtp\ k\ (Subst\ k\ u\ t) = mtp\ (Suc\ k)\ t + mtp\ k\ (raise\ k\ u) * mtp\ k\ t)$

$\implies mtp\ k\ (Subst\ k\ u\ \lambda[t]) =$

$mtp\ (Suc\ k)\ \lambda[t] + mtp\ k\ (Raise\ 0\ k\ u) * mtp\ k\ \lambda[t]$

**using** *mtp-Raise [of 0]*

```

apply auto
by (metis add.left-neutral)
show  $\bigwedge t1\ t2\ u\ k.$ 
  
$$\begin{aligned}
& \llbracket \bigwedge u\ k. mtp\ k\ (Subst\ k\ u\ t1) = mtp\ (Suc\ k)\ t1 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t1; \\
& \bigwedge u\ k. mtp\ k\ (Subst\ k\ u\ t2) = mtp\ (Suc\ k)\ t2 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t2 \rrbracket \\
& \implies mtp\ k\ (Subst\ k\ u\ (t1\ \circ\ t2)) = \\
& \quad mtp\ (Suc\ k)\ (t1\ \circ\ t2) + mtp\ k\ (raise\ k\ u) * mtp\ k\ (t1\ \circ\ t2)
\end{aligned}$$

by (auto simp add: distrib-left)
show  $\bigwedge t1\ t2\ u\ k.$ 
  
$$\begin{aligned}
& \llbracket \bigwedge u\ k. mtp\ k\ (Subst\ k\ u\ t1) = mtp\ (Suc\ k)\ t1 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t1; \\
& \bigwedge u\ k. mtp\ k\ (Subst\ k\ u\ t2) = mtp\ (Suc\ k)\ t2 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t2 \rrbracket \\
& \implies mtp\ k\ (Subst\ k\ u\ (\lambda[t1] \bullet t2)) = \\
& \quad mtp\ (Suc\ k)\ (\lambda[t1] \bullet t2) + mtp\ k\ (raise\ k\ u) * mtp\ k\ (\lambda[t1] \bullet t2)
\end{aligned}$$

proof -
fix t1 t2 u k
assume ind1:  $\bigwedge u\ k. mtp\ k\ (Subst\ k\ u\ t1) =$ 

$$mtp\ (Suc\ k)\ t1 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t1$$

assume ind2:  $\bigwedge u\ k. mtp\ k\ (Subst\ k\ u\ t2) =$ 

$$mtp\ (Suc\ k)\ t2 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t2$$

show  $mtp\ k\ (Subst\ k\ u\ (\lambda[t1] \bullet t2)) =$ 

$$mtp\ (Suc\ k)\ (\lambda[t1] \bullet t2) + mtp\ k\ (raise\ k\ u) * mtp\ k\ (\lambda[t1] \bullet t2)$$

proof -
have  $mtp\ (Suc\ k)\ (Raise\ 0\ (Suc\ k)\ u) * mtp\ (Suc\ k)\ t1 +$ 

$$(mtp\ (Suc\ k)\ t2 + mtp\ k\ (Raise\ 0\ k\ u) * mtp\ k\ t2) * max\ (Suc\ 0)\ (mtp\ 0\ t1) =$$


$$mtp\ (Suc\ k)\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1) +$$


$$mtp\ k\ (Raise\ 0\ k\ u) * (mtp\ (Suc\ k)\ t1 + mtp\ k\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1))$$

proof -
have  $mtp\ (Suc\ k)\ (Raise\ 0\ (Suc\ k)\ u) * mtp\ (Suc\ k)\ t1 +$ 

$$(mtp\ (Suc\ k)\ t2 + mtp\ k\ (Raise\ 0\ k\ u) * mtp\ k\ t2) * max\ (Suc\ 0)\ (mtp\ 0\ t1) =$$


$$mtp\ (Suc\ k)\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1) +$$


$$mtp\ (Suc\ k)\ (Raise\ 0\ (Suc\ k)\ u) * mtp\ (Suc\ k)\ t1 +$$


$$mtp\ k\ (Raise\ 0\ k\ u) * mtp\ k\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1)$$

by algebra
also have  $... = mtp\ (Suc\ k)\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1) +$ 

$$mtp\ (Suc\ k)\ (Raise\ 0\ (Suc\ k)\ u) * mtp\ (Suc\ k)\ t1 +$$


$$mtp\ 0\ u * mtp\ k\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1)$$

using mtp-Raise [of 0 0 0 k u] by auto
also have  $... = mtp\ (Suc\ k)\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1) +$ 

$$mtp\ k\ (Raise\ 0\ k\ u) *$$


$$(mtp\ (Suc\ k)\ t1 + mtp\ k\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1))$$

by (metis (no-types, lifting) ab-semigroup-add-class.add-ac(1)
ab-semigroup-mult-class.mult-ac(1) add-mult-distrib2 le-add1 mtp-Raise
plus-nat.add-0)
finally show ?thesis by blast
qed
thus ?thesis
using ind1 ind2 mtp0-Subst-cancel by auto
qed
qed

```

qed

**lemma** *mtp0-subst-le*:

**shows**  $mtp\ 0\ (subst\ u\ t) \leq mtp\ 1\ t + mtp\ 0\ u * max\ 1\ (mtp\ 0\ t)$

**proof** (*cases t*)

**show**  $t = \# \implies mtp\ 0\ (subst\ u\ t) \leq mtp\ 1\ t + mtp\ 0\ u * max\ 1\ (mtp\ 0\ t)$

**by** *auto*

**show**  $\bigwedge z. t = \langle z \rangle \implies mtp\ 0\ (subst\ u\ t) \leq mtp\ 1\ t + mtp\ 0\ u * max\ 1\ (mtp\ 0\ t)$

**using** *Raise-0* **by** *force*

**show**  $\bigwedge P. t = \lambda[P] \implies mtp\ 0\ (subst\ u\ t) \leq mtp\ 1\ t + mtp\ 0\ u * max\ 1\ (mtp\ 0\ t)$

**using** *mtp-Subst [of 0 u t] Raise-0* **by** *force*

**show**  $\bigwedge t1\ t2. t = t1 \circ t2 \implies mtp\ 0\ (subst\ u\ t) \leq mtp\ 1\ t + mtp\ 0\ u * max\ 1\ (mtp\ 0\ t)$

**using** *mtp-Subst Raise-0 add-mult-distrib2 nat-mult-max-right* **by** *auto*

**show**  $\bigwedge t1\ t2. t = \lambda[t1] \bullet t2 \implies mtp\ 0\ (subst\ u\ t) \leq mtp\ 1\ t + mtp\ 0\ u * max\ 1\ (mtp\ 0\ t)$

**using** *mtp-Subst Raise-0*

**by** (*metis Nat.add-0-right dual-order.eq-iff max-def mult.commute mult-zero-left not-less-eq-eq plus-1-eq-Suc trans-le-add1*)

qed

**lemma** *elementary-reduction-nonincreases-mtp*:

**shows**  $\llbracket elementary\ reduction\ u; u \sqsubseteq t \rrbracket \implies mtp\ x\ (resid\ t\ u) \leq mtp\ x\ t$

**proof** (*induct t arbitrary: u x*)

**show**  $\bigwedge u\ x. \llbracket elementary\ reduction\ u; u \sqsubseteq \# \rrbracket \implies mtp\ x\ (resid\ \# \ u) \leq mtp\ x\ \#$

**by** *simp*

**show**  $\bigwedge x\ u\ i. \llbracket elementary\ reduction\ u; u \sqsubseteq \langle i \rangle \rrbracket$

$\implies mtp\ x\ (resid\ \langle i \rangle \ u) \leq mtp\ x\ \langle i \rangle$

**by** (*meson Ide.simps(2) elementary-reduction-not-ide ide-backward-stable ide-char subs-implies-prfx*)

**fix** *u*

**show**  $\bigwedge t\ x. \llbracket \bigwedge u\ x. \llbracket elementary\ reduction\ u; u \sqsubseteq t \rrbracket \implies mtp\ x\ (resid\ t\ u) \leq mtp\ x\ t;$

$elementary\ reduction\ u; u \sqsubseteq \lambda[t] \rrbracket$

$\implies mtp\ x\ (\lambda[t] \setminus u) \leq mtp\ x\ \lambda[t]$

**by** (*cases u*) *auto*

**show**  $\bigwedge t1\ t2\ x.$

$\llbracket \bigwedge u\ x. \llbracket elementary\ reduction\ u; u \sqsubseteq t1 \rrbracket \implies mtp\ x\ (resid\ t1\ u) \leq mtp\ x\ t1;$

$\bigwedge u\ x. \llbracket elementary\ reduction\ u; u \sqsubseteq t2 \rrbracket \implies mtp\ x\ (resid\ t2\ u) \leq mtp\ x\ t2;$

$elementary\ reduction\ u; u \sqsubseteq t1 \circ t2 \rrbracket$

$\implies mtp\ x\ (resid\ (t1 \circ t2)\ u) \leq mtp\ x\ (t1 \circ t2)$

**apply** (*cases u*)

**apply** *auto*

**apply** (*metis Coinitial-iff-Con add-mono-thms-linordered-semiring(3) resid-Arr-Ide*)

**by** (*metis Coinitial-iff-Con add-mono-thms-linordered-semiring(2) resid-Arr-Ide*)

**show**  $\bigwedge t1\ t2\ x.$

$\llbracket \bigwedge u1\ x. \llbracket elementary\ reduction\ u1; u1 \sqsubseteq t1 \rrbracket \implies mtp\ x\ (resid\ t1\ u1) \leq mtp\ x\ t1;$

$\bigwedge u2\ x. \llbracket elementary\ reduction\ u2; u2 \sqsubseteq t2 \rrbracket \implies mtp\ x\ (resid\ t2\ u2) \leq mtp\ x\ t2;$

$elementary\ reduction\ u; u \sqsubseteq \lambda[t1] \bullet t2 \rrbracket$

$\implies mtp\ x\ ((\lambda[t1] \bullet t2) \setminus u) \leq mtp\ x\ (\lambda[t1] \bullet t2)$

**proof** –

```

fix t1 t2 x
assume ind1:  $\bigwedge u1 x. \llbracket \text{elementary-reduction } u1; u1 \sqsubseteq t1 \rrbracket$ 
            $\implies mtp\ x\ (t1 \setminus u1) \leq mtp\ x\ t1$ 
assume ind2:  $\bigwedge u2 x. \llbracket \text{elementary-reduction } u2; u2 \sqsubseteq t2 \rrbracket$ 
            $\implies mtp\ x\ (t2 \setminus u2) \leq mtp\ x\ t2$ 
assume u: elementary-reduction u
assume subs:  $u \sqsubseteq \lambda[t1] \bullet t2$ 
have 1: is-App u  $\vee$  is-Beta u
  using subs by (metis prfx-Beta-iff subs-implies-prfx)
have is-App u  $\implies mtp\ x\ ((\lambda[t1] \bullet t2) \setminus u) \leq mtp\ x\ (\lambda[t1] \bullet t2)$ 
proof –
  assume 2: is-App u
  obtain u1 u2 where u1u2:  $u = \lambda[u1] \circ u2$ 
  using 2 u
by (metis ConD(3) Con-implies-is-Lam-iff-is-Lam Con-sym con-def is-App-def is-Lam-def
      lambda.disc(8) null-char prfx-implies-con subs subs-implies-prfx)
  have  $mtp\ x\ ((\lambda[t1] \bullet t2) \setminus u) = mtp\ x\ (\lambda[t1 \setminus u1] \bullet (t2 \setminus u2))$ 
  using u1u2 subs
  by (metis Con-sym Ide.simps(1) ide-char resid.simps(6) subs-implies-prfx)
  also have ... =  $mtp\ (Suc\ x)\ (resid\ t1\ u1) +$ 
               $mtp\ x\ (resid\ t2\ u2) * max\ 1\ (mtp\ 0\ (resid\ t1\ u1))$ 
  by simp
  also have ...  $\leq mtp\ (Suc\ x)\ t1 + mtp\ x\ (resid\ t2\ u2) * max\ 1\ (mtp\ 0\ (resid\ t1\ u1))$ 
  using u1u2 ind1 [of u1 Suc x] con-sym ide-char resid-arr-ide prfx-implies-con
      subs subs-implies-prfx u
  by force
  also have ...  $\leq mtp\ (Suc\ x)\ t1 + mtp\ x\ t2 * max\ 1\ (mtp\ 0\ (resid\ t1\ u1))$ 
  using u1u2 ind2 [of u2 x]
  by (metis (no-types, lifting) Con-implies-Coinitial-ind add-left-mono
      dual-order.eq-iff elementary-reduction.simps(4) lambda.disc(11)
      mult-le-cancel2 prfx-App-iff resid.simps(31) resid-Arr-Ide subs subs.simps(4)
      subs-implies-prfx u)
  also have ...  $\leq mtp\ (Suc\ x)\ t1 + mtp\ x\ t2 * max\ 1\ (mtp\ 0\ t1)$ 
  using ind1 [of u1 0]
  by (metis Con-implies-Coinitial-ind Ide.simps(3) elementary-reduction.simps(3)
      elementary-reduction.simps(4) lambda.disc(11) max.mono mult-le-mono
      nat-add-left-cancel-le nat-le-linear prfx-App-iff resid.simps(31) resid-Arr-Ide
      subs subs.simps(4) subs-implies-prfx u u1u2)
  also have ... =  $mtp\ x\ (\lambda[t1] \bullet t2)$ 
  by auto
  finally show  $mtp\ x\ ((\lambda[t1] \bullet t2) \setminus u) \leq mtp\ x\ (\lambda[t1] \bullet t2)$  by blast
qed
moreover have is-Beta u  $\implies mtp\ x\ ((\lambda[t1] \bullet t2) \setminus u) \leq mtp\ x\ (\lambda[t1] \bullet t2)$ 
proof –
  assume 2: is-Beta u
  obtain u1 u2 where u1u2:  $u = \lambda[u1] \bullet u2$ 
  using 2 u is-Beta-def by auto
  have  $mtp\ x\ ((\lambda[t1] \bullet t2) \setminus u) = mtp\ x\ (subst\ (t2 \setminus u2)\ (t1 \setminus u1))$ 
  using u1u2 subs

```

```

    by (metis con-def con-sym null-char prfx-implies-con resid.simps(4) subs-implies-prfx)
  also have ... ≤ mtp (Suc x) (resid t1 u1) +
    mtp x (resid t2 u2) * max 1 (mtp 0 (resid t1 u1))
    apply (cases x = 0)
    using mtp0-subst-le Raise-0 mtp-Subst' [of x - 1 0 resid t2 u2 resid t1 u1]
    by auto
  also have ... ≤ mtp (Suc x) t1 + mtp x t2 * max 1 (mtp 0 t1)
    using ind1 ind2
    apply simp
  by (metis Coinitial-iff-Con Ide.simps(1) dual-order.eq-iff elementary-reduction.simps(5)
    ide-char resid.simps(4) resid-Arr-Ide subs subs-implies-prfx u u1u2)
  also have ... = mtp x (λ[t1] • t2)
    by simp
  finally show mtp x ((λ[t1] • t2) \ u) ≤ mtp x (λ[t1] • t2) by blast
qed
ultimately show mtp x ((λ[t1] • t2) \ u) ≤ mtp x (λ[t1] • t2)
  using 1 by blast
qed
qed

```

Next we define the “height” of a term. This counts the number of steps in a development of maximal length of the given term.

```

fun hgt
where hgt ‡ = 0
  | hgt «-» = 0
  | hgt λ[t] = hgt t
  | hgt (t ◦ u) = hgt t + hgt u
  | hgt (λ[t] • u) = Suc (hgt t + hgt u * max 1 (mtp 0 t))

```

**lemma** *hgt-resid-ide*:

**shows**  $\llbracket \text{ide } u; u \sqsubseteq t \rrbracket \implies \text{hgt } (\text{resid } t \ u) \leq \text{hgt } t$

**by** (metis con-sym eq-imp-le resid-arr-ide prfx-implies-con subs-implies-prfx)

**lemma** *hgt-Raise*:

**shows**  $\text{hgt } (\text{Raise } l \ k \ t) = \text{hgt } t$

**using** *mtpE-eq-Raise*

**by** (induct t arbitrary: l k) auto

**lemma** *hgt-Subst*:

**shows**  $\text{Arr } u \implies \text{hgt } (\text{Subst } k \ u \ t) = \text{hgt } t + \text{hgt } u * \text{mtp } k \ t$

**proof** (induct t arbitrary: u k)

**show**  $\bigwedge u \ k. \text{Arr } u \implies \text{hgt } (\text{Subst } k \ u \ ‡) = \text{hgt } ‡ + \text{hgt } u * \text{mtp } k \ ‡$

**by** *simp*

**show**  $\bigwedge x \ u \ k. \text{Arr } u \implies \text{hgt } (\text{Subst } k \ u \ \langle x \rangle) = \text{hgt } \langle x \rangle + \text{hgt } u * \text{mtp } k \ \langle x \rangle$

**using** *hgt-Raise* **by** *auto*

**show**  $\bigwedge t \ u \ k. \llbracket \bigwedge u \ k. \text{Arr } u \implies \text{hgt } (\text{Subst } k \ u \ t) = \text{hgt } t + \text{hgt } u * \text{mtp } k \ t; \text{Arr } u \rrbracket$

$\implies \text{hgt } (\text{Subst } k \ u \ \lambda[t]) = \text{hgt } \lambda[t] + \text{hgt } u * \text{mtp } k \ \lambda[t]$

**by** *auto*

**show**  $\bigwedge t1 \ t2 \ u \ k.$

$$\begin{aligned} & \llbracket \bigwedge u k. \text{Arr } u \implies \text{hgt } (\text{Subst } k \ u \ t1) = \text{hgt } t1 + \text{hgt } u * \text{mtp } k \ t1; \\ & \bigwedge u k. \text{Arr } u \implies \text{hgt } (\text{Subst } k \ u \ t2) = \text{hgt } t2 + \text{hgt } u * \text{mtp } k \ t2; \text{Arr } u \rrbracket \\ & \implies \text{hgt } (\text{Subst } k \ u \ (t1 \circ t2)) = \text{hgt } (t1 \circ t2) + \text{hgt } u * \text{mtp } k \ (t1 \circ t2) \end{aligned}$$

**by** (*simp add: distrib-left*)

**show**  $\bigwedge t1 \ t2 \ u \ k.$

$$\begin{aligned} & \llbracket \bigwedge u k. \text{Arr } u \implies \text{hgt } (\text{Subst } k \ u \ t1) = \text{hgt } t1 + \text{hgt } u * \text{mtp } k \ t1; \\ & \bigwedge u k. \text{Arr } u \implies \text{hgt } (\text{Subst } k \ u \ t2) = \text{hgt } t2 + \text{hgt } u * \text{mtp } k \ t2; \text{Arr } u \rrbracket \\ & \implies \text{hgt } (\text{Subst } k \ u \ (\lambda[t1] \bullet t2)) = \text{hgt } (\lambda[t1] \bullet t2) + \text{hgt } u * \text{mtp } k \ (\lambda[t1] \bullet t2) \end{aligned}$$

**proof** –

**fix**  $t1 \ t2 \ u \ k$

**assume**  $ind1: \bigwedge u k. \text{Arr } u \implies \text{hgt } (\text{Subst } k \ u \ t1) = \text{hgt } t1 + \text{hgt } u * \text{mtp } k \ t1$

**assume**  $ind2: \bigwedge u k. \text{Arr } u \implies \text{hgt } (\text{Subst } k \ u \ t2) = \text{hgt } t2 + \text{hgt } u * \text{mtp } k \ t2$

**assume**  $u: \text{Arr } u$

**show**  $\text{hgt } (\text{Subst } k \ u \ (\lambda[t1] \bullet t2)) = \text{hgt } (\lambda[t1] \bullet t2) + \text{hgt } u * \text{mtp } k \ (\lambda[t1] \bullet t2)$

**proof** –

**have**  $\text{hgt } (\text{Subst } k \ u \ (\lambda[t1] \bullet t2)) =$   
 $\text{Suc } (\text{hgt } (\text{Subst } (\text{Suc } k) \ u \ t1) +$   
 $\text{hgt } (\text{Subst } k \ u \ t2) * \text{max } 1 \ (\text{mtp } 0 \ (\text{Subst } (\text{Suc } k) \ u \ t1)))$

**by** *simp*

**also have**  $\dots = \text{Suc } ((\text{hgt } t1 + \text{hgt } u * \text{mtp } (\text{Suc } k) \ t1) +$   
 $(\text{hgt } t2 + \text{hgt } u * \text{mtp } k \ t2) * \text{max } 1 \ (\text{mtp } 0 \ (\text{Subst } (\text{Suc } k) \ u \ t1)))$

**using**  $u \ ind1$  [*of u Suc k*]  $ind2$  [*of u k*] **by** *simp*

**also have**  $\dots = \text{Suc } (\text{hgt } t1 + \text{hgt } t2 * \text{max } 1 \ (\text{mtp } 0 \ (\text{Subst } (\text{Suc } k) \ u \ t1)) +$   
 $\text{hgt } u * \text{mtp } (\text{Suc } k) \ t1 +$   
 $\text{hgt } u * \text{mtp } k \ t2 * \text{max } 1 \ (\text{mtp } 0 \ (\text{Subst } (\text{Suc } k) \ u \ t1)))$

**using** *comm-semiring-class.distrib* **by** *force*

**also have**  $\dots = \text{Suc } (\text{hgt } t1 + \text{hgt } t2 * \text{max } 1 \ (\text{mtp } 0 \ (\text{Subst } (\text{Suc } k) \ u \ t1)) +$   
 $\text{hgt } u * (\text{mtp } (\text{Suc } k) \ t1 +$   
 $\text{mtp } k \ t2 * \text{max } 1 \ (\text{mtp } 0 \ (\text{Subst } (\text{Suc } k) \ u \ t1)))$

**by** (*simp add: distrib-left*)

**also have**  $\dots = \text{Suc } (\text{hgt } t1 + \text{hgt } t2 * \text{max } 1 \ (\text{mtp } 0 \ t1) +$   
 $\text{hgt } u * (\text{mtp } (\text{Suc } k) \ t1 +$   
 $\text{mtp } k \ t2 * \text{max } 1 \ (\text{mtp } 0 \ t1)))$

**proof** –

**have**  $\text{mtp } 0 \ (\text{Subst } (\text{Suc } k) \ u \ t1) = \text{mtp } 0 \ t1$

**using** *mtp<sub>0</sub>-Subst-cancel* **by** *auto*

**thus** *?thesis* **by** *simp*

**qed**

**also have**  $\dots = \text{hgt } (\lambda[t1] \bullet t2) + \text{hgt } u * \text{mtp } k \ (\lambda[t1] \bullet t2)$

**by** *simp*

**finally show** *?thesis* **by** *blast*

**qed**

**qed**

**qed**

**lemma** *elementary-reduction-decreases-hgt:*

**shows**  $\llbracket \text{elementary-reduction } u; u \sqsubseteq t \rrbracket \implies \text{hgt } (t \setminus u) < \text{hgt } t$

**proof** (*induct t arbitrary: u*)

**show**  $\bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq \# \rrbracket \implies \text{hgt } (\# \setminus u) < \text{hgt } \#$

**by simp**  
**show**  $\bigwedge u x. \llbracket \text{elementary-reduction } u; u \sqsubseteq \langle x \rangle \rrbracket \implies \text{hgt } (\langle x \rangle \setminus u) < \text{hgt } \langle x \rangle$   
**using** *Ide.simps(2) elementary-reduction-not-ide ide-backward-stable ide-char subs-implies-prfx*  
**by blast**  
**show**  $\bigwedge t u. \llbracket \bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq t \rrbracket \implies \text{hgt } (t \setminus u) < \text{hgt } t; \text{elementary-reduction } u; u \sqsubseteq \lambda[t] \rrbracket \implies \text{hgt } (\lambda[t] \setminus u) < \text{hgt } \lambda[t]$   
**proof** –  
**fix**  $t u$   
**assume**  $\text{ind}: \bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq t \rrbracket \implies \text{hgt } (t \setminus u) < \text{hgt } t$   
**assume**  $u: \text{elementary-reduction } u$   
**assume**  $\text{subs}: u \sqsubseteq \lambda[t]$   
**show**  $\text{hgt } (\lambda[t] \setminus u) < \text{hgt } \lambda[t]$   
**using**  $u \text{ subs ind}$   
**apply** (*cases u*)  
**apply** *simp-all*  
**by** *fastforce*  
**qed**  
**show**  $\bigwedge t1 t2 u. \llbracket \bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq t1 \rrbracket \implies \text{hgt } (t1 \setminus u) < \text{hgt } t1; \bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq t2 \rrbracket \implies \text{hgt } (t2 \setminus u) < \text{hgt } t2; \text{elementary-reduction } u; u \sqsubseteq t1 \circ t2 \rrbracket \implies \text{hgt } ((t1 \circ t2) \setminus u) < \text{hgt } (t1 \circ t2)$   
**proof** –  
**fix**  $t1 t2 u$   
**assume**  $\text{ind1}: \bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq t1 \rrbracket \implies \text{hgt } (t1 \setminus u) < \text{hgt } t1$   
**assume**  $\text{ind2}: \bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq t2 \rrbracket \implies \text{hgt } (t2 \setminus u) < \text{hgt } t2$   
**assume**  $u: \text{elementary-reduction } u$   
**assume**  $\text{subs}: u \sqsubseteq t1 \circ t2$   
**show**  $\text{hgt } ((t1 \circ t2) \setminus u) < \text{hgt } (t1 \circ t2)$   
**using**  $u \text{ subs ind1 ind2}$   
**apply** (*cases u*)  
**apply** *simp-all*  
**by** (*metis add-le-less-mono add-less-le-mono hgt-resid-ide ide-char not-less0 zero-less-iff-neq-zero*)  
**qed**  
**show**  $\bigwedge t1 t2 u. \llbracket \bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq t1 \rrbracket \implies \text{hgt } (t1 \setminus u) < \text{hgt } t1; \bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq t2 \rrbracket \implies \text{hgt } (t2 \setminus u) < \text{hgt } t2; \text{elementary-reduction } u; u \sqsubseteq \lambda[t1] \bullet t2 \rrbracket \implies \text{hgt } ((\lambda[t1] \bullet t2) \setminus u) < \text{hgt } (\lambda[t1] \bullet t2)$   
**proof** –  
**fix**  $t1 t2 u$   
**assume**  $\text{ind1}: \bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq t1 \rrbracket \implies \text{hgt } (t1 \setminus u) < \text{hgt } t1$   
**assume**  $\text{ind2}: \bigwedge u. \llbracket \text{elementary-reduction } u; u \sqsubseteq t2 \rrbracket \implies \text{hgt } (t2 \setminus u) < \text{hgt } t2$   
**assume**  $u: \text{elementary-reduction } u$   
**assume**  $\text{subs}: u \sqsubseteq \lambda[t1] \bullet t2$   
**have**  $\text{is-App } u \vee \text{is-Beta } u$

**using** *subs* **by** (*metis prfx-Beta-iff subs-implies-prfx*)  
**moreover have** *is-App*  $u \implies \text{hgt } ((\lambda[t1] \bullet t2) \setminus u) < \text{hgt } (\lambda[t1] \bullet t2)$   
**proof** –  
**fix**  $u1\ u2$   
**assume**  $0$ : *is-App*  $u$   
**obtain**  $u1\ u1'\ u2$  **where**  $1$ :  $u = u1 \circ u2 \wedge u1 = \lambda[u1]$   
**using**  $u\ 0$   
**by** (*metis ConD(3) Con-implies-is-Lam-iff-is-Lam Con-sym con-def is-App-def is-Lam-def null-char prfx-implies-con subs subs-implies-prfx*)  
**have**  $\text{hgt } ((\lambda[t1] \bullet t2) \setminus u) = \text{hgt } ((\lambda[t1] \bullet t2) \setminus (u1 \circ u2))$   
**using**  $1$  **by** *simp*  
**also have**  $\dots = \text{hgt } (\lambda[t1 \setminus u1'] \bullet t2 \setminus u2)$   
**by** (*metis 1 Con-sym Ide.simps(1) ide-char resid.simps(6) subs subs-implies-prfx*)  
**also have**  $\dots = \text{Suc } (\text{hgt } (t1 \setminus u1') + \text{hgt } (t2 \setminus u2) * \max (\text{Suc } 0) (\text{mtp } 0 (t1 \setminus u1')))$   
**by** *auto*  
**also have**  $\dots < \text{hgt } (\lambda[t1] \bullet t2)$   
**proof** –  
**have** *elementary-reduction*  $(\text{un-App1 } u) \wedge \text{ide } (\text{un-App2 } u) \vee$   
*ide*  $(\text{un-App1 } u) \wedge \text{elementary-reduction } (\text{un-App2 } u)$   
**using**  $u\ 1$  *elementary-reduction-App-iff* [of  $u$ ] **by** *simp*  
**moreover have** *elementary-reduction*  $(\text{un-App1 } u) \wedge \text{ide } (\text{un-App2 } u) \implies ?thesis$   
**proof** –  
**assume**  $2$ : *elementary-reduction*  $(\text{un-App1 } u) \wedge \text{ide } (\text{un-App2 } u)$   
**have** *elementary-reduction*  $u1' \wedge \text{ide } (\text{un-App2 } u)$   
**using**  $1\ 2\ u$  *elementary-reduction-Lam-iff* **by** *force*  
**moreover have**  $\text{mtp } 0 (t1 \setminus u1') \leq \text{mtp } 0\ t1$   
**using**  $1$  *calculation elementary-reduction-nonincreases-mtp subs subs.simps(4)*  
**by** *blast*  
**moreover have**  $\text{mtp } 0 (t2 \setminus u2) \leq \text{mtp } 0\ t2$   
**using**  $1$  *hgt-resid-ide* [of  $u2\ t2$ ]  
**by** (*metis calculation(1) con-sym eq-refl resid-arr-ide lambda.sel(4) prfx-implies-con subs subs.simps(4) subs-implies-prfx*)  
**ultimately show** *?thesis*  
**using**  $1\ 2$  *ind1* [of  $u1'$ ] *hgt-resid-ide*  
**apply** *simp*  
**by** (*metis 1 Suc-le-mono*  $\langle \text{mtp } 0 (t1 \setminus u1') \leq \text{mtp } 0\ t1 \rangle$  *add-less-le-mono le-add1 le-add-same-cancel1 max.mono mult-le-mono subs subs.simps(4)*)  
**qed**  
**moreover have** *ide*  $(\text{un-App1 } u) \wedge \text{elementary-reduction } (\text{un-App2 } u) \implies ?thesis$   
**proof** –  
**assume**  $2$ : *ide*  $(\text{un-App1 } u) \wedge \text{elementary-reduction } (\text{un-App2 } u)$   
**have** *ide*  $(\text{un-App1 } u) \wedge \text{elementary-reduction } u2$   
**using**  $1\ 2\ u$  *elementary-reduction-Lam-iff* **by** *force*  
**moreover have**  $\text{mtp } 0 (t1 \setminus u1') \leq \text{mtp } 0\ t1$   
**using**  $1$  *hgt-resid-ide* [of  $u1'\ t1$ ]  
**by** (*metis Ide.simps(3) calculation con-sym eq-refl ide-char resid-arr-ide lambda.sel(3) prfx-implies-con subs subs.simps(4) subs-implies-prfx*)  
**moreover have**  $\text{mtp } 0 (t2 \setminus u2) \leq \text{mtp } 0\ t2$

```

    using 1 elementary-reduction-nonincreases-mtp subs calculation(1) subs.simps(4)
    by blast
  ultimately show ?thesis
    using 1 2 ind2 [of u2]
    apply simp
    by (metis Coinitial-iff-Con Ide-iff-Src-self Nat.add-0-right add-le-less-mono
        ide-char Ide.simps(1) subs.simps(4) le-add1 max-nat.neutr-eq-iff
        mult-less-cancel2 nat.distinct(1) neq0-conv resid-Arr-Src subs
        subs-implies-prfx)

  qed
  ultimately show ?thesis by blast
  qed
  also have ... = Suc (hgt t1 + hgt t2 * max 1 (mtp 0 t1))
    by simp
  also have ... = hgt ( $\lambda[t1] \bullet t2$ )
    by simp
  finally show hgt ( $(\lambda[t1] \bullet t2) \setminus u$ ) < hgt ( $\lambda[t1] \bullet t2$ )
    by blast
  qed
  moreover have is-Beta u  $\implies$  hgt ( $(\lambda[t1] \bullet t2) \setminus u$ ) < hgt ( $\lambda[t1] \bullet t2$ )
  proof -
    fix u1 u2
    assume 0: is-Beta u
    obtain u1 u2 where 1: u =  $\lambda[u1] \bullet u2$ 
    using u 0 by (metis lambda.collapse(4))
    have hgt ( $(\lambda[t1] \bullet t2) \setminus u$ ) = hgt ( $(\lambda[t1] \bullet t2) \setminus (\lambda[u1] \bullet u2)$ )
      using 1 by simp
    also have ... = hgt (subst (resid t2 u2) (resid t1 u1))
      by (metis 1 con-def con-sym null-char prfx-implies-con resid.simps(4)
          subs subs-implies-prfx)
    also have ... = hgt (resid t1 u1) + hgt (resid t2 u2) * mtp 0 (resid t1 u1)
    proof -
      have Arr (resid t2 u2)
        by (metis 1 Coinitial-resid-resid Con-sym Ide.simps(1) ide-char resid.simps(4)
            subs subs-implies-prfx)
      thus ?thesis
        using hgt-Subst [of resid t2 u2 0 resid t1 u1] by simp
    qed
  also have ... < hgt ( $\lambda[t1] \bullet t2$ )
  proof -
    have ide u1  $\wedge$  ide u2
      using u 1 elementary-reduction-Beta-iff [of u] by auto
    thus ?thesis
      using 1 hgt-resid-ide
      by (metis add-le-mono con-sym hgt.simps(5) resid-arr-ide less-Suc-eq-le
          max.cobounded2 nat-mult-max-right prfx-implies-con subs subs.simps(5)
          subs-implies-prfx)
  qed
  finally show hgt ( $(\lambda[t1] \bullet t2) \setminus u$ ) < hgt ( $\lambda[t1] \bullet t2$ )

```

```

      by blast
    qed
  ultimately show  $\text{hgt } ((\lambda[t1] \bullet t2) \setminus u) < \text{hgt } (\lambda[t1] \bullet t2)$  by blast
  qed
qed
end

context reduction-paths
begin

```

```

lemma length-devel-le-hgt:
shows development  $t U \implies \text{length } U \leq \Lambda.\text{hgt } t$ 
  using  $\Lambda.\text{elementary-reduction-decreases-hgt}$ 
  by (induct  $U$  arbitrary:  $t$ , auto, fastforce)

```

We finally arrive at the main result of this section: the Finite Developments Theorem.

```

theorem finite-developments:
shows  $FD t$ 
  using length-devel-le-hgt [of  $t$ ]  $FD\text{-def}$  by auto

```

### 3.4.2 Complete Developments

A *complete development* is a development in which there are no residuals of originally marked redexes left to contract.

```

definition complete-development
where complete-development  $t U \equiv \text{development } t U \wedge (\Lambda.\text{Ide } t \vee [t] \text{ *}\lesssim^* U)$ 

```

```

lemma complete-development-Ide-iff:
shows complete-development  $t U \implies \Lambda.\text{Ide } t \iff U = []$ 
  using complete-development-def development-Ide  $\text{Ide.simps}(1)$  ide-char
  by (induct  $t$ ) auto

```

```

lemma complete-development-cons:
assumes complete-development  $t (u \# U)$ 
shows complete-development  $(t \setminus u) U$ 
  using assms complete-development-def
  by (metis  $\text{Ide.simps}(1)$   $\text{Ide.simps}(2)$   $\text{Resid-rec}(1)$   $\text{Resid-rec}(3)$ 
    complete-development-Ide-iff ide-char development.simps(2)
     $\Lambda.\text{ide-char list.simps}(3)$ )

```

```

lemma complete-development-cong:
shows  $[[\text{complete-development } t U; \neg \Lambda.\text{Ide } t] \implies [t] \text{ *}\sim^* U$ 
  using complete-development-def development-implies
  by (induct  $U$ ) auto

```

```

lemma complete-developments-cong:
assumes  $\neg \Lambda.\text{Ide } t$  and complete-development  $t U$  and complete-development  $t V$ 
shows  $U \text{ *}\sim^* V$ 

```

**using** *assms complete-development-cong [of t] cong-symmetric cong-transitive*  
**by** *blast*

**lemma** *Trgs-complete-development:*

**shows**  $\llbracket \text{complete-development } t \ U; \neg \Lambda.\text{Ide } t \rrbracket \implies \text{Trgs } U = \{\Lambda.\text{Trg } t\}$

**using** *complete-development-cong Ide.simps(1) Srcs-Resid Trgs.simps(2)*

*Trgs-Resid-sym ide-char complete-development-def development-imp-Arr \Lambda.targets-char \Lambda*

**apply** *simp*

**by** (*metis Srcs-Resid Trgs.simps(2) con-char ide-def*)

Now that we know all developments are finite, it is easy to construct a complete development by an iterative process that at each stage contracts one of the remaining marked redexes at each stage. It is also possible to construct a complete development by structural induction without using the finite developments property, but it is more work to prove the correctness.

**fun** (**in** *lambda-calculus*) *bottom-up-redex*

**where** *bottom-up-redex*  $\# = \#$

| *bottom-up-redex*  $\langle\langle x \rangle\rangle = \langle\langle x \rangle\rangle$

| *bottom-up-redex*  $\lambda[M] = \lambda[\text{bottom-up-redex } M]$

| *bottom-up-redex*  $(M \circ N) =$

*(if*  $\neg \text{Ide } M$  *then* *bottom-up-redex*  $M \circ \text{Src } N$  *else*  $M \circ \text{bottom-up-redex } N$  *)*

| *bottom-up-redex*  $(\lambda[M] \bullet N) =$

*(if*  $\neg \text{Ide } M$  *then*  $\lambda[\text{bottom-up-redex } M] \circ \text{Src } N$

*else if*  $\neg \text{Ide } N$  *then*  $\lambda[M] \circ \text{bottom-up-redex } N$

*else*  $\lambda[M] \bullet N$  *)*

**lemma** (**in** *lambda-calculus*) *elementary-reduction-bottom-up-redex:*

**shows**  $\llbracket \text{Arr } t; \neg \text{Ide } t \rrbracket \implies \text{elementary-reduction } (\text{bottom-up-redex } t)$

**using** *Ide-Src*

**by** (*induct t*) *auto*

**lemma** (**in** *lambda-calculus*) *subs-bottom-up-redex:*

**shows**  $\text{Arr } t \implies \text{bottom-up-redex } t \sqsubseteq t$

**apply** (*induct t*)

**apply** *auto[3]*

**apply** (*metis Arr.simps(4) Ide.simps(4) Ide-Src Ide-iff-Src-self Ide-implies-Arr*

*bottom-up-redex.simps(4) ide-char lambda.disc(14) lambda.sel(3) lambda.sel(4)*

*subs-App subs-Ide*)

**by** (*metis Arr.simps(5) Ide-Src Ide-iff-Src-self Ide-implies-Arr bottom-up-redex.simps(5)*

*ide-char subs.simps(4) subs.simps(5) subs-Ide*)

**function** (*sequential*) *bottom-up-development*

**where** *bottom-up-development*  $t =$

*(if*  $\neg \Lambda.\text{Arr } t \vee \Lambda.\text{Ide } t$  *then*  $\square$

*else*  $\Lambda.\text{bottom-up-redex } t \# (\text{bottom-up-development } (t \setminus \Lambda.\text{bottom-up-redex } t))$  *)*

**by** *pat-completeness auto*

**termination** *bottom-up-development*

**using**  $\Lambda.\text{elementary-reduction-decreases-hgt}$   $\Lambda.\text{elementary-reduction-bottom-up-redex}$

```

     $\Lambda$ .subs-bottom-up-redex
  by (relation measure  $\Lambda$ .hgt) auto

lemma complete-development-bottom-up-development-ind:
shows  $\llbracket \Lambda$ .Arr t; length (bottom-up-development t)  $\leq$  n  $\rrbracket$ 
       $\implies$  complete-development t (bottom-up-development t)
proof (induct n arbitrary: t)
  show  $\bigwedge t$ .  $\llbracket \Lambda$ .Arr t; length (bottom-up-development t)  $\leq$  0  $\rrbracket$ 
       $\implies$  complete-development t (bottom-up-development t)
    using complete-development-def development-Ide by auto
  show  $\bigwedge n t$ .  $\llbracket \bigwedge t$ .  $\llbracket \Lambda$ .Arr t; length (bottom-up-development t)  $\leq$  n  $\rrbracket$ 
       $\implies$  complete-development t (bottom-up-development t);
       $\Lambda$ .Arr t; length (bottom-up-development t)  $\leq$  Suc n  $\rrbracket$ 
       $\implies$  complete-development t (bottom-up-development t)
proof -
  fix n t
  assume t:  $\Lambda$ .Arr t
  assume n: length (bottom-up-development t)  $\leq$  Suc n
  assume ind:  $\bigwedge t$ .  $\llbracket \Lambda$ .Arr t; length (bottom-up-development t)  $\leq$  n  $\rrbracket$ 
       $\implies$  complete-development t (bottom-up-development t)
  show complete-development t (bottom-up-development t)
  proof (cases bottom-up-development t)
    show bottom-up-development t = []  $\implies$  ?thesis
      using ind t by force
    fix u U
    assume uU: bottom-up-development t = u # U
    have 1:  $\Lambda$ .elementary-reduction u  $\wedge$  u  $\sqsubseteq$  t
      using t uU
      by (metis bottom-up-development.simps  $\Lambda$ .elementary-reduction-bottom-up-redex
          list.inject list.simps(3)  $\Lambda$ .subs-bottom-up-redex)
    moreover have complete-development ( $\Lambda$ .resid t u) U
      using 1 ind
      by (metis Suc-le-length-iff  $\Lambda$ .arr-char  $\Lambda$ .arr-resid-iff-con bottom-up-development.simps
          list.discI list.inject n not-less-eq-eq  $\Lambda$ .prfx-implies-con
           $\Lambda$ .con-sym  $\Lambda$ .subs-implies-prfx uU)
    ultimately show ?thesis
      by (metis Con-sym Ide.simps(2) Resid-rec(1) Resid-rec(3)
          complete-development-Ide-iff complete-development-def ide-char
          development.simps(2) development-implies  $\Lambda$ .ide-char list.simps(3) uU)
  qed
qed
qed

```

lemma complete-development-bottom-up-development:  
 assumes  $\Lambda$ .Arr t  
 shows complete-development t (bottom-up-development t)  
 using assms complete-development-bottom-up-development-ind by blast

end

### 3.5 Reduction Strategies

**context** *lambda-calculus*  
**begin**

A *reduction strategy* is a function taking an identity term to an arrow having that identity as its source.

**definition** *reduction-strategy*  
**where** *reduction-strategy*  $f \longleftrightarrow (\forall t. \text{Ide } t \longrightarrow \text{Cinitial } (f t) t)$

The following defines the iterated application of a reduction strategy to an identity term.

**fun** *reduce*  
**where** *reduce*  $f a 0 = a$   
| *reduce*  $f a (\text{Suc } n) = \text{reduce } f (\text{Trg } (f a)) n$

**lemma** *red-reduce*:  
**assumes** *reduction-strategy*  $f$   
**shows**  $\text{Ide } a \implies \text{red } a (\text{reduce } f a n)$   
**apply** (*induct*  $n$  *arbitrary*:  $a, \text{auto}$ )  
**apply** (*metis* *Ide-iff-Src-self* *Ide-iff-Trg-self* *Ide-implies-Arr* *red.simps*)  
**by** (*metis* *Ide-Trg* *Ide-iff-Src-self* *assms* *red.intros(1)* *red.intros(2)* *reduction-strategy-def*)

A reduction strategy is *normalizing* if iterated application of it to a normalizable term eventually yields a normal form.

**definition** *normalizing-strategy*  
**where** *normalizing-strategy*  $f \longleftrightarrow (\forall a. \text{normalizable } a \longrightarrow (\exists n. \text{NF } (\text{reduce } f a n)))$

**end**

**context** *reduction-paths*  
**begin**

The following function constructs the reduction path that results by iterating the application of a reduction strategy to a term.

**fun** *apply-strategy*  
**where** *apply-strategy*  $f a 0 = []$   
| *apply-strategy*  $f a (\text{Suc } n) = f a \# \text{apply-strategy } f (\Lambda.\text{Trg } (f a)) n$

**lemma** *apply-strategy-gives-path-ind*:  
**assumes**  $\Lambda.\text{reduction-strategy } f$   
**shows**  $[\Lambda.\text{Ide } a; n > 0] \implies \text{Arr } (\text{apply-strategy } f a n) \wedge$   
 $\text{length } (\text{apply-strategy } f a n) = n \wedge$   
 $\text{Src } (\text{apply-strategy } f a n) = a \wedge$   
 $\text{Trg } (\text{apply-strategy } f a n) = \Lambda.\text{reduce } f a n$

**proof** (*induct*  $n$  *arbitrary*:  $a, \text{simp}$ )  
**fix**  $n a$   
**assume** *ind*:  $\bigwedge a. [\Lambda.\text{Ide } a; 0 < n] \implies \text{Arr } (\text{apply-strategy } f a n) \wedge$   
 $\text{length } (\text{apply-strategy } f a n) = n \wedge$

$$\begin{aligned} \text{Src } (\text{apply-strategy } f \ a \ n) &= a \wedge \\ \text{Trg } (\text{apply-strategy } f \ a \ n) &= \Lambda.\text{reduce } f \ a \ n \end{aligned}$$

**assume**  $a: \Lambda.\text{Ide } a$   
**show**  $\text{Arr } (\text{apply-strategy } f \ a \ (\text{Suc } n)) \wedge$   
 $\text{length } (\text{apply-strategy } f \ a \ (\text{Suc } n)) = \text{Suc } n \wedge$   
 $\text{Src } (\text{apply-strategy } f \ a \ (\text{Suc } n)) = a \wedge$   
 $\text{Trg } (\text{apply-strategy } f \ a \ (\text{Suc } n)) = \Lambda.\text{reduce } f \ a \ (\text{Suc } n)$   
**proof** (*intro conjI*)  
**have**  $1: \Lambda.\text{Arr } (f \ a) \wedge \Lambda.\text{Src } (f \ a) = a$   
**using** *assms a  $\Lambda$ .reduction-strategy-def*  
**by** (*metis  $\Lambda$ .Ide-iff-Src-self*)  
**show**  $\text{Arr } (\text{apply-strategy } f \ a \ (\text{Suc } n))$   
**using**  $1 \text{ Arr.elims}(3) \text{ ind } \Lambda.\text{targets-char}_\Lambda \ \Lambda.\text{Ide-Trg}$  **by** *fastforce*  
**show**  $\text{Src } (\text{apply-strategy } f \ a \ (\text{Suc } n)) = a$   
**by** (*simp add: 1*)  
**show**  $\text{length } (\text{apply-strategy } f \ a \ (\text{Suc } n)) = \text{Suc } n$   
**by** (*metis 1  $\Lambda$ .Ide-Trg One-nat-def Suc-eq-plus1 ind list.size(3) list.size(4)*  
*neq0-conv apply-strategy.simps(1) apply-strategy.simps(2)*)  
**show**  $\text{Trg } (\text{apply-strategy } f \ a \ (\text{Suc } n)) = \Lambda.\text{reduce } f \ a \ (\text{Suc } n)$   
**proof** (*cases apply-strategy f ( $\Lambda$ .Trg (f a)) n = []*)  
**show**  $\text{apply-strategy } f \ (\Lambda.\text{Trg } (f \ a)) \ n = [] \implies ?thesis$   
**using**  $a \ 1 \text{ ind [of } \Lambda.\text{Trg } (f \ a)] \ \Lambda.\text{Ide-Trg } \Lambda.\text{targets-char}_\Lambda$  **by** *force*  
**assume**  $2: \text{apply-strategy } f \ (\Lambda.\text{Trg } (f \ a)) \ n \neq []$   
**have**  $\text{Trg } (\text{apply-strategy } f \ a \ (\text{Suc } n)) = \text{Trg } (\text{apply-strategy } f \ (\Lambda.\text{Trg } (f \ a)) \ n)$   
**using**  $a \ 1 \text{ ind [of } \Lambda.\text{Trg } (f \ a)]$   
**by** (*simp add: 2*)  
**also have**  $\dots = \Lambda.\text{reduce } f \ a \ (\text{Suc } n)$   
**using**  $1 \ 2 \ \Lambda.\text{Ide-Trg ind [of } \Lambda.\text{Trg } (f \ a)]$  **by** *fastforce*  
**finally show** *?thesis* **by** *blast*  
**qed**  
**qed**  
**qed**

**lemma** *apply-strategy-gives-path*:  
**assumes**  $\Lambda.\text{reduction-strategy } f$  **and**  $\Lambda.\text{Ide } a$  **and**  $n > 0$   
**shows**  $\text{Arr } (\text{apply-strategy } f \ a \ n)$   
**and**  $\text{length } (\text{apply-strategy } f \ a \ n) = n$   
**and**  $\text{Src } (\text{apply-strategy } f \ a \ n) = a$   
**and**  $\text{Trg } (\text{apply-strategy } f \ a \ n) = \Lambda.\text{reduce } f \ a \ n$   
**using** *assms apply-strategy-gives-path-ind* **by** *auto*

**lemma** *reduce-eq-Trg-apply-strategy*:  
**assumes**  $\Lambda.\text{reduction-strategy } S$  **and**  $\Lambda.\text{Ide } a$   
**shows**  $n > 0 \implies \Lambda.\text{reduce } S \ a \ n = \text{Trg } (\text{apply-strategy } S \ a \ n)$   
**using** *assms*  
**apply** (*induct n*)  
**apply** *simp-all*  
**by** (*metis Arr.simps(1) Trg-simp apply-strategy-gives-path-ind  $\Lambda$ .Ide-Trg*  
 $\Lambda.\text{reduce.simps}(1) \ \Lambda.\text{reduction-strategy-def } \Lambda.\text{trg-char neq0-conv}$ )

*apply-strategy.simps(1)*)

end

### 3.5.1 Parallel Reduction

**context** *lambda-calculus*

**begin**

*Parallel reduction* is the strategy that contracts all available redexes at each step.

**fun** *parallel-strategy*

**where** *parallel-strategy* «*i*» = «*i*»

| *parallel-strategy*  $\lambda[t] = \lambda[\textit{parallel-strategy } t]$

| *parallel-strategy*  $(\lambda[t] \circ u) = \lambda[\textit{parallel-strategy } t] \bullet \textit{parallel-strategy } u$

| *parallel-strategy*  $(t \circ u) = \textit{parallel-strategy } t \circ \textit{parallel-strategy } u$

| *parallel-strategy*  $(\lambda[t] \bullet u) = \lambda[\textit{parallel-strategy } t] \bullet \textit{parallel-strategy } u$

| *parallel-strategy*  $\# = \#$

**lemma** *parallel-strategy-is-reduction-strategy*:

**shows** *reduction-strategy parallel-strategy*

**proof** (*unfold reduction-strategy-def, intro allI impI*)

**fix** *t*

**show** *Ide t*  $\implies$  *Coinitial (parallel-strategy t) t*

**using** *Ide-implies-Arr*

**apply** (*induct t, auto*)

**by** *force+*

**qed**

**lemma** *parallel-strategy-Src-eq*:

**shows** *Arr t*  $\implies$  *parallel-strategy (Src t) = parallel-strategy t*

**by** (*induct t*) *auto*

**lemma** *subs-parallel-strategy-Src*:

**shows** *Arr t*  $\implies$  *t*  $\sqsubseteq$  *parallel-strategy (Src t)*

**by** (*induct t*) *auto*

end

**context** *reduction-paths*

**begin**

Parallel reduction is a universal strategy in the sense that every reduction path is  $^*\lesssim^*$ -below the path generated by the parallel reduction strategy.

**lemma** *parallel-strategy-is-universal*:

**shows**  $\llbracket n > 0; n \leq \textit{length } U; \textit{Arr } U \rrbracket$

$\implies$  *take n U*  $^*\lesssim^*$  *apply-strategy*  $\Lambda.\textit{parallel-strategy (Src U) n}$

**proof** (*induct n arbitrary: U, simp*)

**fix** *n a* **and** *U*  $:: \Lambda.\textit{lambda list}$

**assume** *n*: *Suc n*  $\leq$  *length U*

**assume**  $U: \text{Arr } U$   
**assume**  $\text{ind}: \bigwedge U. \llbracket 0 < n; n \leq \text{length } U; \text{Arr } U \rrbracket$   
 $\implies \text{take } n \ U \ * \lesssim^* \ \text{apply-strategy } \Lambda.\text{parallel-strategy } (\text{Src } U) \ n$   
**have** 1:  $\text{take } (\text{Suc } n) \ U = \text{hd } U \ \# \ \text{take } n \ (\text{tl } U)$   
**by** (*metis*  $U \ \text{Arr.simps}(1) \ \text{take-Suc}$ )  
**have** 2:  $\text{hd } U \sqsubseteq \Lambda.\text{parallel-strategy } (\text{Src } U)$   
**by** (*metis*  $\text{Arr-imp-arr-hd} \ \text{Con-single-ideI}(2) \ \text{Resid-Arr-Src} \ \text{Src-resid} \ \text{Srcs-simp}_{\Lambda P}$   
 $\text{Trg.simps}(2) \ U \ \Lambda.\text{source-is-ide} \ \Lambda.\text{trg-ide} \ \text{empty-set} \ \Lambda.\text{arr-char} \ \Lambda.\text{sources-char}_{\Lambda}$   
 $\Lambda.\text{subs-parallel-strategy-Src} \ \text{list.set-intros}(1) \ \text{list.simps}(15)$ )  
**show**  $\text{take } (\text{Suc } n) \ U \ * \lesssim^* \ \text{apply-strategy } \Lambda.\text{parallel-strategy } (\text{Src } U) \ (\text{Suc } n)$   
**proof** (*cases*  $\text{apply-strategy } \Lambda.\text{parallel-strategy } (\text{Src } U) \ (\text{Suc } n)$ )  
**show**  $\text{apply-strategy } \Lambda.\text{parallel-strategy } (\text{Src } U) \ (\text{Suc } n) = [] \implies$   
 $\text{take } (\text{Suc } n) \ U \ * \lesssim^* \ \text{apply-strategy } \Lambda.\text{parallel-strategy } (\text{Src } U) \ (\text{Suc } n)$   
**by** *simp*  
**fix**  $v \ V$   
**assume** 3:  $\text{apply-strategy } \Lambda.\text{parallel-strategy } (\text{Src } U) \ (\text{Suc } n) = v \ \# \ V$   
**show**  $\text{take } (\text{Suc } n) \ U \ * \lesssim^* \ \text{apply-strategy } \Lambda.\text{parallel-strategy } (\text{Src } U) \ (\text{Suc } n)$   
**proof** (*cases*  $V = []$ )  
**show**  $V = [] \implies ?thesis$   
**using** 1 2 3 *ind ide-char*  
**by** (*metis*  $\text{Suc-inject} \ \text{Ide.simps}(2) \ \text{Resid.simps}(3) \ \text{list.discI} \ \text{list.inject}$   
 $\Lambda.\text{prfx-implies-con} \ \text{apply-strategy.elims} \ \Lambda.\text{subs-implies-prfx} \ \text{take0}$ )  
**assume**  $V: V \neq []$   
**have** 4:  $\text{Arr } (v \ \# \ V)$   
**using** 3 *apply-strategy-gives-path(1)*  
**by** (*metis*  $\text{Arr-imp-arr-hd} \ \text{Srcs-simp}_{PWE} \ \text{Srcs-simp}_{\Lambda P} \ U \ \Lambda.\text{Ide-Src} \ \Lambda.\text{arr-iff-has-target}$   
 $\Lambda.\text{parallel-strategy-is-reduction-strategy} \ \Lambda.\text{targets-char}_{\Lambda} \ \text{singleton-insert-inj-eq}'$   
 $\text{zero-less-Suc}$ )  
**have** 5:  $\text{Arr } (\text{hd } U \ \# \ \text{take } n \ (\text{tl } U))$   
**by** (*metis* 1  $U \ \text{Arr-append-iff}_P \ \text{id-take-nth-drop} \ \text{list.discI} \ \text{not-less} \ \text{take-all-iff}$ )  
**have** 6:  $\text{Srcs } (\text{hd } U \ \# \ \text{take } n \ (\text{tl } U)) = \text{Srcs } (v \ \# \ V)$   
**by** (*metis* 2 3  $\Lambda.\text{Cointial-iff-Con} \ \Lambda.\text{Ide.simps}(1) \ \text{Srcs.simps}(2) \ \text{Srcs.simps}(3)$   
 $\Lambda.\text{ide-char} \ \text{list.exhaust-sel} \ \text{list.inject} \ \text{apply-strategy.simps}(2) \ \Lambda.\text{sources-char}_{\Lambda}$   
 $\Lambda.\text{subs-implies-prfx}$ )  
**have**  $\text{take } (\text{Suc } n) \ U \ * \setminus^* \ \text{apply-strategy } \Lambda.\text{parallel-strategy } (\text{Src } U) \ (\text{Suc } n) =$   
 $[\text{hd } U \ \setminus \ v] \ * \setminus^* \ V \ @ \ (\text{take } n \ (\text{tl } U) \ * \setminus^* \ [v \ \setminus \ \text{hd } U]) \ * \setminus^* \ (V \ * \setminus^* \ [\text{hd } U \ \setminus \ v])$   
**using**  $U \ V \ 1 \ 3 \ 4 \ 5 \ 6$   
**by** (*metis*  $\text{Resid.simps}(1) \ \text{Resid-cons}(1) \ \text{Resid-rec}(3-4) \ \text{confluence-ind}$ )  
**moreover** **have**  $\text{Ide } \dots$   
**proof**  
**have** 7:  $v = \Lambda.\text{parallel-strategy } (\text{Src } U) \ \wedge$   
 $V = \text{apply-strategy } \Lambda.\text{parallel-strategy } (\text{Src } U \ \setminus \ v) \ n$   
**using** 3  $\Lambda.\text{subs-implies-prfx} \ \Lambda.\text{subs-parallel-strategy-Src}$   
**apply** *simp*  
**by** (*metis* (*full-types*)  $\Lambda.\text{Cointial-iff-Con} \ \Lambda.\text{Ide.simps}(1) \ \Lambda.\text{Trg.simps}(5)$   
 $\Lambda.\text{parallel-strategy.simps}(9) \ \Lambda.\text{resid-Src-Arr}$ )  
**show** 8:  $\text{Ide } ([\text{hd } U \ \setminus \ v] \ * \setminus^* \ V)$   
**by** (*metis* 2 4 5 6 7  $V \ \text{Con-initial-left} \ \text{Ide.simps}(2)$   
 $\text{confluence-ind} \ \text{Con-rec}(3) \ \text{Resid-Ide-Arr-ind} \ \Lambda.\text{subs-implies-prfx}$ )

**show** 9:  $Ide ((take\ n\ (tl\ U)\ *\ \ [v\ \ hd\ U])\ *\ \ (V\ *\ \ [hd\ U\ \ v]))$   
**proof** –  
**have** 10:  $\Lambda.Ide\ (hd\ U\ \ v)$   
**using** 2 7  $\Lambda.ide-char\ \Lambda.subs-implies-prfx$  **by** *presburger*  
**have** 11:  $V = apply-strategy\ \Lambda.parallel-strategy\ (\Lambda.Trg\ v)\ n$   
**using** 3 **by** *auto*  
**have**  $(take\ n\ (tl\ U)\ *\ \ [v\ \ hd\ U])\ *\ \ (V\ *\ \ [hd\ U\ \ v]) =$   
 $(take\ n\ (tl\ U)\ *\ \ [v\ \ hd\ U])\ *\ \$   
 $apply-strategy\ \Lambda.parallel-strategy\ (\Lambda.Trg\ v)\ n$   
**by** (*metis* 8 10 11 *Ide.simps(1)* *Resid-single-ide(2)*  $\Lambda.prfx-char$ )  
**moreover** **have** *Ide ...*  
**proof** –  
**have** *Ide*  $(take\ n\ (take\ n\ (tl\ U)\ *\ \ [v\ \ hd\ U])\ *\ \$   
 $apply-strategy\ \Lambda.parallel-strategy\ (\Lambda.Trg\ v)\ n)$   
**proof** –  
**have**  $0 < n$   
**proof** –  
**have**  $length\ V = n$   
**using** *apply-strategy-gives-path*  
**by** (*metis* 10 11  $V\ \Lambda.Coinitial-iff-Con\ \Lambda.Ide-Trg\ \Lambda.Arr-not-Nil$   
 $\Lambda.Ide-implies-Arr\ \Lambda.parallel-strategy-is-reduction-strategy\ neq0-conv$   
*apply-strategy.simps(1)*)  
**thus** *?thesis*  
**using**  $V$  **by** *blast*  
**qed**  
**moreover** **have**  $n \leq length\ (take\ n\ (tl\ U)\ *\ \ [v\ \ hd\ U])$   
**proof** –  
**have**  $length\ (take\ n\ (tl\ U)) = n$   
**using**  $n$  **by** *force*  
**thus** *?thesis*  
**using**  $n\ U\ length-Resid$  [of  $take\ n\ (tl\ U)\ [v\ \ hd\ U]$ ]  
**by** (*metis* 4 5 6 *Arr.simps(1)* *Con-cons(2)* *Con-rec(2)*  
*confluence-ind\ dual-order.eq-iff*)  
**qed**  
**moreover** **have**  $\Lambda.Trg\ v = Src\ (take\ n\ (tl\ U)\ *\ \ [v\ \ hd\ U])$   
**proof** –  
**have**  $Src\ (take\ n\ (tl\ U)\ *\ \ [v\ \ hd\ U]) = Trg\ [v\ \ hd\ U]$   
**by** (*metis* *Src-resid\ calculation(1-2)* *linorder-not-less\ list.size(3)*)  
**also** **have**  $\dots = \Lambda.Trg\ v$   
**by** (*metis* 10 *Trg.simps(2)*  $\Lambda.Arr-not-Nil\ \Lambda.apex-sym\ \Lambda.trg-ide$   
 $\Lambda.Ide-iff-Src-self\ \Lambda.Ide-implies-Arr\ \Lambda.Src-resid\ \Lambda.prfx-char$ )  
**finally** **show** *?thesis* **by** *simp*  
**qed**  
**ultimately** **show** *?thesis*  
**using** *ind* [of *Resid*  $(take\ n\ (tl\ U))\ [\Lambda.resid\ v\ (hd\ U)]$ ] *ide-char*  
**by** (*metis* *Con-imp-Arr-Resid\ le-zero-eq\ less-not-refl\ list.size(3)*)  
**qed**  
**moreover** **have**  $take\ n\ (take\ n\ (tl\ U)\ *\ \ [v\ \ hd\ U]) =$   
 $take\ n\ (tl\ U)\ *\ \ [v\ \ hd\ U]$

```

proof –
  have Arr (take n (tl U) *\* [v \ hd U])
    by (metis Con-imp-Arr-Resid Con-implies-Arr(1) Ide.simps(1) calculation
      take-Nil)
  thus ?thesis
    by (metis 1 Arr.simps(1) length-Resid dual-order.eq-iff length-Cons
      length-take min.absorb2 n old.nat.inject take-all)
  qed
  ultimately show ?thesis by simp
  qed
  ultimately show ?thesis by auto
  qed
  show Trg ([hd U \ v] *\* V) =
    Src ((take n (tl U) *\* [v \ hd U]) *\* (V *\* [hd U \ v]))
    by (metis 9 Ide.simps(1) Src-resid Trg-resid-sym)
  qed
  ultimately show ?thesis
    using ide-char by presburger
  qed
  qed
  qed
end

context lambda-calculus
begin

  Parallel reduction is a normalizing strategy.

  lemma parallel-strategy-is-normalizing:
  shows normalizing-strategy parallel-strategy
  proof –
    interpret  $\Lambda x$ : reduction-paths .

    have  $\bigwedge a$ . normalizable a  $\implies \exists n$ . NF (reduce parallel-strategy a n)
    proof –
      fix a
      assume 1: normalizable a
      obtain U b where U:  $\Lambda x$ .Arr U  $\wedge$   $\Lambda x$ .Src U = a  $\wedge$   $\Lambda x$ .Trg U = b  $\wedge$  NF b
        using 1 normalizable-def  $\Lambda x$ .red-iff by blast
      have 2:  $\bigwedge n$ .  $[0 < n; n \leq \text{length } U]$ 
         $\implies \Lambda x$ .Ide ( $\Lambda x$ .Resid (take n U) ( $\Lambda x$ .apply-strategy parallel-strategy a n))
        using U  $\Lambda x$ .parallel-strategy-is-universal  $\Lambda x$ .ide-char by blast
      let ?PR =  $\Lambda x$ .apply-strategy parallel-strategy a (length U)
      have  $\Lambda x$ .Trg ?PR = b
      proof –
        have 3:  $\Lambda x$ .Ide ( $\Lambda x$ .Resid U ?PR)
          using U 2 [of length U] by force
        have  $\Lambda x$ .Trg ( $\Lambda x$ .Resid ?PR U) = b
          by (metis 3 NF-reduct-is-trivial U  $\Lambda x$ .Con-imp-Arr-Resid  $\Lambda x$ .Con-sym  $\Lambda x$ .Ide.simps(1))
    
```

```

       $\Lambda x.$ Src-resid reduction-paths.red-iff)
thus ?thesis
      by (metis 3  $\Lambda x.$ Con-Arr-self  $\Lambda x.$ Ide-implies-Arr  $\Lambda x.$ Resid-Arr-Ide-ind
           $\Lambda x.$ Src-resid  $\Lambda x.$ Trg-resid-sym)
qed
hence reduce parallel-strategy a (length U) = b
      using 1 U
      by (metis  $\Lambda x.$ Arr.simps(1) length-greater-0-conv normalizable-def
           $\Lambda x.$ apply-strategy-gives-path(4) parallel-strategy-is-reduction-strategy)
thus  $\exists n.$  NF (reduce parallel-strategy a n)
      using U by blast
qed
thus ?thesis
      using normalizing-strategy-def by blast
qed

```

An alternative characterization of a normal form is a term on which the parallel reduction strategy yields an identity.

**abbreviation** has-redex  
**where** has-redex  $t \equiv \text{Arr } t \wedge \neg \text{Ide (parallel-strategy } t)$

**lemma** NF-iff-has-no-redex:

**shows**  $\text{Arr } t \implies \text{NF } t \longleftrightarrow \neg \text{has-redex } t$

**proof** (induct t)

**show**  $\text{Arr } \# \implies \text{NF } \# \longleftrightarrow \neg \text{has-redex } \#$

**using** NF-def **by** simp

**show**  $\bigwedge x. \text{Arr } \langle\langle x \rangle\rangle \implies \text{NF } \langle\langle x \rangle\rangle \longleftrightarrow \neg \text{has-redex } \langle\langle x \rangle\rangle$

**using** NF-def **by** force

**show**  $\bigwedge t. [\text{Arr } t \implies \text{NF } t \longleftrightarrow \neg \text{has-redex } t; \text{Arr } \lambda[t]] \implies \text{NF } \lambda[t] \longleftrightarrow \neg \text{has-redex } \lambda[t]$

**proof** –

**fix** t

**assume** ind:  $\text{Arr } t \implies \text{NF } t \longleftrightarrow \neg \text{has-redex } t$

**assume** t:  $\text{Arr } \lambda[t]$

**show**  $\text{NF } \lambda[t] \longleftrightarrow \neg \text{has-redex } \lambda[t]$

**proof**

**show**  $\text{NF } \lambda[t] \implies \neg \text{has-redex } \lambda[t]$

**using** t ind

**by** (metis NF-def Arr.simps(3) Ide.simps(3) Src.simps(3) parallel-strategy.simps(2))

**show**  $\neg \text{has-redex } \lambda[t] \implies \text{NF } \lambda[t]$

**using** t ind

**by** (metis NF-def ide-backward-stable ide-char parallel-strategy-Src-eq  
subs-implies-prfx subs-parallel-strategy-Src)

**qed**

**qed**

**show**  $\bigwedge t1 t2. [\text{Arr } t1 \implies \text{NF } t1 \longleftrightarrow \neg \text{has-redex } t1;$

$\text{Arr } t2 \implies \text{NF } t2 \longleftrightarrow \neg \text{has-redex } t2;$

$\text{Arr } (\lambda[t1] \bullet t2)]$

$\implies \text{NF } (\lambda[t1] \bullet t2) \longleftrightarrow \neg \text{has-redex } (\lambda[t1] \bullet t2)$

**using** NF-def Ide.simps(5) parallel-strategy.simps(8) **by** presburger

**show**  $\bigwedge t1\ t2. \llbracket \text{Arr } t1 \implies \text{NF } t1 \longleftrightarrow \neg \text{has-redex } t1;$   
 $\text{Arr } t2 \implies \text{NF } t2 \longleftrightarrow \neg \text{has-redex } t2;$   
 $\text{Arr } (t1 \circ t2) \rrbracket$   
 $\implies \text{NF } (t1 \circ t2) \longleftrightarrow \neg \text{has-redex } (t1 \circ t2)$

**proof** –

**fix**  $t1\ t2$

**assume**  $ind1: \text{Arr } t1 \implies \text{NF } t1 \longleftrightarrow \neg \text{has-redex } t1$

**assume**  $ind2: \text{Arr } t2 \implies \text{NF } t2 \longleftrightarrow \neg \text{has-redex } t2$

**assume**  $t: \text{Arr } (t1 \circ t2)$

**show**  $\text{NF } (t1 \circ t2) \longleftrightarrow \neg \text{has-redex } (t1 \circ t2)$

**using**  $t\ ind1\ ind2\ \text{NF-def}$

**apply** ( $\text{intro } \text{iffI}$ )

**apply** ( $\text{metis } \text{Ide-iff-Src-self } \text{parallel-strategy-is-reduction-strategy}$   
 $\text{reduction-strategy-def}$ )

**apply** ( $\text{cases } t1$ )

**apply**  $\text{simp-all}$

**apply** ( $\text{metis } \text{Ide-iff-Src-self } \text{ide-char } \text{parallel-strategy.simps}(1,5)$   
 $\text{parallel-strategy-is-reduction-strategy } \text{reduction-strategy-def } \text{resid-Arr-Src}$   
 $\text{subs-implies-prfx } \text{subs-parallel-strategy-Src}$ )

**by** ( $\text{metis } \text{Ide-iff-Src-self } \text{ide-char } ind1\ \text{Arr.simps}(4)\ \text{parallel-strategy.simps}(6)$   
 $\text{parallel-strategy-is-reduction-strategy } \text{reduction-strategy-def } \text{resid-Arr-Src}$   
 $\text{subs-implies-prfx } \text{subs-parallel-strategy-Src}$ )

**qed**

**qed**

**lemma** (**in**  $\text{lambda-calculus}$ )  $\text{not-NF-elim}$ :

**assumes**  $\neg \text{NF } t$  **and**  $\text{Ide } t$

**obtains**  $u$  **where**  $\text{coinitial } t\ u \wedge \neg \text{Ide } u$

**using**  $\text{assms } \text{NF-def}$  **by**  $\text{auto}$

**lemma** (**in**  $\text{lambda-calculus}$ )  $\text{NF-Lam-iff}$ :

**shows**  $\text{NF } \lambda[t] \longleftrightarrow \text{NF } t$

**using**  $\text{NF-def}$

**by** ( $\text{metis } \text{Ide-implies-Arr } \text{NF-iff-has-no-redex } \text{Ide.simps}(3)\ \text{parallel-strategy.simps}(2)$ )

**lemma** (**in**  $\text{lambda-calculus}$ )  $\text{NF-App-iff}$ :

**shows**  $\text{NF } (t1 \circ t2) \longleftrightarrow \neg \text{is-Lam } t1 \wedge \text{NF } t1 \wedge \text{NF } t2$

**proof** –

**have**  $\neg \text{NF } (t1 \circ t2) \implies \text{is-Lam } t1 \vee \neg \text{NF } t1 \vee \neg \text{NF } t2$

**apply** ( $\text{cases } \text{is-Lam } t1$ )

**apply**  $\text{simp-all}$

**apply** ( $\text{cases } t1$ )

**apply**  $\text{simp-all}$

**using**  $\text{NF-def } \text{Ide.simps}(1)$  **apply**  $\text{presburger}$

**apply** ( $\text{metis } \text{Ide-implies-Arr } \text{NF-def } \text{NF-iff-has-no-redex } \text{Ide.simps}(4)$   
 $\text{parallel-strategy.simps}(5)$ )

**apply** ( $\text{metis } \text{Ide-implies-Arr } \text{NF-def } \text{NF-iff-has-no-redex } \text{Ide.simps}(4)$   
 $\text{parallel-strategy.simps}(6)$ )

**using**  $\text{NF-def } \text{Ide.simps}(5)$  **by**  $\text{presburger}$

```

moreover have is-Lam t1  $\vee \neg NF$  t1  $\vee \neg NF$  t2  $\implies \neg NF$  (t1  $\circ$  t2)
proof –
  have is-Lam t1  $\implies \neg NF$  (t1  $\circ$  t2)
    by (metis Ide-implies-Arr NF-def NF-iff-has-no-redex Ide.simps(5) lambda.collapse(2)
      parallel-strategy.simps(3,8))
  moreover have  $\neg NF$  t1  $\implies \neg NF$  (t1  $\circ$  t2)
    using NF-def Ide-iff-Src-self Ide-implies-Arr
    apply auto
    by (metis (full-types) Arr.simps(4) Ide.simps(4) Src.simps(4))
  moreover have  $\neg NF$  t2  $\implies \neg NF$  (t1  $\circ$  t2)
    using NF-def Ide-iff-Src-self Ide-implies-Arr
    apply auto
    by (metis (full-types) Arr.simps(4) Ide.simps(4) Src.simps(4))
  ultimately show is-Lam t1  $\vee \neg NF$  t1  $\vee \neg NF$  t2  $\implies \neg NF$  (t1  $\circ$  t2)
    by auto
qed
ultimately show ?thesis by blast
qed

```

### 3.5.2 Head Reduction

*Head reduction* is the strategy that only contracts a redex at the “head” position, which is found at the end of the “left spine” of applications, and does nothing if there is no such redex.

The following function applies to an arbitrary arrow  $t$ , and it marks the redex at the head position, if any, otherwise it yields *Src*  $t$ .

```

fun head-strategy
where head-strategy «i» = «i»
  | head-strategy  $\lambda[t]$  =  $\lambda[\text{head-strategy } t]$ 
  | head-strategy ( $\lambda[t] \circ u$ ) =  $\lambda[\text{Src } t] \bullet \text{Src } u$ 
  | head-strategy (t  $\circ$  u) = head-strategy t  $\circ$  Src u
  | head-strategy ( $\lambda[t] \bullet u$ ) =  $\lambda[\text{Src } t] \bullet \text{Src } u$ 
  | head-strategy  $\#$  =  $\#$ 

```

```

lemma Arr-head-strategy:
shows Arr t  $\implies$  Arr (head-strategy t)
  apply (induct t)
  apply auto
proof –
  fix t u
  assume ind: Arr (head-strategy t)
  assume t: Arr t and u: Arr u
  show Arr (head-strategy (t  $\circ$  u))
    using t u ind
    by (cases t) auto
qed

```

```

lemma Src-head-strategy:

```

```

shows  $Arr\ t \implies Src\ (head\text{-}strategy\ t) = Src\ t$ 
  apply (induct t)
    apply auto
proof –
  fix  $t\ u$ 
  assume  $ind: Src\ (head\text{-}strategy\ t) = Src\ t$ 
  assume  $t: Arr\ t$  and  $u: Arr\ u$ 
  have  $Src\ (head\text{-}strategy\ (t\ \circ\ u)) = Src\ (head\text{-}strategy\ t\ \circ\ Src\ u)$ 
    using  $t\ ind$ 
    by (cases t) auto
  also have  $\dots = Src\ t\ \circ\ Src\ u$ 
    using  $t\ u\ ind$  by auto
  finally show  $Src\ (head\text{-}strategy\ (t\ \circ\ u)) = Src\ t\ \circ\ Src\ u$  by simp
qed

```

```

lemma Con-head-strategy:
shows  $Arr\ t \implies Con\ t\ (head\text{-}strategy\ t)$ 
  apply (induct t)
    apply auto
    apply (simp add: Arr-head-strategy Src-head-strategy)
    using Arr-Subst Arr-not-Nil by auto

```

```

lemma head-strategy-Src:
shows  $Arr\ t \implies head\text{-}strategy\ (Src\ t) = head\text{-}strategy\ t$ 
  apply (induct t)
    apply auto
    using Arr.elims(2) by fastforce

```

```

lemma head-strategy-is-elementary:
shows  $\llbracket Arr\ t; \neg Ide\ (head\text{-}strategy\ t) \rrbracket \implies elementary\text{-}reduction\ (head\text{-}strategy\ t)$ 
  using Ide-Src
  apply (induct t)
    apply auto

```

```

proof –
  fix  $t1\ t2$ 
  assume  $t1: Arr\ t1$  and  $t2: Arr\ t2$ 
  assume  $t: \neg Ide\ (head\text{-}strategy\ (t1\ \circ\ t2))$ 
  assume  $1: \neg Ide\ (head\text{-}strategy\ t1) \implies elementary\text{-}reduction\ (head\text{-}strategy\ t1)$ 
  assume  $2: \neg Ide\ (head\text{-}strategy\ t2) \implies elementary\text{-}reduction\ (head\text{-}strategy\ t2)$ 
  show  $elementary\text{-}reduction\ (head\text{-}strategy\ (t1\ \circ\ t2))$ 
    using  $t\ t1\ t2\ 1\ 2\ Ide\text{-}Src\ Ide\text{-}implies\text{-}Arr$ 
    by (cases t1) auto
qed

```

```

lemma head-strategy-is-reduction-strategy:
shows reduction-strategy head-strategy
proof (unfold reduction-strategy-def, intro allI impI)
  fix  $t$ 
  show  $Ide\ t \implies Cinitial\ (head\text{-}strategy\ t)\ t$ 

```

```

proof (induct t)
  show  $Ide \# \implies Coinitial (head-strategy \#) \#$ 
    by simp
  show  $\bigwedge x. Ide \langle x \rangle \implies Coinitial (head-strategy \langle x \rangle) \langle x \rangle$ 
    by simp
  show  $\bigwedge t. \llbracket Ide t \implies Coinitial (head-strategy t) t; Ide \lambda[t] \rrbracket$ 
     $\implies Coinitial (head-strategy \lambda[t]) \lambda[t]$ 
    by simp
  fix t1 t2
    assume ind1:  $Ide t1 \implies Coinitial (head-strategy t1) t1$ 
    assume ind2:  $Ide t2 \implies Coinitial (head-strategy t2) t2$ 
    assume t:  $Ide (t1 \circ t2)$ 
    show  $Coinitial (head-strategy (t1 \circ t2)) (t1 \circ t2)$ 
      using t ind1 Ide-implies-Arr Ide-iff-Src-self
      by (cases t1) simp-all
    next
    fix t1 t2
      assume ind1:  $Ide t1 \implies Coinitial (head-strategy t1) t1$ 
      assume ind2:  $Ide t2 \implies Coinitial (head-strategy t2) t2$ 
      assume t:  $Ide (\lambda[t1] \bullet t2)$ 
      show  $Coinitial (head-strategy (\lambda[t1] \bullet t2)) (\lambda[t1] \bullet t2)$ 
        using t by auto
  qed
qed

```

The following function tests whether a term is an elementary reduction of the head redex.

```

fun is-head-reduction
where is-head-reduction «-»  $\longleftrightarrow False$ 
  | is-head-reduction  $\lambda[t] \longleftrightarrow is-head-reduction t$ 
  | is-head-reduction  $(\lambda[-] \circ -) \longleftrightarrow False$ 
  | is-head-reduction  $(t \circ u) \longleftrightarrow is-head-reduction t \wedge Ide u$ 
  | is-head-reduction  $(\lambda[t] \bullet u) \longleftrightarrow Ide t \wedge Ide u$ 
  | is-head-reduction  $\# \longleftrightarrow False$ 

```

**lemma** is-head-reduction-char:

```

shows  $is-head-reduction t \longleftrightarrow elementary-reduction t \wedge head-strategy (Src t) = t$ 
  apply (induct t)
  apply simp-all
proof -
  fix t1 t2
  assume ind:  $is-head-reduction t1 \longleftrightarrow$ 
     $elementary-reduction t1 \wedge head-strategy (Src t1) = t1$ 
  show  $is-head-reduction (t1 \circ t2) \longleftrightarrow$ 
     $(elementary-reduction t1 \wedge Ide t2 \vee Ide t1 \wedge elementary-reduction t2) \wedge$ 
     $head-strategy (Src t1 \circ Src t2) = t1 \circ t2$ 
  using ind Ide-implies-Arr Ide-iff-Src-self Ide-Src elementary-reduction-not-ide
    ide-char
  apply (cases t1)

```

```

    apply simp-all
    apply (metis Ide-Src arr-char elementary-reduction-is-arr)
    apply (metis Ide-Src arr-char elementary-reduction-is-arr)
  by metis
next
fix t1 t2
show Ide t1  $\wedge$  Ide t2  $\longleftrightarrow$  Ide t1  $\wedge$  Ide t2  $\wedge$  Src (Src t1) = t1  $\wedge$  Src (Src t2) = t2
  by (metis Ide-iff-Src-self Ide-implies-Arr)
qed

```

**lemma** *is-head-reductionI*:  
**assumes** *Arr t and elementary-reduction t and head-strategy (Src t) = t*  
**shows** *is-head-reduction t*  
**using** *assms is-head-reduction-char by blast*

The following function tests whether a redex in the head position of a term is marked.

```

fun contains-head-reduction
where contains-head-reduction «-»  $\longleftrightarrow$  False
  | contains-head-reduction  $\lambda[t]$   $\longleftrightarrow$  contains-head-reduction t
  | contains-head-reduction ( $\lambda[-] \circ -$ )  $\longleftrightarrow$  False
  | contains-head-reduction (t  $\circ$  u)  $\longleftrightarrow$  contains-head-reduction t  $\wedge$  Arr u
  | contains-head-reduction ( $\lambda[t] \bullet u$ )  $\longleftrightarrow$  Arr t  $\wedge$  Arr u
  | contains-head-reduction  $\#$   $\longleftrightarrow$  False

```

**lemma** *is-head-reduction-imp-contains-head-reduction*:  
**shows** *is-head-reduction t  $\implies$  contains-head-reduction t*  
**using** *Ide-implies-Arr*  
**apply** (induct t)  
**apply** auto  
**proof** –  
 fix t1 t2  
**assume** *ind1: is-head-reduction t1  $\implies$  contains-head-reduction t1*  
**assume** *ind2: is-head-reduction t2  $\implies$  contains-head-reduction t2*  
**assume** *t: is-head-reduction (t1  $\circ$  t2)*  
**show** *contains-head-reduction (t1  $\circ$  t2)*  
**using** *t ind1 ind2 Ide-implies-Arr*  
**by** (cases t1) auto  
**qed**

An *internal reduction* is one that does not contract any redex at the head position.

```

fun is-internal-reduction
where is-internal-reduction «-»  $\longleftrightarrow$  True
  | is-internal-reduction  $\lambda[t]$   $\longleftrightarrow$  is-internal-reduction t
  | is-internal-reduction ( $\lambda[t] \circ u$ )  $\longleftrightarrow$  Arr t  $\wedge$  Arr u
  | is-internal-reduction (t  $\circ$  u)  $\longleftrightarrow$  is-internal-reduction t  $\wedge$  Arr u
  | is-internal-reduction ( $\lambda[-] \bullet -$ )  $\longleftrightarrow$  False
  | is-internal-reduction  $\#$   $\longleftrightarrow$  False

```

**lemma** *is-internal-reduction-iff*:

**shows** *is-internal-reduction*  $t \iff \text{Arr } t \wedge \neg \text{contains-head-reduction } t$   
**apply** (*induct*  $t$ )  
**apply** *simp-all*  
**proof** –  
**fix**  $t1\ t2$   
**assume** *ind1*: *is-internal-reduction*  $t1 \iff \text{Arr } t1 \wedge \neg \text{contains-head-reduction } t1$   
**assume** *ind2*: *is-internal-reduction*  $t2 \iff \text{Arr } t2 \wedge \neg \text{contains-head-reduction } t2$   
**show** *is-internal-reduction*  $(t1 \circ t2) \iff$   
 $\text{Arr } t1 \wedge \text{Arr } t2 \wedge \neg \text{contains-head-reduction } (t1 \circ t2)$   
**using** *ind1 ind2*  
**apply** (*cases*  $t1$ )  
**apply** *simp-all*  
**by** *blast*  
**qed**

Head reduction steps are either  $\lesssim$ -prefixes of, or are preserved by, residuation along arbitrary reductions.

**lemma** *is-head-reduction-resid*:  
**shows**  $\llbracket \text{is-head-reduction } t; \text{Arr } u; \text{Src } t = \text{Src } u \rrbracket \implies t \lesssim u \vee \text{is-head-reduction } (t \setminus u)$   
**proof** (*induct*  $t$  *arbitrary*:  $u$ )  
**show**  $\bigwedge u. \llbracket \text{is-head-reduction } \sharp; \text{Arr } u; \text{Src } \sharp = \text{Src } u \rrbracket$   
 $\implies \sharp \lesssim u \vee \text{is-head-reduction } (\sharp \setminus u)$   
**by** *auto*  
**show**  $\bigwedge x u. \llbracket \text{is-head-reduction } \langle\langle x \rangle\rangle; \text{Arr } u; \text{Src } \langle\langle x \rangle\rangle = \text{Src } u \rrbracket$   
 $\implies \langle\langle x \rangle\rangle \lesssim u \vee \text{is-head-reduction } (\langle\langle x \rangle\rangle \setminus u)$   
**by** *auto*  
**fix**  $t\ u$   
**assume** *ind*:  $\bigwedge u. \llbracket \text{is-head-reduction } t; \text{Arr } u; \text{Src } t = \text{Src } u \rrbracket$   
 $\implies t \lesssim u \vee \text{is-head-reduction } (t \setminus u)$   
**assume**  $t$ : *is-head-reduction*  $\lambda[t]$   
**assume**  $u$ : *Arr*  $u$   
**assume**  $tu$ :  $\text{Src } \lambda[t] = \text{Src } u$   
**have**  $1$ : *Arr*  $t$   
**by** (*metis* *Arr-head-strategy* *head-strategy-Src* *is-head-reduction-char* *Arr.simps(3)*  $t\ tu\ u$ )  
**show**  $\lambda[t] \lesssim u \vee \text{is-head-reduction } (\lambda[t] \setminus u)$   
**using**  $t\ u\ tu\ 1\ \text{ind}$   
**by** (*cases*  $u$ ) *auto*  
**next**  
**fix**  $t1\ t2\ u$   
**assume** *ind1*:  $\bigwedge u1. \llbracket \text{is-head-reduction } t1; \text{Arr } u1; \text{Src } t1 = \text{Src } u1 \rrbracket$   
 $\implies t1 \lesssim u1 \vee \text{is-head-reduction } (t1 \setminus u1)$   
**assume** *ind2*:  $\bigwedge u2. \llbracket \text{is-head-reduction } t2; \text{Arr } u2; \text{Src } t2 = \text{Src } u2 \rrbracket$   
 $\implies t2 \lesssim u2 \vee \text{is-head-reduction } (t2 \setminus u2)$   
**assume**  $t$ : *is-head-reduction*  $(\lambda[t1] \bullet t2)$   
**assume**  $u$ : *Arr*  $u$   
**assume**  $tu$ :  $\text{Src } (\lambda[t1] \bullet t2) = \text{Src } u$   
**show**  $\lambda[t1] \bullet t2 \lesssim u \vee \text{is-head-reduction } ((\lambda[t1] \bullet t2) \setminus u)$   
**using**  $t\ u\ tu\ \text{ind1}\ \text{ind2}\ \text{Cinitial-iff-Con}\ \text{Ide-implies-Arr}\ \text{ide-char}\ \text{resid-Ide-Arr}\ \text{Ide-Subst}$   
**by** (*cases*  $u$ ; *cases* *un-App1*  $u$ ) *auto*

```

next
fix t1 t2 u
assume ind1:  $\bigwedge u1. \llbracket is-head-reduction\ t1; Arr\ u1; Src\ t1 = Src\ u1 \rrbracket$ 
            $\implies t1 \lesssim u1 \vee is-head-reduction\ (t1 \setminus u1)$ 
assume ind2:  $\bigwedge u2. \llbracket is-head-reduction\ t2; Arr\ u2; Src\ t2 = Src\ u2 \rrbracket$ 
            $\implies t2 \lesssim u2 \vee is-head-reduction\ (t2 \setminus u2)$ 
assume t: is-head-reduction (t1  $\circ$  t2)
assume u: Arr u
assume tu: Src (t1  $\circ$  t2) = Src u
have Arr (t1  $\circ$  t2)
  using is-head-reduction-char elementary-reduction-is-arr t by blast
hence t1: Arr t1 and t2: Arr t2
  by auto
have 0:  $\neg is-Lam\ t1$ 
  using t is-Lam-def by fastforce
have 1: is-head-reduction t1
  using t t1 by force
show t1  $\circ$  t2  $\lesssim$  u  $\vee is-head-reduction$  ((t1  $\circ$  t2)  $\setminus$  u)
proof -
  have  $\neg Ide\ ((t1 \circ t2) \setminus u) \implies is-head-reduction\ ((t1 \circ t2) \setminus u)$ 
  proof (intro is-head-reductionI)
    assume 2:  $\neg Ide\ ((t1 \circ t2) \setminus u)$ 
    have 3: is-App u  $\implies \neg Ide\ (t1 \setminus un-App1\ u) \vee \neg Ide\ (t2 \setminus un-App2\ u)$ 
    by (metis 2 ide-char lambda.collapse(3) lambda.discI(3) lambda.sel(3-4) prfx-App-iff)
    have 4: is-Beta u  $\implies \neg Ide\ (t1 \setminus un-Beta1\ u) \vee \neg Ide\ (t2 \setminus un-Beta2\ u)$ 
    using u tu 2
    by (metis 0 ConI Con-implies-is-Lam-iff-is-Lam  $\langle Arr\ (t1 \circ t2) \rangle$ 
        ConD(4) lambda.collapse(4) lambda.disc(8))
  show 5: Arr ((t1  $\circ$  t2)  $\setminus$  u)
    using Arr-resid  $\langle Arr\ (t1 \circ t2) \rangle$  tu u by auto
  show head-strategy (Src ((t1  $\circ$  t2)  $\setminus$  u)) = (t1  $\circ$  t2)  $\setminus$  u
  proof (cases u)
    show u =  $\# \implies head-strategy\ (Src\ ((t1 \circ t2) \setminus u)) = (t1 \circ t2) \setminus u$ 
    by simp
    show  $\bigwedge x. u = \langle x \rangle \implies head-strategy\ (Src\ ((t1 \circ t2) \setminus u)) = (t1 \circ t2) \setminus u$ 
    by auto
    show  $\bigwedge v. u = \lambda[v] \implies head-strategy\ (Src\ ((t1 \circ t2) \setminus u)) = (t1 \circ t2) \setminus u$ 
    by simp
    show  $\bigwedge u1\ u2. u = \lambda[u1] \bullet u2 \implies head-strategy\ (Src\ ((t1 \circ t2) \setminus u)) = (t1 \circ t2) \setminus u$ 
    by (metis 0 5 Arr-not-Nil ConD(4) Con-implies-is-Lam-iff-is-Lam lambda.disc(8))
  show  $\bigwedge u1\ u2. u = App\ u1\ u2 \implies head-strategy\ (Src\ ((t1 \circ t2) \setminus u)) = (t1 \circ t2) \setminus u$ 
  proof -
    fix u1 u2
    assume u1u2: u = u1  $\circ$  u2
    have head-strategy (Src ((t1  $\circ$  t2)  $\setminus$  u)) =
      head-strategy (Src (t1  $\setminus$  u1)  $\circ$  Src (t2  $\setminus$  u2))
    using u u1u2 tu t1 t2 Coinitial-iff-Con by auto
    also have ... = head-strategy (Trg u1  $\circ$  Trg u2)
    using 5 u1u2 Src-resid

```

```

    by (metis Arr-not-Nil ConD(1))
  also have ... = (t1 o t2) \ u
  proof (cases Trg u1)
    show Trg u1 = # ⇒ head-strategy (Trg u1 o Trg u2) = (t1 o t2) \ u
      using Arr-not-Nil u u1u2 by force
    show ∧x. Trg u1 = «x» ⇒ head-strategy (Trg u1 o Trg u2) = (t1 o t2) \ u
      using tu t u t1 t2 u1u2 Arr-not-Nil Ide-iff-Src-self
      by (cases u1; cases t1) auto
    show ∧v. Trg u1 = λ[v] ⇒ head-strategy (Trg u1 o Trg u2) = (t1 o t2) \ u
      using tu t u t1 t2 u1u2 Arr-not-Nil Ide-iff-Src-self
      apply (cases u1; cases t1)
        apply auto
      by (metis 2 5 Src-resid Trg.simps(3-4) resid.simps(3-4) resid-Src-Arr)
    show ∧u11 u12. Trg u1 = u11 o u12
      ⇒ head-strategy (Trg u1 o Trg u2) = (t1 o t2) \ u
  proof -
    fix u11 u12
    assume u1: Trg u1 = u11 o u12
    show head-strategy (Trg u1 o Trg u2) = (t1 o t2) \ u
    proof (cases Trg u1)
      show Trg u1 = # ⇒ ?thesis
        using u1 by simp
      show ∧x. Trg u1 = «x» ⇒ ?thesis
        apply simp
        using u1 by force
      show ∧v. Trg u1 = λ[v] ⇒ ?thesis
        using u1 by simp
      show ∧u11 u12. Trg u1 = u11 o u12 ⇒ ?thesis
        using t u tu u1u2 1 2 ind1 elementary-reduction-not-ide
          is-head-reduction-char Src-resid Ide-iff-Src-self
          ⟨Arr (t1 o t2)⟩ Coinitial-iff-Con
        by fastforce
      show ∧u11 u12. Trg u1 = λ[u11] • u12 ⇒ ?thesis
        using u1 by simp
    qed
  qed
  show ∧u11 u12. Trg u1 = λ[u11] • u12 ⇒ ?thesis
    using u1u2 u Ide-Trg by fastforce
  qed
  finally show head-strategy (Src ((t1 o t2) \ u)) = (t1 o t2) \ u
    by simp
  qed
  qed
  thus elementary-reduction ((t1 o t2) \ u)
    by (metis 2 5 Ide-Src Ide-implies-Arr head-strategy-is-elementary)
  qed
  thus ?thesis by blast
  qed
  qed

```

Internal reductions are closed under residuation.

**lemma** *is-internal-reduction-resid*:

**shows**  $\llbracket \text{is-internal-reduction } t; \text{is-internal-reduction } u; \text{Src } t = \text{Src } u \rrbracket$   
 $\implies \text{is-internal-reduction } (t \setminus u)$

**apply** (*induct t arbitrary: u*)

**apply** *auto*

**apply** (*metis Con-implies-Arr2 con-char weak-extensionality Arr.simps(2) Src.simps(2)*  
*parallel-strategy.simps(1) prfx-implies-con resid-Arr-Src subs-Ide*  
*subs-implies-prfx subs-parallel-strategy-Src*)

**proof** –

**fix** *t u*

**assume** *ind*:  $\bigwedge u. \llbracket \text{is-internal-reduction } u; \text{Src } t = \text{Src } u \rrbracket \implies \text{is-internal-reduction } (t \setminus u)$

**assume** *t*: *is-internal-reduction t*

**assume** *u*: *is-internal-reduction u*

**assume** *tu*:  $\lambda[\text{Src } t] = \text{Src } u$

**show** *is-internal-reduction* ( $\lambda[t] \setminus u$ )

**using** *t u tu ind*

**apply** (*cases u*)

**by** *auto fastforce*

**next**

**fix** *t1 t2 u*

**assume** *ind1*:  $\bigwedge u. \llbracket \text{is-internal-reduction } t1; \text{is-internal-reduction } u; \text{Src } t1 = \text{Src } u \rrbracket$   
 $\implies \text{is-internal-reduction } (t1 \setminus u)$

**assume** *t*: *is-internal-reduction* (*t1*  $\circ$  *t2*)

**assume** *u*: *is-internal-reduction u*

**assume** *tu*:  $\text{Src } t1 \circ \text{Src } t2 = \text{Src } u$

**show** *is-internal-reduction* ( $((t1 \circ t2) \setminus u)$ )

**using** *t u tu ind1 Coinitial-resid-resid Coinitial-iff-Con Arr-Src*  
*is-internal-reduction-iff*

**apply** *auto*

**apply** (*metis Arr.simps(4) Src.simps(4)*)

**proof** –

**assume** *t1*: *Arr t1* **and** *t2*: *Arr t2* **and** *u*: *Arr u*

**assume** *tu*:  $\text{Src } t1 \circ \text{Src } t2 = \text{Src } u$

**assume** *1*:  $\neg \text{contains-head-reduction } u$

**assume** *2*:  $\neg \text{contains-head-reduction } (t1 \circ t2)$

**assume** *3*:  $\text{contains-head-reduction } ((t1 \circ t2) \setminus u)$

**show** *False*

**using** *t1 t2 u tu 1 2 3 is-internal-reduction-iff*

**apply** (*cases u*)

**apply** *simp-all*

**apply** (*cases t1; cases un-App1 u*)

**apply** *simp-all*

**by** (*metis Coinitial-iff-Con ind1 Arr.simps(4) Src.simps(4) resid.simps(3)*)

**qed**

**qed**

A head reduction is preserved by residuation along an internal reduction, so a head reduction can only be canceled by a transition that contains a head reduction.

**lemma** *is-head-reduction-resid'*  
**shows**  $\llbracket \text{is-head-reduction } t; \text{is-internal-reduction } u; \text{Src } t = \text{Src } u \rrbracket$   
 $\implies \text{is-head-reduction } (t \setminus u)$   
**proof** (*induct t arbitrary: u*)  
**show**  $\bigwedge u. \llbracket \text{is-head-reduction } \#; \text{is-internal-reduction } u; \text{Src } \# = \text{Src } u \rrbracket$   
 $\implies \text{is-head-reduction } (\# \setminus u)$   
**by** *simp*  
**show**  $\bigwedge x u. \llbracket \text{is-head-reduction } \langle x \rangle; \text{is-internal-reduction } u; \text{Src } \langle x \rangle = \text{Src } u \rrbracket$   
 $\implies \text{is-head-reduction } (\langle x \rangle \setminus u)$   
**by** *simp*  
**show**  $\bigwedge t. \llbracket \bigwedge u. \llbracket \text{is-head-reduction } t; \text{is-internal-reduction } u; \text{Src } t = \text{Src } u \rrbracket$   
 $\implies \text{is-head-reduction } (t \setminus u);$   
 $\text{is-head-reduction } \lambda[t]; \text{is-internal-reduction } u; \text{Src } \lambda[t] = \text{Src } u \rrbracket$   
 $\implies \text{is-head-reduction } (\lambda[t] \setminus u)$   
**for** *u*  
**by** (*cases u, simp-all*) *fastforce*  
**fix** *t1 t2 u*  
**assume** *ind1*:  $\bigwedge u. \llbracket \text{is-head-reduction } t1; \text{is-internal-reduction } u; \text{Src } t1 = \text{Src } u \rrbracket$   
 $\implies \text{is-head-reduction } (t1 \setminus u)$   
**assume** *t*: *is-head-reduction* (*t1*  $\circ$  *t2*)  
**assume** *u*: *is-internal-reduction* *u*  
**assume** *tu*: *Src* (*t1*  $\circ$  *t2*) = *Src* *u*  
**show** *is-head-reduction* ((*t1*  $\circ$  *t2*)  $\setminus$  *u*)  
**using** *t u tu ind1*  
**apply** (*cases u*)  
**apply** *simp-all*  
**proof** (*intro conjI impI*)  
**fix** *u1 u2*  
**assume** *u1u2*: *u* = *u1*  $\circ$  *u2*  
**show** *1*: *Con* *t1 u1*  
**using** *Coinitial-iff-Con tu u1u2 ide-char*  
**by** (*metis ConD(1) Ide.simps(1) is-head-reduction.simps(9) is-head-reduction-resid*  
*is-internal-reduction.simps(9) is-internal-reduction-resid t u*)  
**show** *Con* *t2 u2*  
**using** *Coinitial-iff-Con tu u1u2 ide-char*  
**by** (*metis ConD(1) Ide.simps(1) is-head-reduction.simps(9) is-head-reduction-resid*  
*is-internal-reduction.simps(9) is-internal-reduction-resid t u*)  
**show** *is-head-reduction* (*t1*  $\setminus$  *u1*  $\circ$  *t2*  $\setminus$  *u2*)  
**using** *t u u1u2 1 Coinitial-iff-Con <Con t2 u2> ide-char ind1 resid-Ide-Arr*  
**apply** (*cases t1; simp-all; cases u1; simp-all; cases un-App1 u1*)  
**apply** *auto*  
**by** (*metis 1 ind1 is-internal-reduction.simps(6) resid.simps(3)*)  
**qed**  
**next**  
**fix** *t1 t2 u*  
**assume** *ind1*:  $\bigwedge u. \llbracket \text{is-head-reduction } t1; \text{is-internal-reduction } u; \text{Src } t1 = \text{Src } u \rrbracket$   
 $\implies \text{is-head-reduction } (t1 \setminus u)$   
**assume** *t*: *is-head-reduction* ( $\lambda[t1] \bullet t2$ )  
**assume** *u*: *is-internal-reduction* *u*

```

assume  $tu: \text{Src } (\lambda[t1] \bullet t2) = \text{Src } u$ 
show  $\text{is-head-reduction } ((\lambda[t1] \bullet t2) \setminus u)$ 
  using  $t\ u\ tu\ \text{ind1}$ 
  apply  $(\text{cases } u)$ 
    apply  $\text{simp-all}$ 
  by  $(\text{metis } \text{Con-implies-Arr1 } \text{is-head-reduction-resid } \text{is-internal-reduction.simps}(9)$ 
     $\text{is-internal-reduction-resid } \text{lambda.disc}(15) \text{ prfx-App-iff } t\ tu)$ 
qed

```

The following function differs from *head-strategy* in that it only selects an already-marked redex, whereas *head-strategy* marks the redex at the head position.

```

fun  $\text{head-redex}$ 
where  $\text{head-redex } \# = \#$ 
  |  $\text{head-redex } \langle\langle x \rangle\rangle = \langle\langle x \rangle\rangle$ 
  |  $\text{head-redex } \lambda[t] = \lambda[\text{head-redex } t]$ 
  |  $\text{head-redex } (\lambda[t] \circ u) = \lambda[\text{Src } t] \circ \text{Src } u$ 
  |  $\text{head-redex } (t \circ u) = \text{head-redex } t \circ \text{Src } u$ 
  |  $\text{head-redex } (\lambda[t] \bullet u) = (\lambda[\text{Src } t] \bullet \text{Src } u)$ 

```

**lemma** *elementary-reduction-head-redex*:

```

shows  $\llbracket \text{Arr } t; \neg \text{Ide } (\text{head-redex } t) \rrbracket \implies \text{elementary-reduction } (\text{head-redex } t)$ 
  using  $\text{Ide-Src}$ 
  apply  $(\text{induct } t)$ 
  apply  $\text{auto}$ 
proof –
  show  $\bigwedge t2. \llbracket \neg \text{Ide } (\text{head-redex } t1) \implies \text{elementary-reduction } (\text{head-redex } t1);$ 
     $\neg \text{Ide } (\text{head-redex } (t1 \circ t2));$ 
     $\bigwedge t. \text{Arr } t \implies \text{Ide } (\text{Src } t); \text{Arr } t1; \text{Arr } t2 \rrbracket$ 
     $\implies \text{elementary-reduction } (\text{head-redex } (t1 \circ t2))$ 
  for  $t1$ 
  using  $\text{Ide-Src}$ 
  by  $(\text{cases } t1) \text{ auto}$ 
qed

```

**lemma** *subs-head-redex*:

```

shows  $\text{Arr } t \implies \text{head-redex } t \sqsubseteq t$ 
  using  $\text{Ide-Src } \text{subs-Ide}$ 
  apply  $(\text{induct } t)$ 
  apply  $\text{simp-all}$ 
proof –
  show  $\bigwedge t2. \llbracket \text{head-redex } t1 \sqsubseteq t1; \text{head-redex } t2 \sqsubseteq t2;$ 
     $\text{Arr } t1 \wedge \text{Arr } t2; \bigwedge t. \text{Arr } t \implies \text{Ide } (\text{Src } t);$ 
     $\bigwedge u\ t. \llbracket \text{Ide } u; \text{Src } t = \text{Src } u \rrbracket \implies u \sqsubseteq t \rrbracket$ 
     $\implies \text{head-redex } (t1 \circ t2) \sqsubseteq t1 \circ t2$ 
  for  $t1$ 
  using  $\text{Ide-Src } \text{subs-Ide}$ 
  by  $(\text{cases } t1) \text{ auto}$ 
qed

```

**lemma** *contains-head-reduction-iff*:  
**shows**  $\text{contains-head-reduction } t \iff \text{Arr } t \wedge \neg \text{Ide } (\text{head-redex } t)$   
**apply** (*induct t*)  
**apply** *simp-all*  
**proof** –  
**show**  $\bigwedge t2. \text{contains-head-reduction } t1 = (\text{Arr } t1 \wedge \neg \text{Ide } (\text{head-redex } t1))$   
 $\implies \text{contains-head-reduction } (t1 \circ t2) =$   
 $(\text{Arr } t1 \wedge \text{Arr } t2 \wedge \neg \text{Ide } (\text{head-redex } (t1 \circ t2)))$   
**for** *t1*  
**using** *Ide-Src*  
**by** (*cases t1*) *auto*  
**qed**

**lemma** *head-redex-is-head-reduction*:  
**shows**  $\llbracket \text{Arr } t; \text{contains-head-reduction } t \rrbracket \implies \text{is-head-reduction } (\text{head-redex } t)$   
**using** *Ide-Src*  
**apply** (*induct t*)  
**apply** *simp-all*  
**proof** –  
**show**  $\bigwedge t2. \llbracket \text{contains-head-reduction } t1 \implies \text{is-head-reduction } (\text{head-redex } t1);$   
 $\text{Arr } t1 \wedge \text{Arr } t2;$   
 $\text{contains-head-reduction } (t1 \circ t2); \bigwedge t. \text{Arr } t \implies \text{Ide } (\text{Src } t) \rrbracket$   
 $\implies \text{is-head-reduction } (\text{head-redex } (t1 \circ t2))$   
**for** *t1*  
**using** *Ide-Src contains-head-reduction-iff subs-implies-prfx*  
**by** (*cases t1*) *auto*  
**qed**

**lemma** *Arr-head-redex*:  
**assumes** *Arr t*  
**shows**  $\text{Arr } (\text{head-redex } t)$   
**using** *assms Ide-implies-Arr elementary-reduction-head-redex elementary-reduction-is-arr*  
**by** *blast*

**lemma** *Src-head-redex*:  
**assumes** *Arr t*  
**shows**  $\text{Src } (\text{head-redex } t) = \text{Src } t$   
**using** *assms*  
**by** (*metis Cinitial-iff-Con Ide.simps(1) ide-char subs-head-redex subs-implies-prfx*)

**lemma** *Con-Arr-head-redex*:  
**assumes** *Arr t*  
**shows**  $\text{Con } t (\text{head-redex } t)$   
**using** *assms*  
**by** (*metis Con-sym Ide.simps(1) ide-char subs-head-redex subs-implies-prfx*)

**lemma** *is-head-reduction-if*:  
**shows**  $\llbracket \text{contains-head-reduction } u; \text{elementary-reduction } u \rrbracket \implies \text{is-head-reduction } u$   
**apply** (*induct u*)

```

    apply auto
  using contains-head-reduction.elims(2)
  apply fastforce
proof -
  fix u1 u2
  assume u1: Ide u1
  assume u2: elementary-reduction u2
  assume 1: contains-head-reduction (u1 o u2)
  have False
    using u1 u2 1
    apply (cases u1)
    apply auto
    by (metis Arr-head-redex Ide-iff-Src-self Src-head-redex contains-head-reduction-iff
        ide-char resid-Arr-Src subs-head-redex subs-implies-prfx u1)
  thus is-head-reduction (u1 o u2)
    by blast
qed

```

**lemma** (in *reduction-paths*) *head-redex-decomp*:  
**assumes**  $\Lambda.Arr\ t$   
**shows**  $[\Lambda.head-redex\ t] @ [t \setminus \Lambda.head-redex\ t] \sim^* [t]$   
**using** *assms prfx-decomp*  $\Lambda.subs-head-redex$   $\Lambda.subs-implies-prfx$   
**by** (metis *Ide.simps(2)* *Resid.simps(3)*  $\Lambda.prfx-implies-con\ ide-char$ )

An internal reduction cannot create a new head redex.

**lemma** *internal-reduction-preserves-no-head-redex*:  
**shows**  $\llbracket is-internal-reduction\ u; Ide\ (head-strategy\ (Src\ u)) \rrbracket$   
 $\implies Ide\ (head-strategy\ (Trg\ u))$   
**apply** (induct u)  
**apply** *simp-all*  
**proof** -  
 fix u1 u2  
**assume** *ind1*:  $\llbracket is-internal-reduction\ u1; Ide\ (head-strategy\ (Src\ u1)) \rrbracket$   
 $\implies Ide\ (head-strategy\ (Trg\ u1))$   
**assume** *ind2*:  $\llbracket is-internal-reduction\ u2; Ide\ (head-strategy\ (Src\ u2)) \rrbracket$   
 $\implies Ide\ (head-strategy\ (Trg\ u2))$   
**assume** *u*: *is-internal-reduction* (u1 o u2)  
**assume** *1*: *Ide* (head-strategy (Src u1 o Src u2))  
**show** *Ide* (head-strategy (Trg u1 o Trg u2))  
**using** *u 1 ind1 ind2 Ide-Src Ide-Trg Ide-implies-Arr*  
**by** (cases u1) *auto*  
**qed**

**lemma** *head-reduction-unique*:  
**shows**  $\llbracket is-head-reduction\ t; is-head-reduction\ u; coinital\ t\ u \rrbracket \implies t = u$   
**by** (metis *Coinitial-iff-Con con-def confluence is-head-reduction-char null-char*)

Residuation along internal reductions preserves head reductions.

**lemma** *resid-head-strategy-internal*:

**shows** *is-internal-reduction*  $u \implies \text{head-strategy } (Src\ u) \setminus u = \text{head-strategy } (Trg\ u)$   
**using** *internal-reduction-preserves-no-head-redex* *Arr-head-strategy* *Ide-iff-Src-self*  
*Src-head-strategy* *Src-resid* *head-strategy-is-elementary* *is-head-reduction-char*  
*is-head-reduction-resid' is-internal-reduction-iff*  
**apply** (*cases*  $u$ )  
**apply** *simp-all*  
**apply** (*metis* *head-strategy-Src* *resid-Src-Arr*)  
**apply** (*metis* *head-strategy-Src* *Arr.simps(4)* *Src.simps(4)* *Trg.simps(3)* *resid-Src-Arr*)  
**by** *blast*

An internal reduction followed by a head reduction can be expressed as a join of the internal reduction with a head reduction.

**lemma** *resid-head-strategy-Src*:  
**assumes** *is-internal-reduction*  $t$  **and** *is-head-reduction*  $u$   
**and** *seq*  $t\ u$   
**shows** *head-strategy*  $(Src\ t) \setminus t = u$   
**and** *composite-of*  $t\ u$  (*Join* (*head-strategy*  $(Src\ t)$ )  $t$ )  
**proof** –  
**show** *1*: *head-strategy*  $(Src\ t) \setminus t = u$   
**using** *assms* *internal-reduction-preserves-no-head-redex* *resid-head-strategy-internal*  
*elementary-reduction-not-ide* *ide-char* *is-head-reduction-char* *seq-char*  
**by** *force*  
**show** *composite-of*  $t\ u$  (*Join* (*head-strategy*  $(Src\ t)$ )  $t$ )  
**using** *assms(3)* *1* *Arr-head-strategy* *Src-head-strategy* *join-of-Join* *join-of-def* *seq-char*  
**by** *force*  
**qed**

**lemma** *App-Var-contains-no-head-reduction*:  
**shows**  $\neg \text{contains-head-reduction } (\llbracket x \rrbracket \circ u)$   
**by** *simp*

**lemma** *hgt-resid-App-head-redex*:  
**assumes** *Arr*  $(t \circ u)$  **and**  $\neg \text{Ide } (\text{head-redex } (t \circ u))$   
**shows** *hgt*  $((t \circ u) \setminus \text{head-redex } (t \circ u)) < \text{hgt } (t \circ u)$   
**using** *assms* *contains-head-reduction-iff* *elementary-reduction-decreases-hgt*  
*elementary-reduction-head-redex* *subs-head-redex*  
**by** *blast*

### 3.5.3 Leftmost Reduction

Leftmost (or normal-order) reduction is the strategy that produces an elementary reduction path by contracting the leftmost redex at each step. It agrees with head reduction as long as there is a head redex, otherwise it continues on with the next subterm to the right.

**fun** *leftmost-strategy*  
**where** *leftmost-strategy*  $\llbracket x \rrbracket = \llbracket x \rrbracket$   
| *leftmost-strategy*  $\lambda[t] = \lambda[\text{leftmost-strategy } t]$   
| *leftmost-strategy*  $(\lambda[t] \circ u) = \lambda[t] \bullet u$

```

| leftmost-strategy (t ◦ u) =
  (if ¬ Ide (leftmost-strategy t)
   then leftmost-strategy t ◦ u
   else t ◦ leftmost-strategy u)
| leftmost-strategy (λ[t] • u) = λ[t] • u
| leftmost-strategy ‡ = ‡

```

**definition** *is-leftmost-reduction*

**where** *is-leftmost-reduction*  $t \iff \text{elementary-reduction } t \wedge \text{leftmost-strategy } (\text{Src } t) = t$

**lemma** *leftmost-strategy-is-reduction-strategy*:

**shows** *reduction-strategy leftmost-strategy*

**proof** (*unfold reduction-strategy-def, intro allI impI*)

**fix**  $t$

**show**  $\text{Ide } t \implies \text{Coinitial } (\text{leftmost-strategy } t) \ t$

**proof** (*induct t, auto*)

```

show  $\bigwedge t2. \llbracket \text{Arr } (\text{leftmost-strategy } t1); \text{Arr } (\text{leftmost-strategy } t2);$ 
   $\text{Ide } t1; \text{Ide } t2;$ 
   $\text{Arr } t1; \text{Src } (\text{leftmost-strategy } t1) = \text{Src } t1;$ 
   $\text{Arr } t2; \text{Src } (\text{leftmost-strategy } t2) = \text{Src } t2 \rrbracket$ 
   $\implies \text{Arr } (\text{leftmost-strategy } (t1 \circ t2))$ 

```

**for**  $t1$

**by** (*cases t1*) *auto*

**qed**

**qed**

**lemma** *elementary-reduction-leftmost-strategy*:

**shows**  $\text{Ide } t \implies \text{elementary-reduction } (\text{leftmost-strategy } t) \vee \text{Ide } (\text{leftmost-strategy } t)$

**apply** (*induct t*)

**apply** *simp-all*

**proof** –

**fix**  $t1 \ t2$

```

show  $\llbracket \text{elementary-reduction } (\text{leftmost-strategy } t1) \vee \text{Ide } (\text{leftmost-strategy } t1);$ 
   $\text{elementary-reduction } (\text{leftmost-strategy } t2) \vee \text{Ide } (\text{leftmost-strategy } t2);$ 
   $\text{Ide } t1 \wedge \text{Ide } t2 \rrbracket$ 
   $\implies \text{elementary-reduction } (\text{leftmost-strategy } (t1 \circ t2)) \vee$ 
   $\text{Ide } (\text{leftmost-strategy } (t1 \circ t2))$ 

```

**by** (*cases t1*) *auto*

**qed**

**lemma** (*in lambda-calculus*) *leftmost-strategy-selects-head-reduction*:

**shows** *is-head-reduction*  $t \implies t = \text{leftmost-strategy } (\text{Src } t)$

**proof** (*induct t*)

```

show  $\bigwedge t1 \ t2. \llbracket \text{is-head-reduction } t1 \implies t1 = \text{leftmost-strategy } (\text{Src } t1);$ 
   $\text{is-head-reduction } (t1 \circ t2) \rrbracket$ 
   $\implies t1 \circ t2 = \text{leftmost-strategy } (\text{Src } (t1 \circ t2))$ 

```

**proof** –

**fix**  $t1 \ t2$

```

assume ind1: is-head-reduction t1  $\implies$  t1 = leftmost-strategy (Src t1)
assume t: is-head-reduction (t1  $\circ$  t2)
show t1  $\circ$  t2 = leftmost-strategy (Src (t1  $\circ$  t2))
  using t ind1
  apply (cases t1)
    apply simp-all
    apply (cases Src t1)
      apply simp-all
  using ind1
    apply force
  using ind1
    apply force
  using ind1
    apply force
  apply (metis Ide-iff-Src-self Ide-implies-Arr elementary-reduction-not-ide
    ide-char ind1 is-head-reduction-char)
  using ind1
  apply force
  by (metis Ide-iff-Src-self Ide-implies-Arr)
qed
show  $\bigwedge t1\ t2.$   $\llbracket is-head-reduction\ t1 \implies t1 = leftmost-strategy\ (Src\ t1);$ 
  is-head-reduction ( $\lambda[t1] \bullet t2$ )  $\rrbracket$ 
   $\implies \lambda[t1] \bullet t2 = leftmost-strategy\ (Src\ (\lambda[t1] \bullet t2))$ 
  by (metis Ide-iff-Src-self Ide-implies-Arr Src.simps(5)
    is-head-reduction.simps(8) leftmost-strategy.simps(3))
qed auto

```

```

lemma has-redex-iff-not-Ide-leftmost-strategy:
shows Arr t  $\implies$  has-redex t  $\longleftrightarrow$   $\neg$  Ide (leftmost-strategy (Src t))
  apply (induct t)
  apply simp-all
proof –
  fix t1 t2
  assume ind1: Ide (parallel-strategy t1)  $\longleftrightarrow$  Ide (leftmost-strategy (Src t1))
  assume ind2: Ide (parallel-strategy t2)  $\longleftrightarrow$  Ide (leftmost-strategy (Src t2))
  assume t: Arr t1  $\wedge$  Arr t2
  show Ide (parallel-strategy (t1  $\circ$  t2))  $\longleftrightarrow$ 
    Ide (leftmost-strategy (Src t1  $\circ$  Src t2))
  using t ind1 ind2 Ide-Src Ide-iff-Src-self
  by (cases t1) auto
qed

```

```

lemma leftmost-reduction-preservation:
shows  $\llbracket is-leftmost-reduction\ t; elementary-reduction\ u; \neg is-leftmost-reduction\ u;$ 
  coinitial t u  $\rrbracket \implies is-leftmost-reduction\ (t \setminus u)$ 
proof (induct t arbitrary: u)
  show  $\bigwedge u.$  coinitial  $\#$  u  $\implies is-leftmost-reduction\ (\# \setminus u)$ 
  by simp
  show  $\bigwedge x\ u.$  is-leftmost-reduction  $\langle\langle x \rangle\rangle \implies is-leftmost-reduction\ (\langle\langle x \rangle\rangle \setminus u)$ 

```

```

    by (simp add: is-leftmost-reduction-def)
  fix t u
  show  $\llbracket \bigwedge u. \llbracket \text{is-leftmost-reduction } t; \text{ elementary-reduction } u; \neg \text{is-leftmost-reduction } u; \text{ coinitial } t \ u \rrbracket \implies \text{is-leftmost-reduction } (t \setminus u); \text{is-leftmost-reduction } (\text{Lam } t); \text{ elementary-reduction } u; \neg \text{is-leftmost-reduction } u; \text{ coinitial } \lambda[t] \ u \rrbracket \implies \text{is-leftmost-reduction } (\lambda[t] \setminus u) \rrbracket$ 
    using is-leftmost-reduction-def
    by (cases u) auto
  next
  fix t1 t2 u
  show  $\llbracket \text{is-leftmost-reduction } (\lambda[t1] \bullet t2); \text{ elementary-reduction } u; \neg \text{is-leftmost-reduction } u; \text{ coinitial } (\lambda[t1] \bullet t2) \ u \rrbracket \implies \text{is-leftmost-reduction } ((\lambda[t1] \bullet t2) \setminus u) \rrbracket$ 
    using is-leftmost-reduction-def Src-resid Ide-Trg Ide-iff-Src-self Arr-Trg Arr-not-Nil
    apply (cases u)
    apply simp-all
    by (cases un-App1 u) auto
  assume ind1:  $\bigwedge u. \llbracket \text{is-leftmost-reduction } t1; \text{ elementary-reduction } u; \neg \text{is-leftmost-reduction } u; \text{ coinitial } t1 \ u \rrbracket \implies \text{is-leftmost-reduction } (t1 \setminus u) \rrbracket$ 
  assume ind2:  $\bigwedge u. \llbracket \text{is-leftmost-reduction } t2; \text{ elementary-reduction } u; \neg \text{is-leftmost-reduction } u; \text{ coinitial } t2 \ u \rrbracket \implies \text{is-leftmost-reduction } (t2 \setminus u) \rrbracket$ 
  assume 1: is-leftmost-reduction (t1  $\circ$  t2)
  assume 2: elementary-reduction u
  assume 3:  $\neg \text{is-leftmost-reduction } u$ 
  assume 4: coinitial (t1  $\circ$  t2) u
  show is-leftmost-reduction ((t1  $\circ$  t2)  $\setminus$  u)
    using 1 2 3 4 ind1 ind2 is-leftmost-reduction-def Src-resid
    apply (cases u)
    apply auto[3]
  proof -
    show  $\bigwedge u1 \ u2. u = \lambda[u1] \bullet u2 \implies \text{is-leftmost-reduction } ((t1 \circ t2) \setminus u) \rrbracket$ 
      by (metis 2 3 is-leftmost-reduction-def elementary-reduction.simps(5) is-head-reduction.simps(8) leftmost-strategy-selects-head-reduction)
    fix u1 u2
    assume u:  $u = u1 \circ u2$ 
    show is-leftmost-reduction ((t1  $\circ$  t2)  $\setminus$  u)
      using u 1 2 3 4 ind1 ind2 is-leftmost-reduction-def Src-resid Ide-Trg elementary-reduction-not-ide
      apply (cases u)
      apply simp-all
      apply (cases u1)
      apply simp-all
      apply auto[1]
    using Ide-iff-Src-self
    apply simp-all
  proof -

```

```

fix u11 u12
assume u: u = u11 ◦ u12 ◦ u2
assume u1: u1 = u11 ◦ u12
have A: (elementary-reduction t1 ∧ Src u2 = t2 ∨
  Src u11 ◦ Src u12 = t1 ∧ elementary-reduction t2) ∧
  (if ¬ Ide (leftmost-strategy (Src u11 ◦ Src u12))
  then leftmost-strategy (Src u11 ◦ Src u12) ◦ Src u2
  else Src u11 ◦ Src u12 ◦ leftmost-strategy (Src u2)) = t1 ◦ t2
  using 1 4 Ide-iff-Src-self is-leftmost-reduction-def u by auto
have B: (elementary-reduction u11 ∧ Src u12 = u12 ∨
  Src u11 = u11 ∧ elementary-reduction u12) ∧ Src u2 = u2 ∨
  Src u11 = u11 ∧ Src u12 = u12 ∧ elementary-reduction u2
  using 2 4 Ide-iff-Src-self u by force
have C: t1 = u11 ◦ u12 ⟶ t2 ≠ u2
  using 1 3 u by fastforce
have D: Arr t1 ∧ Arr t2 ∧ Arr u11 ∧ Arr u12 ∧ Arr u2 ∧
  Src t1 = Src u11 ◦ Src u12 ∧ Src t2 = Src u2
  using 4 u by force
have E:  $\bigwedge u. \llbracket \text{elementary-reduction } t1 \wedge \text{leftmost-strategy } (Src\ u) = t1; \text{elementary-reduction } u; \text{t1} \neq u; \text{Arr } u \wedge \text{Src } u11 \circ \text{Src } u12 = \text{Src } u \rrbracket$ 
   $\implies \text{elementary-reduction } (t1 \setminus u) \wedge$ 
   $\text{leftmost-strategy } (Trg\ u) = t1 \setminus u$ 
  using D Src-resid ind1 is-leftmost-reduction-def by auto
have F:  $\bigwedge u. \llbracket \text{elementary-reduction } t2 \wedge \text{leftmost-strategy } (Src\ u) = t2; \text{elementary-reduction } u; \text{t2} \neq u; \text{Arr } u \wedge \text{Src } u2 = \text{Src } u \rrbracket$ 
   $\implies \text{elementary-reduction } (t2 \setminus u) \wedge$ 
   $\text{leftmost-strategy } (Trg\ u) = t2 \setminus u$ 
  using D Src-resid ind2 is-leftmost-reduction-def by auto
have G:  $\bigwedge t. \text{elementary-reduction } t \implies \neg \text{Ide } t$ 
  using elementary-reduction-not-ide ide-char by blast
have H: elementary-reduction (t1 \ (u11 ◦ u12)) ∧ Ide (t2 \ u2) ∨
  Ide (t1 \ (u11 ◦ u12)) ∧ elementary-reduction (t2 \ u2)
proof (cases Ide (t2 \ u2))
  assume 1: Ide (t2 \ u2)
  hence elementary-reduction (t1 \ (u11 ◦ u12))
  by (metis A B C D E F G Ide-Src Arr.simps(4) Src.simps(4)
  elementary-reduction.simps(4) lambda.inject(3) resid-Arr-Src)
  thus ?thesis
  using 1 by auto
next
assume 1: ¬ Ide (t2 \ u2)
hence Ide (t1 \ (u11 ◦ u12)) ∧ elementary-reduction (t2 \ u2)
apply (intro conjI)
  apply (metis 1 A D Ide-Src Arr.simps(4) Src.simps(4) resid-Ide-Arr)
  by (metis A B C D F Ide-iff-Src-self lambda.inject(3) resid-Arr-Src resid-Ide-Arr)

```

**thus** *?thesis by simp*  
**qed**  
**show**  $(\neg \text{Ide } (\text{leftmost-strategy } (\text{Trg } u11 \circ \text{Trg } u12)) \longrightarrow$   
 $(\text{elementary-reduction } (t1 \setminus (u11 \circ u12)) \wedge \text{Ide } (t2 \setminus u2) \vee$   
 $\text{Ide } (t1 \setminus (u11 \circ u12)) \wedge \text{elementary-reduction } (t2 \setminus u2)) \wedge$   
 $\text{leftmost-strategy } (\text{Trg } u11 \circ \text{Trg } u12) = t1 \setminus (u11 \circ u12) \wedge \text{Trg } u2 = t2 \setminus u2) \wedge$   
 $(\text{Ide } (\text{leftmost-strategy } (\text{Trg } u11 \circ \text{Trg } u12)) \longrightarrow$   
 $(\text{elementary-reduction } (t1 \setminus (u11 \circ u12)) \wedge \text{Ide } (t2 \setminus u2) \vee$   
 $\text{Ide } (t1 \setminus (u11 \circ u12)) \wedge \text{elementary-reduction } (t2 \setminus u2)) \wedge$   
 $\text{Trg } u11 \circ \text{Trg } u12 = t1 \setminus (u11 \circ u12) \wedge \text{leftmost-strategy } (\text{Trg } u2) = t2 \setminus u2)$   
**proof** (*intro conjI impI*)  
**show**  $H: \text{elementary-reduction } (t1 \setminus (u11 \circ u12)) \wedge \text{Ide } (t2 \setminus u2) \vee$   
 $\text{Ide } (t1 \setminus (u11 \circ u12)) \wedge \text{elementary-reduction } (t2 \setminus u2)$   
**by fact**  
**show**  $H: \text{elementary-reduction } (t1 \setminus (u11 \circ u12)) \wedge \text{Ide } (t2 \setminus u2) \vee$   
 $\text{Ide } (t1 \setminus (u11 \circ u12)) \wedge \text{elementary-reduction } (t2 \setminus u2)$   
**by fact**  
**assume**  $K: \neg \text{Ide } (\text{leftmost-strategy } (\text{Trg } u11 \circ \text{Trg } u12))$   
**show**  $J: \text{Trg } u2 = t2 \setminus u2$   
**using**  $A B D G K \text{ has-redex-iff-not-Ide-leftmost-strategy}$   
 $\text{NF-def NF-iff-has-no-redex NF-App-iff resid-Arr-Src resid-Src-Arr}$   
**by** (*metis lambda.inject(3)*)  
**show**  $\text{leftmost-strategy } (\text{Trg } u11 \circ \text{Trg } u12) = t1 \setminus (u11 \circ u12)$   
**using**  $2 A B C D E G H J u \text{ Ide-Trg Src-Src}$   
 $\text{has-redex-iff-not-Ide-leftmost-strategy resid-Arr-Ide resid-Src-Arr}$   
**by** (*metis Arr.simps(4) Ide.simps(4) Src.simps(4) Trg.simps(3)*)  
 $\text{elementary-reduction.simps(4) lambda.inject(3)}$   
**next**  
**assume**  $K: \text{Ide } (\text{leftmost-strategy } (\text{Trg } u11 \circ \text{Trg } u12))$   
**show**  $I: \text{Trg } u11 \circ \text{Trg } u12 = t1 \setminus (u11 \circ u12)$   
**using**  $2 A D E K u \text{ Coinitial-resid-resid ConI resid-Arr-self resid-Ide-Arr}$   
 $\text{resid-Arr-Ide Ide-iff-Src-self Src-resid}$   
**apply** (*cases Ide (leftmost-strategy (Src u11 o Src u12))*)  
**apply simp**  
**using** *lambda-calculus.Con-Arr-Src(2)*  
**apply force**  
**apply simp**  
**using**  $u1 G H \text{ Coinitial-iff-Con}$   
**apply** (*cases elementary-reduction u11;*  
*cases elementary-reduction u12*)  
**apply simp-all**  
**apply metis**  
**apply** (*metis Src.simps(4) Trg.simps(3) elementary-reduction.simps(1,4)*)  
**apply** (*metis Src.simps(4) Trg.simps(3) elementary-reduction.simps(1,4)*)  
**by** (*metis Trg-Src*)  
**show**  $\text{leftmost-strategy } (\text{Trg } u2) = t2 \setminus u2$   
**using**  $2 A C D F G H I u \text{ Ide-Trg Ide-iff-Src-self NF-def NF-iff-has-no-redex}$   
 $\text{has-redex-iff-not-Ide-leftmost-strategy resid-Ide-Arr}$   
**by** (*metis Arr.simps(4) Src.simps(4) Trg.simps(3) elementary-reduction.simps(4)*)

```

      lambda.inject(3))
    qed
  qed
qed
end

```

## 3.6 Standard Reductions

In this section, we define the notion of a *standard reduction*, which is an elementary reduction path that performs reductions from left to right, possibly skipping some redexes that could be contracted. Once a redex has been skipped, neither that redex nor any redex to its left will subsequently be contracted. We then define and prove correct a function that transforms an arbitrary elementary reduction path into a congruent standard reduction path. Using this function, we prove the Standardization Theorem, which says that every elementary reduction path is congruent to a standard reduction path. We then show that a standard reduction path that reaches a normal form is in fact a leftmost reduction path. From this fact and the Standardization Theorem we prove the Leftmost Reduction Theorem: leftmost reduction is a normalizing strategy.

The Standardization Theorem was first proved by Curry and Feys [3], with subsequent proofs given by a number of authors. Formalized proofs have also been given; a recent one (using Agda) is presented in [2], with references to earlier work. The version of the theorem that we formalize here is a “strong” version, which asserts the existence of a standard reduction path congruent to a given elementary reduction path. At the core of the proof is a function that directly transforms a given reduction path into a standard one, using an algorithm roughly analogous to insertion sort. The Finite Development Theorem is used in the proof of termination. The proof of correctness is long, due to the number of cases that have to be considered, but the use of a proof assistant makes this manageable.

### 3.6.1 Standard Reduction Paths

#### ‘Standardly Sequential’ Reductions

We first need to define the notion of a “standard reduction”. In contrast to what is typically done by other authors, we define this notion by direct comparison of adjacent terms in an elementary reduction path, rather than by using devices such as a numbering of subterms from left to right.

The following function decides when two terms  $t$  and  $u$  are elementary reductions that are “standardly sequential”. This means that  $t$  and  $u$  are sequential, but in addition no marked redex in  $u$  is the residual of an (unmarked) redex “to the left of” any marked redex in  $t$ . Some care is required to make sure that the recursive definition captures what we intend. Most of the clauses are readily understandable. One clause that perhaps could use some explanation is the one for  $sseq ((\lambda[t] \bullet u) \circ v) w$ . Referring to the

previously proved fact *seq-cases*, which classifies the way in which two terms  $t$  and  $u$  can be sequential, we see that one case that must be covered is when  $t$  has the form  $\lambda[t] \bullet v) \circ w$  and the top-level constructor of  $u$  is *Beta*. In this case, it is the reduction of  $t$  that creates the top-level redex contracted in  $u$ , so it is impossible for  $u$  to be a residual of a redex that already exists in *Src*  $t$ .

**context** *lambda-calculus*  
**begin**

```

fun sseq
where sseq -  $\#$  = False
  | sseq «-» «-» = False
  | sseq  $\lambda[t]$   $\lambda[t']$  = sseq  $t$   $t'$ 
  | sseq  $(t \circ u)$   $(t' \circ u')$  =
    ((sseq  $t$   $t' \wedge$  Ide  $u \wedge u = u'$ )  $\vee$ 
     (Ide  $t \wedge t = t' \wedge$  sseq  $u$   $u'$ )  $\vee$ 
     (elementary-reduction  $t \wedge$  Trg  $t = t' \wedge$ 
      ( $u =$  Src  $u' \wedge$  elementary-reduction  $u'$ )))
  | sseq  $(\lambda[t] \circ u)$   $(\lambda[t'] \bullet u')$  = False
  | sseq  $(\lambda[t] \bullet u) \circ v$   $w$  =
    (Ide  $t \wedge$  Ide  $u \wedge$  Ide  $v \wedge$  elementary-reduction  $w \wedge$  seq  $(\lambda[t] \bullet u) \circ v$ )  $w$ )
  | sseq  $(\lambda[t] \bullet u)$   $v$  = (Ide  $t \wedge$  Ide  $u \wedge$  elementary-reduction  $v \wedge$  seq  $(\lambda[t] \bullet u)$   $v$ )
  | sseq - - = False

```

**lemma** *sseq-imp-imp-imp*:

**shows** *sseq*  $t$   $u \implies$  *seq*  $t$   $u$

**proof** (*induct*  $t$  *arbitrary*:  $u$ )

**show**  $\bigwedge u. \text{sseq } \# u \implies \text{seq } \# u$

**using** *sseq.elims(1)* **by** *blast*

**fix**  $u$

**show**  $\bigwedge x. \text{sseq } \langle x \rangle u \implies \text{seq } \langle x \rangle u$

**using** *sseq.elims(1)* **by** *blast*

**show**  $\bigwedge t. \llbracket \bigwedge u. \text{sseq } t u \implies \text{seq } t u; \text{sseq } \lambda[t] u \rrbracket \implies \text{seq } \lambda[t] u$

**using** *seq-char* **by** (*cases*  $u$ ) *auto*

**show**  $\bigwedge t1 t2. \llbracket \bigwedge u. \text{sseq } t1 u \implies \text{seq } t1 u; \bigwedge u. \text{sseq } t2 u \implies \text{seq } t2 u; \text{sseq } (\lambda[t1] \bullet t2) u \rrbracket$

$\implies \text{seq } (\lambda[t1] \bullet t2) u$

$\implies \text{seq } (\lambda[t1] \bullet t2) u$

**using** *seq-char* *Ide-implies-Arr*

**by** (*cases*  $u$ ) *auto*

**fix**  $t1 t2$

**show**  $\llbracket \bigwedge u. \text{sseq } t1 u \implies \text{seq } t1 u; \bigwedge u. \text{sseq } t2 u \implies \text{seq } t2 u; \text{seq } (t1 \circ t2) u \rrbracket$

$\implies \text{seq } (t1 \circ t2) u$

**proof** -

**assume** *ind1*:  $\bigwedge u. \text{sseq } t1 u \implies \text{seq } t1 u$

**assume** *ind2*:  $\bigwedge u. \text{sseq } t2 u \implies \text{seq } t2 u$

**assume** *1*: *sseq*  $(t1 \circ t2) u$

**show** *?thesis*

**using** *1 ind1 ind2 seq-char arr-char elementary-reduction-is-arr*

*Ide-Src Ide-Trg Ide-implies-Arr Coinitial-iff-Con resid-Arr-self*

```

apply (cases u, simp-all)
  apply (cases t1, simp-all)
  apply (cases t1, simp-all)
apply (cases Ide t1; cases Ide t2)
  apply simp-all
  apply (metis Ide-iff-Src-self Ide-iff-Trg-self)
  apply (metis Ide-iff-Src-self Ide-iff-Trg-self)
apply (metis Ide-iff-Trg-self Src-Trg)
by (cases t1) auto
qed
qed

lemma sseq-imp-elementary-reduction1:
shows sseq t u  $\implies$  elementary-reduction t
proof (induct u arbitrary: t)
  show  $\bigwedge t$ . sseq t  $\# \implies$  elementary-reduction t
  by simp
  show  $\bigwedge x t$ . sseq t  $\langle\langle x \rangle\rangle \implies$  elementary-reduction t
  using elementary-reduction.simps(2) sseq.elims(1) by blast
  show  $\bigwedge u$ .  $\llbracket \bigwedge t$ . sseq t u  $\implies$  elementary-reduction t; sseq t  $\lambda[u] \rrbracket$ 
     $\implies$  elementary-reduction t for t
  using seq-cases sseq-imp-seq
  apply (cases t, simp-all)
  by force
  show  $\bigwedge u1 u2$ .  $\llbracket \bigwedge t$ . sseq t u1  $\implies$  elementary-reduction t;
     $\bigwedge t$ . sseq t u2  $\implies$  elementary-reduction t;
    sseq t (u1  $\circ$  u2)  $\rrbracket$ 
     $\implies$  elementary-reduction t for t
  using seq-cases sseq-imp-seq Ide-Src elementary-reduction-is-arr
  apply (cases t, simp-all)
  by blast
  show  $\bigwedge u1 u2$ .
     $\llbracket \bigwedge t$ . sseq t u1  $\implies$  elementary-reduction t;  $\bigwedge t$ . sseq t u2  $\implies$  elementary-reduction t;
    sseq t ( $\lambda[u1] \bullet u2$ )  $\rrbracket$ 
     $\implies$  elementary-reduction t for t
  using seq-cases sseq-imp-seq
  apply (cases t, simp-all)
  by fastforce
qed
qed

lemma sseq-imp-elementary-reduction2:
shows sseq t u  $\implies$  elementary-reduction u
proof (induct u arbitrary: t)
  show  $\bigwedge t$ . sseq t  $\# \implies$  elementary-reduction  $\#$ 
  by simp
  show  $\bigwedge x t$ . sseq t  $\langle\langle x \rangle\rangle \implies$  elementary-reduction  $\langle\langle x \rangle\rangle$ 
  using elementary-reduction.simps(2) sseq.elims(1) by blast
  show  $\bigwedge u$ .  $\llbracket \bigwedge t$ . sseq t u  $\implies$  elementary-reduction u; sseq t  $\lambda[u] \rrbracket$ 
     $\implies$  elementary-reduction  $\lambda[u]$  for t

```

```

using seq-cases sseq-imp-seq
apply (cases t, simp-all)
by force
show  $\bigwedge u1\ u2. [\bigwedge t. sseq\ t\ u1 \implies elementary\text{-reduction}\ u1;$ 
 $\bigwedge t. sseq\ t\ u2 \implies elementary\text{-reduction}\ u2;$ 
 $sseq\ t\ (u1\ \circ\ u2)]$ 
 $\implies elementary\text{-reduction}\ (u1\ \circ\ u2)$  for t
using seq-cases sseq-imp-seq Ide-Trg elementary-reduction-is-arr
by (cases t) auto
show  $\bigwedge u1\ u2. [\bigwedge t. sseq\ t\ u1 \implies elementary\text{-reduction}\ u1;$ 
 $\bigwedge t. sseq\ t\ u2 \implies elementary\text{-reduction}\ u2;$ 
 $sseq\ t\ (\lambda[u1] \bullet u2)]$ 
 $\implies elementary\text{-reduction}\ (\lambda[u1] \bullet u2)$  for t
using seq-cases sseq-imp-seq
apply (cases t, simp-all)
by fastforce
qed

```

**lemma** *sseq-Beta*:  
**shows**  $sseq\ (\lambda[t] \bullet u)\ v \longleftrightarrow Ide\ t \wedge Ide\ u \wedge elementary\text{-reduction}\ v \wedge seq\ (\lambda[t] \bullet u)\ v$   
**by** (cases v) auto

**lemma** *sseq-BetaI* [intro]:  
**assumes** *Ide t and Ide u and elementary-reduction v and seq ( $\lambda[t] \bullet u$ ) v*  
**shows**  $sseq\ (\lambda[t] \bullet u)\ v$   
**using** *assms sseq-Beta by simp*

A head reduction is standardly sequential with any elementary reduction that can be performed after it.

**lemma** *sseq-head-reductionI*:  
**shows**  $[[is\text{-head-reduction}\ t; elementary\text{-reduction}\ u; seq\ t\ u]] \implies sseq\ t\ u$   
**proof** (induct t arbitrary: u)  
**show**  $\bigwedge u. [[is\text{-head-reduction}\ \#\!; elementary\text{-reduction}\ u; seq\ \#\! u]] \implies sseq\ \#\! u$   
**by** simp  
**show**  $\bigwedge x\ u. [[is\text{-head-reduction}\ \langle\!x\!\rangle; elementary\text{-reduction}\ u; seq\ \langle\!x\!\rangle\ u]] \implies sseq\ \langle\!x\!\rangle\ u$   
**by** auto  
**show**  $\bigwedge t. [\bigwedge u. [[is\text{-head-reduction}\ t; elementary\text{-reduction}\ u; seq\ t\ u]] \implies sseq\ t\ u;$   
 $is\text{-head-reduction}\ \lambda[t]; elementary\text{-reduction}\ u; seq\ \lambda[t]\ u]$   
 $\implies sseq\ \lambda[t]\ u$  **for** u  
**by** (cases u) auto  
**show**  $\bigwedge t2. [\bigwedge u. [[is\text{-head-reduction}\ t1; elementary\text{-reduction}\ u; seq\ t1\ u]] \implies sseq\ t1\ u;$   
 $\bigwedge u. [[is\text{-head-reduction}\ t2; elementary\text{-reduction}\ u; seq\ t2\ u]] \implies sseq\ t2\ u;$   
 $is\text{-head-reduction}\ (t1\ \circ\ t2); elementary\text{-reduction}\ u; seq\ (t1\ \circ\ t2)\ u]$   
 $\implies sseq\ (t1\ \circ\ t2)\ u$  **for** t1 u  
**using** seq-char  
**apply** (cases u)  
**apply** simp-all  
**apply** (metis ArrE Ide-iff-Src-self Ide-iff-Trg-self App-Var-contains-no-head-reduction  
is-head-reduction-char is-head-reduction-imp-contains-head-reduction)

```

    is-head-reduction.simps(3,6-7))
  by (cases t1) auto
  show  $\bigwedge t1\ t2\ u. \llbracket \bigwedge u. \llbracket is-head-reduction\ t1; elementary-reduction\ u; seq\ t1\ u \rrbracket \implies sseq\ t1\ u; \bigwedge u. \llbracket is-head-reduction\ t2; elementary-reduction\ u; seq\ t2\ u \rrbracket \implies sseq\ t2\ u; is-head-reduction\ (\lambda[t1] \bullet t2); elementary-reduction\ u; seq\ (\lambda[t1] \bullet t2)\ u \rrbracket \implies sseq\ (\lambda[t1] \bullet t2)\ u$ 
  by auto
qed

```

Once a head reduction is skipped in an application, then all terms that follow it in a standard reduction path are also applications that do not contain head reductions.

**lemma** *sseq-preserves-App-and-no-head-reduction*:

**shows**  $\llbracket sseq\ t\ u; is-App\ t \wedge \neg\ contains-head-reduction\ t \rrbracket \implies is-App\ u \wedge \neg\ contains-head-reduction\ u$

**apply** (*induct t arbitrary: u*)

**apply** *simp-all*

**proof** –

**fix** *t1 t2 u*

**assume** *ind1*:  $\bigwedge u. \llbracket sseq\ t1\ u; is-App\ t1 \wedge \neg\ contains-head-reduction\ t1 \rrbracket \implies is-App\ u \wedge \neg\ contains-head-reduction\ u$

**assume** *ind2*:  $\bigwedge u. \llbracket sseq\ t2\ u; is-App\ t2 \wedge \neg\ contains-head-reduction\ t2 \rrbracket \implies is-App\ u \wedge \neg\ contains-head-reduction\ u$

**assume** *sseq*:  $sseq\ (t1 \circ t2)\ u$

**assume** *t*:  $\neg\ contains-head-reduction\ (t1 \circ t2)$

**have** *u*:  $\neg\ is-Beta\ u$

**using** *sseq t sseq-imp-seq seq-cases*

**by** (*cases t1; cases u*) *auto*

**have** *1*: *is-App u*

**using** *u sseq sseq-imp-seq*

**apply** (*cases u*)

**apply** *simp-all*

**by** *fastforce+*

**moreover** **have**  $\neg\ contains-head-reduction\ u$

**proof** (*cases u*)

**show**  $\bigwedge v. u = \lambda[v] \implies \neg\ contains-head-reduction\ u$

**using** *1* **by** *auto*

**show**  $\bigwedge v\ w. u = \lambda[v] \bullet w \implies \neg\ contains-head-reduction\ u$

**using** *u* **by** *auto*

**fix** *u1 u2*

**assume** *u*:  $u = u1 \circ u2$

**have** *1*:  $(sseq\ t1\ u1 \wedge Ide\ t2 \wedge t2 = u2) \vee (Ide\ t1 \wedge t1 = u1 \wedge sseq\ t2\ u2) \vee (elementary-reduction\ t1 \wedge u1 = Trg\ t1 \wedge t2 = Src\ u2 \wedge elementary-reduction\ u2)$

**using** *sseq u* **by** *force*

**moreover** **have**  $Ide\ t1 \wedge t1 = u1 \wedge sseq\ t2\ u2 \implies ?thesis$

**using** *Ide-implies-Arr ide-char sseq-imp-seq t u* **by** *fastforce*

**moreover** **have**  $elementary-reduction\ t1 \wedge u1 = Trg\ t1 \wedge t2 = Src\ u2 \wedge elementary-reduction\ u2$

$\implies ?thesis$

**proof** –

```

assume 2: elementary-reduction t1 ∧ u1 = Trg t1 ∧ t2 = Src u2 ∧
        elementary-reduction u2
have contains-head-reduction u ⇒ contains-head-reduction u1
    using u
    apply simp
    using contains-head-reduction.elims(2) by fastforce
hence contains-head-reduction u ⇒ ¬ Ide u1
    using contains-head-reduction-iff
    by (metis Coinitial-iff-Con Ide-iff-Src-self Ide-implies-Arr ide-char resid-Arr-Src
            subs-head-redex subs-implies-prfx)
thus ?thesis
    using 2
    by (metis Arr.simps(4) Ide-Trg seq-char sseq sseq-imp-imp-imp)
qed
moreover have sseq t1 u1 ∧ Ide t2 ∧ t2 = u2 ⇒ ?thesis
    using t u ind1 [of u1] Ide-implies-Arr sseq-imp-elementary-reduction1
    apply (cases t1, simp-all)
    using elementary-reduction.simps(1)
    apply blast
    using elementary-reduction.simps(2)
    apply blast
    using contains-head-reduction.elims(2)
    apply fastforce
    apply (metis contains-head-reduction.simps(6) is-App-def)
    using sseq-Beta by blast
ultimately show ?thesis by blast
qed auto
ultimately show is-App u ∧ ¬ contains-head-reduction u
    by blast
qed

end

```

## Standard Reduction Paths

```

context reduction-paths
begin

```

A *standard reduction path* is an elementary reduction path in which successive reductions are standardly sequential.

```

fun Std
where Std [] = True
    | Std [t] =  $\Lambda$ .elementary-reduction t
    | Std (t # U) = ( $\Lambda$ .sseq t (hd U) ∧ Std U)

```

```

lemma Std-consE [elim]:
assumes Std (t # U)
and [ $\Lambda$ .Arr t; U ≠ [] ⇒  $\Lambda$ .sseq t (hd U); Std U] ⇒ thesis
shows thesis

```

```

using assms
by (metis  $\Lambda$ .arr-char  $\Lambda$ .elementary-reduction-is-arr  $\Lambda$ .seq-char  $\Lambda$ .sseq-imp-imp-imp
      list.exhaust-sel list.sel(1) Std.simps(1-3))

lemma Std-imp-Arr [simp]:
shows  $\llbracket \text{Std } T; T \neq [] \rrbracket \Longrightarrow \text{Arr } T$ 
proof (induct T)
  show  $[] \neq [] \Longrightarrow \text{Arr } []$ 
    by simp
  fix t U
  assume ind:  $\llbracket \text{Std } U; U \neq [] \rrbracket \Longrightarrow \text{Arr } U$ 
  assume tU: Std (t # U)
  show Arr (t # U)
  proof (cases  $U = []$ )
    show  $U = [] \Longrightarrow \text{Arr } (t \# U)$ 
      using  $\Lambda$ .elementary-reduction-is-arr tU  $\Lambda$ .Ide-implies-Arr Std.simps(2) Arr.simps(2)
      by blast
    assume  $U: U \neq []$ 
    show Arr (t # U)
    proof –
      have  $\Lambda$ .sseq t (hd U)
        using tU U
        by (metis list.exhaust-sel reduction-paths.Std.simps(3))
      thus ?thesis
        using U ind  $\Lambda$ .sseq-imp-imp-imp
        apply auto
        using reduction-paths.Std.elims(3) tU
        by fastforce
    qed
  qed
qed

lemma Std-imp-sseq-last-hd:
shows  $\llbracket \text{Std } (T @ U); T \neq []; U \neq [] \rrbracket \Longrightarrow \Lambda$ .sseq (last T) (hd U)
  apply (induct T arbitrary: U)
  apply simp-all
  by (metis Std.elims(3) Std.simps(3) append-self-conv2 neq-Nil-conv)

lemma Std-implies-set-subset-elementary-reduction:
shows Std U  $\Longrightarrow$  set U  $\subseteq$  Collect  $\Lambda$ .elementary-reduction
  apply (induct U)
  apply auto
  by (metis Std.simps(2) Std.simps(3) neq-Nil-conv  $\Lambda$ .sseq-imp-elementary-reduction1)

lemma Std-map-Lam:
shows Std T  $\Longrightarrow$  Std (map  $\Lambda$ .Lam T)
proof (induct T)
  show Std  $[] \Longrightarrow$  Std (map  $\Lambda$ .Lam [])
    by simp

```

```

fix t U
assume ind: Std U  $\implies$  Std (map  $\Lambda$ .Lam U)
assume tU: Std (t # U)
have Std (map  $\Lambda$ .Lam (t # U))  $\longleftrightarrow$  Std ( $\lambda[t]$  # map  $\Lambda$ .Lam U)
  by auto
also have ... = True
  apply (cases U = [])
  apply simp-all
  using Arr.simps(3) Std.simps(2) arr-char tU
  apply presburger
proof -
  assume U: U  $\neq$  []
  have Std ( $\lambda[t]$  # map  $\Lambda$ .Lam U)  $\longleftrightarrow$   $\Lambda$ .sseq  $\lambda[t]$   $\lambda[hd\ U]$   $\wedge$  Std (map  $\Lambda$ .Lam U)
    using U
    by (metis Nil-is-map-conv Std.simps(3) hd-map list.exhaust-sel)
  also have ...  $\longleftrightarrow$   $\Lambda$ .sseq t (hd U)  $\wedge$  Std (map  $\Lambda$ .Lam U)
    by auto
  also have ... = True
    using ind tU U
    by (metis Std.simps(3) list.exhaust-sel)
  finally show Std ( $\lambda[t]$  # map  $\Lambda$ .Lam U) by blast
qed
finally show Std (map  $\Lambda$ .Lam (t # U)) by blast
qed

```

```

lemma Std-map-App1:
shows [ $\Lambda$ .Ide b; Std T]  $\implies$  Std (map ( $\lambda X$ . X  $\circ$  b) T)
proof (induct T)
  show [ $\Lambda$ .Ide b; Std []]  $\implies$  Std (map ( $\lambda X$ . X  $\circ$  b) [])
    by simp
  fix t U
  assume ind: [ $\Lambda$ .Ide b; Std U]  $\implies$  Std (map ( $\lambda X$ . X  $\circ$  b) U)
  assume b:  $\Lambda$ .Ide b
  assume tU: Std (t # U)
  show Std (map ( $\lambda v$ . v  $\circ$  b) (t # U))
  proof (cases U = [])
    show U = []  $\implies$  ?thesis
      using Ide-implies-Arr b  $\Lambda$ .arr-char tU by force
    assume U: U  $\neq$  []
    have Std (map ( $\lambda v$ . v  $\circ$  b) (t # U)) = Std ((t  $\circ$  b) # map ( $\lambda X$ . X  $\circ$  b) U)
      by simp
    also have ... = ( $\Lambda$ .sseq (t  $\circ$  b) (hd U  $\circ$  b)  $\wedge$  Std (map ( $\lambda X$ . X  $\circ$  b) U))
      using U reduction-paths.Std.simps(3) hd-map
      by (metis Nil-is-map-conv neq-Nil-conv)
    also have ... = True
      using b tU U ind
      by (metis Std.simps(3) list.exhaust-sel  $\Lambda$ .sseq.simps(4))
    finally show Std (map ( $\lambda v$ . v  $\circ$  b) (t # U)) by blast
  qed

```

qed

**lemma** *Std-map-App2*:

**shows**  $\llbracket \Lambda.\text{Ide } a; \text{Std } T \rrbracket \Longrightarrow \text{Std } (\text{map } (\lambda u. a \circ u) T)$

**proof** (*induct*  $T$ )

**show**  $\llbracket \Lambda.\text{Ide } a; \text{Std } [] \rrbracket \Longrightarrow \text{Std } (\text{map } (\lambda u. a \circ u) [])$

**by** *simp*

**fix**  $t U$

**assume** *ind*:  $\llbracket \Lambda.\text{Ide } a; \text{Std } U \rrbracket \Longrightarrow \text{Std } (\text{map } (\lambda u. a \circ u) U)$

**assume**  $a: \Lambda.\text{Ide } a$

**assume**  $tU: \text{Std } (t \# U)$

**show**  $\text{Std } (\text{map } (\lambda u. a \circ u) (t \# U))$

**proof** (*cases*  $U = []$ )

**show**  $U = [] \Longrightarrow ?thesis$

**using**  $a tU$  **by** *force*

**assume**  $U: U \neq []$

**have**  $\text{Std } (\text{map } (\lambda u. a \circ u) (t \# U)) = \text{Std } ((a \circ t) \# \text{map } (\lambda u. a \circ u) U)$

**by** *simp*

**also have**  $\dots = (\Lambda.\text{sseq } (a \circ t) (a \circ \text{hd } U) \wedge \text{Std } (\text{map } (\lambda u. a \circ u) U))$

**using**  $U$

**by** (*metis Nil-is-map-conv Std.simps(3) hd-map list.exhaust-sel*)

**also have**  $\dots = \text{True}$

**using**  $a tU U$  *ind*

**by** (*metis Std.simps(3) list.exhaust-sel  $\Lambda.\text{sseq.simps(4)}$* )

**finally show**  $\text{Std } (\text{map } (\lambda u. a \circ u) (t \# U))$  **by** *blast*

qed

qed

**lemma** *Std-map-un-Lam*:

**shows**  $\llbracket \text{Std } T; \text{set } T \subseteq \text{Collect } \Lambda.\text{is-Lam} \rrbracket \Longrightarrow \text{Std } (\text{map } \Lambda.\text{un-Lam } T)$

**proof** (*induct*  $T$ )

**show**  $\llbracket \text{Std } []; \text{set } [] \subseteq \text{Collect } \Lambda.\text{is-Lam} \rrbracket \Longrightarrow \text{Std } (\text{map } \Lambda.\text{un-Lam } [])$

**by** *simp*

**fix**  $t T$

**assume** *ind*:  $\llbracket \text{Std } T; \text{set } T \subseteq \text{Collect } \Lambda.\text{is-Lam} \rrbracket \Longrightarrow \text{Std } (\text{map } \Lambda.\text{un-Lam } T)$

**assume**  $tT: \text{Std } (t \# T)$

**assume**  $1: \text{set } (t \# T) \subseteq \text{Collect } \Lambda.\text{is-Lam}$

**show**  $\text{Std } (\text{map } \Lambda.\text{un-Lam } (t \# T))$

**proof** (*cases*  $T = []$ )

**show**  $T = [] \Longrightarrow \text{Std } (\text{map } \Lambda.\text{un-Lam } (t \# T))$

**by** (*metis 1 Std.simps(2)  $\Lambda.\text{elementary-reduction.simps(3)}$   $\Lambda.\text{lambda.collapse(2)}$* )

*list.set-intros(1) list.simps(8) list.simps(9) mem-Collect-eq subset-code(1) tT*)

**assume**  $T: T \neq []$

**show**  $\text{Std } (\text{map } \Lambda.\text{un-Lam } (t \# T))$

**using**  $T tT 1$  *ind Std.simps(3) [of  $\Lambda.\text{un-Lam } t \Lambda.\text{un-Lam } (\text{hd } T) \text{map } \Lambda.\text{un-Lam } (\text{tl } T)$ ]*

**by** (*metis  $\Lambda.\text{lambda.collapse(2)}$   $\Lambda.\text{sseq.simps(3)}$  list.exhaust-sel list.sel(1)*)

*list.set-intros(1) map-eq-Cons-conv mem-Collect-eq reduction-paths.Std.simps(3)*

*set-subset-Cons subset-code(1)*)

qed

qed

**lemma** *Std-append-single*:

**shows**  $\llbracket \text{Std } T; T \neq []; \Lambda.\text{sseq } (\text{last } T) u \rrbracket \implies \text{Std } (T @ [u])$

**proof** (*induct*  $T$ )

**show**  $\llbracket \text{Std } []; [] \neq []; \Lambda.\text{sseq } (\text{last } []) u \rrbracket \implies \text{Std } ([] @ [u])$

**by** *blast*

**fix**  $t T$

**assume**  $\text{ind}$ :  $\llbracket \text{Std } T; T \neq []; \Lambda.\text{sseq } (\text{last } T) u \rrbracket \implies \text{Std } (T @ [u])$

**assume**  $tT$ :  $\text{Std } (t \# T)$

**assume**  $\text{sseq}$ :  $\Lambda.\text{sseq } (\text{last } (t \# T)) u$

**have**  $\text{Std } (t \# (T @ [u]))$

**using**  $\Lambda.\text{sseq-imp-elementary-reduction2 sseq ind tT}$

**apply** (*cases*  $T = []$ )

**apply** *simp*

**by** (*metis append-Cons last-ConsR list.sel(1) neq-Nil-conv reduction-paths.Std.simps(3)*)

**thus**  $\text{Std } ((t \# T) @ [u])$  **by** *simp*

qed

**lemma** *Std-append*:

**shows**  $\llbracket \text{Std } T; \text{Std } U; T = [] \vee U = [] \vee \Lambda.\text{sseq } (\text{last } T) (\text{hd } U) \rrbracket \implies \text{Std } (T @ U)$

**proof** (*induct*  $U$  *arbitrary*:  $T$ )

**show**  $\bigwedge T. \llbracket \text{Std } T; \text{Std } []; T = [] \vee [] = [] \vee \Lambda.\text{sseq } (\text{last } T) (\text{hd } []) \rrbracket \implies \text{Std } (T @ [])$

**by** *simp*

**fix**  $u T U$

**assume**  $\text{ind}$ :  $\bigwedge T. \llbracket \text{Std } T; \text{Std } U; T = [] \vee U = [] \vee \Lambda.\text{sseq } (\text{last } T) (\text{hd } U) \rrbracket$   
 $\implies \text{Std } (T @ U)$

**assume**  $T$ :  $\text{Std } T$

**assume**  $uU$ :  $\text{Std } (u \# U)$

**have**  $U$ :  $\text{Std } U$

**using**  $uU \text{Std.elims}(3)$  **by** *fastforce*

**assume**  $\text{seq}$ :  $T = [] \vee u \# U = [] \vee \Lambda.\text{sseq } (\text{last } T) (\text{hd } (u \# U))$

**show**  $\text{Std } (T @ (u \# U))$

**by** (*metis Std-append-single T U append.assoc append.left-neutral append-Cons ind last-snoc list.distinct(1) list.exhaust-sel list.sel(1) Std.simps(3) seq uU*)

qed

## Projections of Standard ‘App Paths’

Given a standard reduction path, all of whose transitions have *App* as their top-level constructor, we can apply *un-App1* or *un-App2* to each transition to project the path onto paths formed from the “rator” and the “rand” of each application. These projected paths are not standard, since the projection operation will introduce identities, in general. However, in this section we show that if we remove the identities, then in fact we do obtain standard reduction paths.

**abbreviation** *notIde*

**where**  $\text{notIde} \equiv \lambda u. \neg \Lambda.\text{Ide } u$

**lemma** *filter-notIde-Ide*:

**shows**  $\llbracket \text{Ide } U \rrbracket \implies \text{filter notIde } U = []$

**by** (*induct U*) *auto*

**lemma** *cong-filter-notIde*:

**shows**  $\llbracket \text{Arr } U; \neg \text{Ide } U \rrbracket \implies \text{filter notIde } U \text{ }^*\sim^* U$

**proof** (*induct U*)

**show**  $\llbracket \text{Arr } []; \neg \text{Ide } [] \rrbracket \implies \text{filter notIde } [] \text{ }^*\sim^* []$

**by** *simp*

**fix**  $u \ U$

**assume**  $\text{ind}: \llbracket \text{Arr } U; \neg \text{Ide } U \rrbracket \implies \text{filter notIde } U \text{ }^*\sim^* U$

**assume**  $\text{Arr}: \text{Arr } (u \# U)$

**assume**  $1: \neg \text{Ide } (u \# U)$

**show**  $\text{filter notIde } (u \# U) \text{ }^*\sim^* (u \# U)$

**proof** (*cases*  $\Lambda.\text{Ide } u$ )

**assume**  $u: \Lambda.\text{Ide } u$

**have**  $U: \text{Arr } U \wedge \neg \text{Ide } U$

**using**  $\text{Arr } u \ 1 \ \text{Ide.elims}(3)$  **by** *fastforce*

**have**  $\text{filter notIde } (u \# U) = \text{filter notIde } U$

**using**  $u$  **by** *simp*

**also have**  $\dots \text{ }^*\sim^* U$

**using**  $U \ \text{ind}$  **by** *blast*

**also have**  $U \text{ }^*\sim^* [u] @ U$

**using**  $u$

**by** (*metis* (*full-types*)  $\text{Arr Arr-has-Src Cons-eq-append-conv Ide.elims}(3) \ \text{Ide.simps}(2) \ \text{Srcs.simps}(1) \ U \ \text{arrI}_P \ \text{arr-append-imp-seq cong-append-ideI}(3) \ \text{ide-char} \ \Lambda.\text{ide-char not-Cons-self2}$ )

**also have**  $[u] @ U = u \# U$

**by** *simp*

**finally show** *?thesis* **by** *blast*

**next**

**assume**  $u: \neg \Lambda.\text{Ide } u$

**show** *?thesis*

**proof** (*cases*  $\text{Ide } U$ )

**assume**  $U: \text{Ide } U$

**have**  $\text{filter notIde } (u \# U) = [u]$

**using**  $u \ U \ \text{filter-notIde-Ide}$  **by** *simp*

**moreover have**  $[u] \text{ }^*\sim^* [u] @ U$

**using**  $u \ U \ \text{cong-append-ideI}(4)$  [*of*  $[u] \ U$ ]

**by** (*metis*  $\text{Arr Con-Arr-self Cons-eq-appendI Resid-Ide}(1) \ \text{arr-append-imp-seq} \ \text{arr-char ide-char ide-implies-arr neq-Nil-conv self-append-conv2}$ )

**moreover have**  $[u] @ U = u \# U$

**by** *simp*

**ultimately show** *?thesis* **by** *auto*

**next**

**assume**  $U: \neg \text{Ide } U$

**have**  $\text{filter notIde } (u \# U) = [u] @ \text{filter notIde } U$

**using**  $u \ U \ \text{Arr}$  **by** *simp*

**also have**  $\dots \text{ }^*\sim^* [u] @ U$

```

proof (cases U = [])
  show U = []  $\implies$  ?thesis
    by (metis Arr arr-char cong-reflexive append-Nil2 filter.simps(1))
  assume 1: U  $\neq$  []
  have seq [u] (filter notIde U)
    by (metis (full-types) 1 Arr Arr.simps(2-3) Con-imp-eq-Srcs Con-implies-Arr(1)
      Ide.elims(3) Ide.simps(1) Trgs.simps(2) U ide-char ind seq-char
      seq-implies-Trgs-eq-Srcs)
  thus ?thesis
    using u U Arr ind cong-append [of [u] filter notIde U [u] U]
    by (meson 1 Arr-consE cong-reflexive seqE)
qed
also have [u] @ U = u # U
  by simp
finally show ?thesis by argo
qed
qed
qed

```

**lemma** Std-filter-map-un-App1:

```

shows [[Std U; set U  $\subseteq$  Collect  $\Lambda$ .is-App]]  $\implies$  Std (filter notIde (map  $\Lambda$ .un-App1 U))
proof (induct U)
  show [[Std []; set []  $\subseteq$  Collect  $\Lambda$ .is-App]]  $\implies$  Std (filter notIde (map  $\Lambda$ .un-App1 []))
    by simp
  fix u U
  assume ind: [[Std U; set U  $\subseteq$  Collect  $\Lambda$ .is-App]]  $\implies$  Std (filter notIde (map  $\Lambda$ .un-App1 U))
  assume 1: Std (u # U)
  assume 2: set (u # U)  $\subseteq$  Collect  $\Lambda$ .is-App
  show Std (filter notIde (map  $\Lambda$ .un-App1 (u # U)))
    using 1 2 ind
    apply (cases u)
      apply simp-all
proof -
  fix u1 u2
  assume uU: Std ((u1  $\circ$  u2) # U)
  assume set: set U  $\subseteq$  Collect  $\Lambda$ .is-App
  assume ind: Std U  $\implies$  Std (filter notIde (map  $\Lambda$ .un-App1 U))
  assume u: u = u1  $\circ$  u2
  show ( $\neg$   $\Lambda$ .Ide u1  $\longrightarrow$  Std (u1 # filter notIde (map  $\Lambda$ .un-App1 U)))  $\wedge$ 
    ( $\Lambda$ .Ide u1  $\longrightarrow$  Std (filter notIde (map  $\Lambda$ .un-App1 U)))
  proof (intro conjI impI)
    assume u1:  $\Lambda$ .Ide u1
    show Std (filter notIde (map  $\Lambda$ .un-App1 U))
      by (metis 1 Std.simps(1) Std.simps(3) ind neq-Nil-conv)
    next
    assume u1:  $\neg$   $\Lambda$ .Ide u1
    show Std (u1 # filter notIde (map  $\Lambda$ .un-App1 U))
  proof (cases Ide (map  $\Lambda$ .un-App1 U))
    show Ide (map  $\Lambda$ .un-App1 U)  $\implies$  ?thesis

```

```

proof –
  assume  $U: \text{Ide } (\text{map } \Lambda.\text{un-App1 } U)$ 
  have  $\text{filter notIde } (\text{map } \Lambda.\text{un-App1 } U) = []$ 
    by ( $\text{metis } U \text{ Ide-char filter-False } \Lambda.\text{ide-char}$ 
       $\text{mem-Collect-eq subsetD}$ )
  thus  $?thesis$ 
    by ( $\text{metis Std.elims}(1) \text{ Std.simps}(2) \Lambda.\text{elementary-reduction.simps}(4) \text{ list.discI}$ 
       $\text{list.sel}(1) \Lambda.\text{sseq-imp-elementary-reduction1 } u1 \ uU$ )
qed
assume  $U: \neg \text{Ide } (\text{map } \Lambda.\text{un-App1 } U)$ 
show  $?thesis$ 
proof ( $\text{cases } U = []$ )
  show  $U = [] \implies ?thesis$ 
    using  $1 \ u \ u1$  by  $\text{fastforce}$ 
  assume  $U \neq []$ 
  hence  $U: U \neq [] \wedge \neg \text{Ide } (\text{map } \Lambda.\text{un-App1 } U)$ 
    using  $U$  by  $\text{simp}$ 
  have  $\Lambda.\text{sseq } u1 \ (\text{hd } (\text{filter notIde } (\text{map } \Lambda.\text{un-App1 } U)))$ 
proof –
  have  $\bigwedge u1 \ u2. \llbracket \text{set } U \subseteq \text{Collect } \Lambda.\text{is-App}; \neg \text{Ide } (\text{map } \Lambda.\text{un-App1 } U); U \neq [];$ 
     $\text{Std } ((u1 \circ u2) \# U); \neg \Lambda.\text{Ide } u1 \rrbracket$ 
     $\implies \Lambda.\text{sseq } u1 \ (\text{hd } (\text{filter notIde } (\text{map } \Lambda.\text{un-App1 } U)))$ 
    for  $U$ 
    apply ( $\text{induct } U$ )
    apply  $\text{simp-all}$ 
    apply ( $\text{intro conjI impI}$ )
proof –
  fix  $u \ U \ u1 \ u2$ 
  assume  $\text{ind}: \bigwedge u1 \ u2. \llbracket \neg \text{Ide } (\text{map } \Lambda.\text{un-App1 } U); U \neq [];$ 
     $\text{Std } ((u1 \circ u2) \# U); \neg \Lambda.\text{Ide } u1 \rrbracket$ 
     $\implies \Lambda.\text{sseq } u1 \ (\text{hd } (\text{filter notIde } (\text{map } \Lambda.\text{un-App1 } U)))$ 
  assume  $1: \Lambda.\text{is-App } u \wedge \text{set } U \subseteq \text{Collect } \Lambda.\text{is-App}$ 
  assume  $2: \neg \text{Ide } (\Lambda.\text{un-App1 } u \# \text{map } \Lambda.\text{un-App1 } U)$ 
  assume  $3: \Lambda.\text{sseq } (u1 \circ u2) \ u \wedge \text{Std } (u \# U)$ 
  show  $\neg \Lambda.\text{Ide } (\Lambda.\text{un-App1 } u) \implies \Lambda.\text{sseq } u1 \ (\Lambda.\text{un-App1 } u)$ 
    by ( $\text{metis } 1 \ 3 \ \Lambda.\text{Arr.simps}(4) \ \Lambda.\text{Ide-Trg } \Lambda.\text{lambda.collapse}(3) \ \Lambda.\text{seq-char}$ 
       $\Lambda.\text{sseq.simps}(4) \ \Lambda.\text{sseq-imp-imp}$ )
  assume  $4: \neg \Lambda.\text{Ide } u1$ 
  assume  $5: \Lambda.\text{Ide } (\Lambda.\text{un-App1 } u)$ 
  have  $u1: \Lambda.\text{elementary-reduction } u1$ 
    using  $3 \ 4 \ \Lambda.\text{elementary-reduction.simps}(4) \ \Lambda.\text{sseq-imp-elementary-reduction1}$ 
    by  $\text{blast}$ 
  have  $6: \text{Arr } (\Lambda.\text{un-App1 } u \# \text{map } \Lambda.\text{un-App1 } U)$ 
    using  $1 \ 3 \ \text{Std-imp-Arr } \text{Arr-map-un-App1 } [\text{of } u \# U]$  by  $\text{auto}$ 
  have  $7: \text{Arr } (\text{map } \Lambda.\text{un-App1 } U)$ 
    using  $1 \ 2 \ 3 \ 5 \ 6 \ \text{Arr-map-un-App1 } \text{Std-imp-Arr } \Lambda.\text{ide-char}$  by  $\text{fastforce}$ 
  have  $8: \neg \text{Ide } (\text{map } \Lambda.\text{un-App1 } U)$ 
    using  $2 \ 5 \ 6 \ \text{set-Ide-subset-ide}$  by  $\text{fastforce}$ 
  have  $9: \Lambda.\text{seq } u \ (\text{hd } U)$ 

```

by (*metis* 3 7 *Std.simps*(3) *Arr.simps*(1) *list.collapse* *list.simps*(8)  
 $\Lambda.sseq\text{-}imp\text{-}seq$ )  
**show**  $\Lambda.sseq\ u1\ (hd\ (filter\ notIde\ (map\ \Lambda.un\text{-}App1\ U)))$   
**proof** –  
 have  $\Lambda.sseq\ (u1\ \circ\ \Lambda.Trig\ (\Lambda.un\text{-}App2\ u))\ (hd\ U)$   
**proof** (*cases*  $\Lambda.Ide\ (\Lambda.un\text{-}App1\ (hd\ U))$ )  
 assume 10:  $\Lambda.Ide\ (\Lambda.un\text{-}App1\ (hd\ U))$   
 hence  $\Lambda.elementary\text{-}reduction\ (\Lambda.un\text{-}App2\ (hd\ U))$   
 by (*metis* (*full-types*) 1 3 7 *Std.elims*(2) *Arr.simps*(1)  
 $\Lambda.elementary\text{-}reduction\text{-}App\text{-}iff$   $\Lambda.elementary\text{-}reduction\text{-}not\text{-}ide$   
 $\Lambda.ide\text{-}char$  *list.sel*(2) *list.sel*(3) *list.set-sel*(1) *list.simps*(8)  
 $mem\text{-}Collect\text{-}eq\ \Lambda.sseq\text{-}imp\text{-}elementary\text{-}reduction2\ subsetD$ )  
**moreover** have  $\Lambda.Trig\ u1 = \Lambda.un\text{-}App1\ (hd\ U)$   
**proof** –  
 have  $\Lambda.Trig\ u1 = \Lambda.Src\ (\Lambda.un\text{-}App1\ u)$   
 by (*metis* 1 3 5  $\Lambda.Ide\text{-}iff\text{-}Src\text{-}self$   $\Lambda.Ide\text{-}implies\text{-}Arr$   $\Lambda.Trig\text{-}Src$   
 $\Lambda.elementary\text{-}reduction\text{-}not\text{-}ide$   $\Lambda.ide\text{-}char$   $\Lambda.lambda.collapse$ (3)  
 $\Lambda.sseq.simps$ (4)  $\Lambda.sseq\text{-}imp\text{-}elementary\text{-}reduction2$ )  
**also** have  $\dots = \Lambda.Trig\ (\Lambda.un\text{-}App1\ u)$   
 by (*metis* 5  $\Lambda.Ide\text{-}iff\text{-}Src\text{-}self$   $\Lambda.Ide\text{-}iff\text{-}Trig\text{-}self$   
 $\Lambda.Ide\text{-}implies\text{-}Arr$ )  
**also** have  $\dots = \Lambda.un\text{-}App1\ (hd\ U)$   
 using 1 3 5 7  $\Lambda.Ide\text{-}iff\text{-}Trig\text{-}self$   
 by (*metis* 9 10 *Arr.simps*(1)  $lambda\text{-}calculus.Ide\text{-}iff\text{-}Src\text{-}self$   
 $\Lambda.Ide\text{-}implies\text{-}Arr$   $\Lambda.Src\text{-}Src$   $\Lambda.Src\text{-}eq\text{-}iff$ (2)  $\Lambda.Trig.simps$ (3)  
 $\Lambda.lambda.collapse$ (3)  $\Lambda.seqE_{\Lambda}$  *list.set-sel*(1) *list.simps*(8)  
 $mem\text{-}Collect\text{-}eq\ subsetD$ )  
**finally** show *?thesis* by *argo*  
**qed**  
**moreover** have  $\Lambda.Trig\ (\Lambda.un\text{-}App2\ u) = \Lambda.Src\ (\Lambda.un\text{-}App2\ (hd\ U))$   
 by (*metis* 1 7 9 *Arr.simps*(1) *hd-in-set*  $\Lambda.Src.simps$ (4)  $\Lambda.Src\text{-}Src$   
 $\Lambda.Trig.simps$ (3)  $\Lambda.lambda.collapse$ (3)  $\Lambda.lambda.sel$ (4)  
 $\Lambda.seq\text{-}char$  *list.simps*(8)  $mem\text{-}Collect\text{-}eq\ subset\text{-}code$ (1))  
**ultimately** show *?thesis*  
 using  $\Lambda.sseq.simps$ (4)  
 by (*metis* 1 7 *u1* *Arr.simps*(1) *hd-in-set*  $\Lambda.lambda.collapse$ (3)  
 $list.simps$ (8)  $mem\text{-}Collect\text{-}eq\ subsetD$ )  
**next**  
**assume** 10:  $\neg\ \Lambda.Ide\ (\Lambda.un\text{-}App1\ (hd\ U))$   
**have** *False*  
**proof** –  
 have  $\Lambda.elementary\text{-}reduction\ (\Lambda.un\text{-}App2\ u)$   
 using 1 3 5  $\Lambda.elementary\text{-}reduction\text{-}App\text{-}iff$   
 $\Lambda.elementary\text{-}reduction\text{-}not\text{-}ide$   $\Lambda.sseq\text{-}imp\text{-}elementary\text{-}reduction2$   
 by *blast*  
**moreover** have  $\Lambda.sseq\ u\ (hd\ U)$   
 by (*metis* 3 7 *Std.simps*(3) *Arr.simps*(1)  
 $hd\text{-}Cons\text{-}tl$  *list.simps*(8))  
**moreover** have  $\Lambda.elementary\text{-}reduction\ (\Lambda.un\text{-}App1\ (hd\ U))$

```

    by (metis 1 7 10 Nil-is-map-conv Arr.simps(1)
        calculation(2)  $\Lambda$ .elementary-reduction-App-iff hd-in-set  $\Lambda$ .ide-char
        mem-Collect-eq  $\Lambda$ .sseq-imp-elementary-reduction2 subset-iff)
  ultimately show ?thesis
  using  $\Lambda$ .sseq.simps(4)
  by (metis 1 5 7 Arr.simps(1)  $\Lambda$ .elementary-reduction-not-ide
      hd-in-set  $\Lambda$ .ide-char  $\Lambda$ .lambda.collapse(3) list.simps(8)
      mem-Collect-eq subset-iff)
  qed
  thus ?thesis by argo
  qed
  hence Std ((u1  $\circ$   $\Lambda$ .Trg ( $\Lambda$ .un-App2 u)) # U)
  by (metis 3 7 Std.simps(3) Arr.simps(1) list.exhaust-sel list.simps(8))
  thus ?thesis
  using ind
  by (metis 7 8 u1 Arr.simps(1)  $\Lambda$ .elementary-reduction-not-ide  $\Lambda$ .ide-char
      list.simps(8))
  qed
  qed
  thus ?thesis
  using U set u1 uU by blast
  qed
  thus ?thesis
  by (metis 1 Std.simps(2-3)  $\langle U \neq [] \rangle$  ind list.exhaust-sel list.sel(1)
       $\Lambda$ .sseq-imp-elementary-reduction1)
  qed
  qed
  qed
  qed
  qed

```

**lemma** Std-filter-map-un-App2:

**shows**  $\llbracket \text{Std } U; \text{ set } U \subseteq \text{Collect } \Lambda.\text{is-App} \rrbracket \implies \text{Std } (\text{filter notIde } (\text{map } \Lambda.\text{un-App2 } U))$

**proof** (induct U)

**show**  $\llbracket \text{Std } []; \text{ set } [] \subseteq \text{Collect } \Lambda.\text{is-App} \rrbracket \implies \text{Std } (\text{filter notIde } (\text{map } \Lambda.\text{un-App2 } []))$

  by simp

  fix u U

**assume** ind:  $\llbracket \text{Std } U; \text{ set } U \subseteq \text{Collect } \Lambda.\text{is-App} \rrbracket \implies \text{Std } (\text{filter notIde } (\text{map } \Lambda.\text{un-App2 } U))$

**assume** 1: Std (u # U)

**assume** 2: set (u # U)  $\subseteq$  Collect  $\Lambda$ .is-App

**show** Std (filter notIde (map  $\Lambda$ .un-App2 (u # U)))

  using 1 2 ind

  apply (cases u)

    apply simp-all

**proof** –

  fix u1 u2

**assume** uU: Std ((u1  $\circ$  u2) # U)

**assume** set: set U  $\subseteq$  Collect  $\Lambda$ .is-App

**assume** ind: Std U  $\implies$  Std (filter notIde (map  $\Lambda$ .un-App2 U))

```

assume  $u: u = u1 \circ u2$ 
show  $(\neg \Lambda.Ide\ u2 \longrightarrow Std\ (u2 \# filter\ notIde\ (map\ \Lambda.un-App2\ U))) \wedge$ 
 $(\Lambda.Ide\ u2 \longrightarrow Std\ (filter\ notIde\ (map\ \Lambda.un-App2\ U)))$ 
proof  $(intro\ conjI\ impI)$ 
  assume  $u2: \Lambda.Ide\ u2$ 
  show  $Std\ (filter\ notIde\ (map\ \Lambda.un-App2\ U))$ 
    by  $(metis\ 1\ Std.simps(1)\ Std.simps(3)\ ind\ neq-Nil-conv)$ 
  next
  assume  $u2: \neg \Lambda.Ide\ u2$ 
  show  $Std\ (u2 \# filter\ notIde\ (map\ \Lambda.un-App2\ U))$ 
  proof  $(cases\ Ide\ (map\ \Lambda.un-App2\ U))$ 
    show  $Ide\ (map\ \Lambda.un-App2\ U) \implies ?thesis$ 
    proof  $-$ 
      assume  $U: Ide\ (map\ \Lambda.un-App2\ U)$ 
      have  $filter\ notIde\ (map\ \Lambda.un-App2\ U) = []$ 
        by  $(metis\ U\ Ide-char\ filter-False\ \Lambda.ide-char\ mem-Collect-eq\ subsetD)$ 
      thus  $?thesis$ 
      by  $(metis\ Std.elims(1)\ Std.simps(2)\ \Lambda.elementary-reduction.simps(4)\ list.discI$ 
 $list.sel(1)\ \Lambda.sseq-imp-elementary-reduction1\ u2\ uU)$ 
    qed
  assume  $U: \neg Ide\ (map\ \Lambda.un-App2\ U)$ 
  show  $?thesis$ 
  proof  $(cases\ U = [])$ 
    show  $U = [] \implies ?thesis$ 
      using  $1\ u\ u2$  by  $fastforce$ 
    assume  $U \neq []$ 
    hence  $U: U \neq [] \wedge \neg Ide\ (map\ \Lambda.un-App2\ U)$ 
      using  $U$  by  $simp$ 
    have  $\Lambda.sseq\ u2\ (hd\ (filter\ notIde\ (map\ \Lambda.un-App2\ U)))$ 
    proof  $-$ 
      have  $\bigwedge u1\ u2. \llbracket set\ U \subseteq Collect\ \Lambda.is-App; \neg Ide\ (map\ \Lambda.un-App2\ U); U \neq [];$ 
 $Std\ ((u1 \circ u2) \# U); \neg \Lambda.Ide\ u2 \rrbracket$ 
 $\implies \Lambda.sseq\ u2\ (hd\ (filter\ notIde\ (map\ \Lambda.un-App2\ U)))$ 
      for  $U$ 
      apply  $(induct\ U)$ 
      apply  $simp-all$ 
      apply  $(intro\ conjI\ impI)$ 
    proof  $-$ 
      fix  $u\ U\ u1\ u2$ 
      assume  $ind: \bigwedge u1\ u2. \llbracket \neg Ide\ (map\ \Lambda.un-App2\ U); U \neq [];$ 
 $Std\ ((u1 \circ u2) \# U); \neg \Lambda.Ide\ u2 \rrbracket$ 
 $\implies \Lambda.sseq\ u2\ (hd\ (filter\ notIde\ (map\ \Lambda.un-App2\ U)))$ 
      assume  $1: \Lambda.is-App\ u \wedge set\ U \subseteq Collect\ \Lambda.is-App$ 
      assume  $2: \neg Ide\ (\Lambda.un-App2\ u \# map\ \Lambda.un-App2\ U)$ 
      assume  $3: \Lambda.sseq\ (u1 \circ u2)\ u \wedge Std\ (u \# U)$ 
      assume  $4: \neg \Lambda.Ide\ u2$ 
      show  $\neg \Lambda.Ide\ (\Lambda.un-App2\ u) \implies \Lambda.sseq\ u2\ (\Lambda.un-App2\ u)$ 
        by  $(metis\ 1\ 3\ 4\ \Lambda.elementary-reduction.simps(4)$ 
 $\Lambda.elementary-reduction-not-ide\ \Lambda.ide-char\ \Lambda.lambda.collapse(3))$ 

```

```

       $\Lambda.sseq.simps(4)$   $\Lambda.sseq-imp-elementary-reduction1$ )
assume 5:  $\Lambda.Ide$  ( $\Lambda.un-App2$   $u$ )
have False
  by (metis 1 3 4 5  $\Lambda.elementary-reduction-not-ide$   $\Lambda.ide-char$ 
       $\Lambda.lambda.collapse(3)$   $\Lambda.sseq.simps(4)$   $\Lambda.sseq-imp-elementary-reduction2$ )
thus  $\Lambda.sseq$   $u2$  (hd (filter notIde (map  $\Lambda.un-App2$   $U$ ))) by argo
qed
thus ?thesis
  using  $U$  set  $u2$   $uU$  by blast
qed
thus ?thesis
  by (metis 1  $Std.simps(2)$   $Std.simps(3)$   $\langle U \neq [] \rangle$  ind list.exhaust-sel list.sel(1)
       $\Lambda.sseq-imp-elementary-reduction1$ )
qed
qed
qed
qed
qed

```

If the first step in a standard reduction path contracts a redex that is not at the head position, then all subsequent terms have *App* as their top-level operator.

```

lemma seq-App-Std-implies:
shows  $\llbracket Std$  ( $t \# U$ );  $\Lambda.is-App$   $t \wedge \neg \Lambda.contains-head-reduction$   $t \rrbracket$ 
   $\implies set$   $U \subseteq Collect$   $\Lambda.is-App$ 
proof (induct  $U$  arbitrary:  $t$ )
  show  $\bigwedge t. \llbracket Std$  [ $t$ ];  $\Lambda.is-App$   $t \wedge \neg \Lambda.contains-head-reduction$   $t \rrbracket$ 
     $\implies set$   $[] \subseteq Collect$   $\Lambda.is-App$ 
    by simp
fix  $t$   $u$   $U$ 
assume ind:  $\bigwedge t. \llbracket Std$  ( $t \# U$ );  $\Lambda.is-App$   $t \wedge \neg \Lambda.contains-head-reduction$   $t \rrbracket$ 
   $\implies set$   $U \subseteq Collect$   $\Lambda.is-App$ 
assume Std:  $Std$  ( $t \# u \# U$ )
assume  $t$ :  $\Lambda.is-App$   $t \wedge \neg \Lambda.contains-head-reduction$   $t$ 
have  $U$ :  $set$  ( $u \# U$ )  $\subseteq Collect$   $\Lambda.elementary-reduction$ 
  using Std Std-implies-set-subset-elementary-reduction by fastforce
have  $u$ :  $\Lambda.elementary-reduction$   $u$ 
  using  $U$  by simp
have  $set$   $U \subseteq Collect$   $\Lambda.elementary-reduction$ 
  using  $U$  by simp
show  $set$  ( $u \# U$ )  $\subseteq Collect$   $\Lambda.is-App$ 
proof (cases  $U = []$ )
  show  $U = [] \implies ?thesis$ 
    by (metis Std empty-set empty-subsetI insert-subset
         $\Lambda.sseq-preserved-App-and-no-head-reduction$  list.sel(1) list.simps(15)
        mem-Collect-eq reduction-paths.Std.simps(3)  $t$ )
assume  $U$ :  $U \neq []$ 
have  $\Lambda.sseq$   $t$   $u$ 
  using Std by auto
hence  $\Lambda.is-App$   $u \wedge \neg \Lambda.Ide$   $u \wedge \neg \Lambda.contains-head-reduction$   $u$ 

```

```

using  $t\ u\ U\ \Lambda.sseq\text{-preserves-App-and-no-head-reduction}$  [of  $t\ u$ ]
       $\Lambda.elementary\text{-reduction-not-ide}$ 
by blast
thus ?thesis
      using  $Std\ ind$  [of  $u$ ]  $\langle set\ U\ \subseteq\ Collect\ \Lambda.elementary\text{-reduction} \rangle$  by simp
qed
qed

```

### 3.6.2 Standard Developments

The following function takes a term  $t$  (representing a parallel reduction) and produces a standard reduction path that is a complete development of  $t$  and is thus congruent to  $[t]$ . The proof of termination makes use of the Finite Development Theorem.

```

function (sequential) standard-development
where standard-development  $\# = []$ 
      | standard-development  $\llcorner = []$ 
      | standard-development  $\lambda[t] = map\ \Lambda.Lam\ (standard-development\ t)$ 
      | standard-development  $(t\ \circ\ u) =$ 
        (if  $\Lambda.Arr\ t\ \wedge\ \Lambda.Arr\ u$  then
           $map\ (\lambda v.\ v\ \circ\ \Lambda.Src\ u)\ (standard-development\ t)\ @$ 
           $map\ (\lambda v.\ \Lambda.Trig\ t\ \circ\ v)\ (standard-development\ u)$ 
          else  $[]$ )
      | standard-development  $(\lambda[t] \bullet u) =$ 
        (if  $\Lambda.Arr\ t\ \wedge\ \Lambda.Arr\ u$  then
           $(\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u)\ \# standard-development\ (\Lambda.subst\ u\ t)$ 
          else  $[]$ )
by pat-completeness auto

```

**abbreviation** (*in lambda-calculus*) *stddev-term-rel*  
**where** *stddev-term-rel*  $\equiv mlex\text{-prod hgt subterm-rel}$

**lemma** (*in lambda-calculus*) *subst-lt-Beta*:

**assumes**  $Arr\ t$  **and**  $Arr\ u$

**shows**  $(subst\ u\ t,\ \lambda[t] \bullet u) \in stddev\text{-term-rel}$

**proof** –

**have**  $(\lambda[t] \bullet u) \setminus (\lambda[Src\ t] \bullet Src\ u) = subst\ u\ t$

**using** *assms*

**by** (*metis Arr-not-Nil Ide-Src Ide-iff-Src-self Ide-implies-Arr resid.simps(4)*  
*resid-Arr-Ide*)

**moreover have** *elementary-reduction*  $(\lambda[Src\ t] \bullet Src\ u)$

**by** (*simp add: assms Ide-Src*)

**moreover have**  $\lambda[Src\ t] \bullet Src\ u \sqsubseteq \lambda[t] \bullet u$

**by** (*metis assms Arr.simps(5) head-redex.simps(9) subs-head-redex*)

**ultimately show** *?thesis*

**using** *assms elementary-reduction-decreases-hgt* [of  $\lambda[Src\ t] \bullet Src\ u\ \lambda[t] \bullet u$ ]

**by** (*metis mlex-less*)

**qed**

**termination** *standard-development*

**proof** (relation  $\Lambda.stddev-term-rel$ )  
**show**  $wf \ \Lambda.stddev-term-rel$   
**using**  $\Lambda.wf-subterm-rel \ wf-mlex$  **by** *blast*  
**show**  $\bigwedge t. (t, \lambda[t]) \in \Lambda.stddev-term-rel$   
**by** (*simp add:  $\Lambda.subterm-lemmas(1) \ mlex-prod-def$* )  
**show**  $\bigwedge t \ u. (t, t \circ u) \in \Lambda.stddev-term-rel$   
**using**  $\Lambda.subterm-lemmas(3)$   
**by** (*metis antisym-conv1  $\Lambda.hgt.simps(4) \ le-add1 \ mem-Collect-eq \ mlex-iff \ old.prod.case$* )  
**show**  $\bigwedge t \ u. (u, t \circ u) \in \Lambda.stddev-term-rel$   
**using**  $\Lambda.subterm-lemmas(3)$  **by** (*simp add:  $mlex-leq$* )  
**show**  $\bigwedge t \ u. \Lambda.Arr \ t \wedge \Lambda.Arr \ u \implies (\Lambda.subst \ u \ t, \lambda[t] \bullet u) \in \Lambda.stddev-term-rel$   
**using**  $\Lambda.subst-lt-Beta$  **by** *simp*  
**qed**

**lemma** *Ide-iff-standard-development-empty*:  
**shows**  $\Lambda.Arr \ t \implies \Lambda.Ide \ t \longleftrightarrow standard-development \ t = []$   
**by** (*induct t*) *auto*

**lemma** *set-standard-development*:  
**shows**  $\Lambda.Arr \ t \longrightarrow set \ (standard-development \ t) \subseteq Collect \ \Lambda.elementary-reduction$   
**apply** (*rule standard-development.induct*)  
**using**  $\Lambda.Ide-Src \ \Lambda.Ide-Trg \ \Lambda.Arr-Subst$  **by** *auto*

**lemma** *cong-standard-development*:  
**shows**  $\Lambda.Arr \ t \wedge \neg \Lambda.Ide \ t \longrightarrow standard-development \ t \ \sim^* [t]$   
**proof** (*rule standard-development.induct*)  
**show**  $\Lambda.Arr \ \# \wedge \neg \Lambda.Ide \ \# \longrightarrow standard-development \ \# \ \sim^* [\#]$   
**by** *simp*  
**show**  $\bigwedge x. \Lambda.Arr \ \langle x \rangle \wedge \neg \Lambda.Ide \ \langle x \rangle \longrightarrow standard-development \ \langle x \rangle \ \sim^* [\langle x \rangle]$   
**by** *simp*  
**show**  $\bigwedge t. \Lambda.Arr \ t \wedge \neg \Lambda.Ide \ t \longrightarrow standard-development \ t \ \sim^* [t] \implies \Lambda.Arr \ \lambda[t] \wedge \neg \Lambda.Ide \ \lambda[t] \longrightarrow standard-development \ \lambda[t] \ \sim^* [\lambda[t]]$   
**by** (*metis (mono-tags, lifting) cong-map-Lam  $\Lambda.Arr.simps(3) \ \Lambda.Ide.simps(3) \ list.simps(8,9) \ standard-development.simps(3)$* )  
**show**  $\bigwedge t \ u. \llbracket \Lambda.Arr \ t \wedge \Lambda.Arr \ u \implies \Lambda.Arr \ t \wedge \neg \Lambda.Ide \ t \longrightarrow standard-development \ t \ \sim^* [t]; \ \Lambda.Arr \ t \wedge \Lambda.Arr \ u \implies \Lambda.Arr \ u \wedge \neg \Lambda.Ide \ u \longrightarrow standard-development \ u \ \sim^* [u] \rrbracket \implies \Lambda.Arr \ (t \circ u) \wedge \neg \Lambda.Ide \ (t \circ u) \longrightarrow standard-development \ (t \circ u) \ \sim^* [t \circ u]$

**proof**  
**fix**  $t \ u$   
**assume**  $ind1: \Lambda.Arr \ t \wedge \Lambda.Arr \ u \implies \Lambda.Arr \ t \wedge \neg \Lambda.Ide \ t \longrightarrow standard-development \ t \ \sim^* [t]$   
**assume**  $ind2: \Lambda.Arr \ t \wedge \Lambda.Arr \ u \implies \Lambda.Arr \ u \wedge \neg \Lambda.Ide \ u \longrightarrow standard-development \ u \ \sim^* [u]$   
**assume**  $1: \Lambda.Arr \ (t \circ u) \wedge \neg \Lambda.Ide \ (t \circ u)$   
**show**  $standard-development \ (t \circ u) \ \sim^* [t \circ u]$

```

proof (cases standard-development t = [])
  show standard-development t = []  $\implies$  ?thesis
    using 1 ind2 cong-map-App1 Ide-iff-standard-development-empty  $\Lambda$ .Ide-iff-Trg-self
    apply simp
    by (metis (no-types, opaque-lifting) list.simps(8,9))
assume t: standard-development t  $\neq$  []
show ?thesis
proof (cases standard-development u = [])
  assume u: standard-development u = []
  have standard-development (t  $\circ$  u) = map ( $\lambda X. X \circ u$ ) (standard-development t)
    using u 1  $\Lambda$ .Ide-iff-Src-self ide-char ind2 by auto
  also have ...  $\sim^*$  map ( $\lambda a. a \circ u$ ) [t]
    using cong-map-App2 [of u]
    by (meson 1  $\Lambda$ .Arr.simps(4) Ide-iff-standard-development-empty t u ind1)
  also have map ( $\lambda a. a \circ u$ ) [t] = [t  $\circ$  u]
    by simp
  finally show ?thesis by blast
next
assume u: standard-development u  $\neq$  []
have standard-development (t  $\circ$  u) =
  map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t) @
  map ( $\lambda b. \Lambda$ .Trg t  $\circ$  b) (standard-development u)
  using 1 by force
moreover have map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t)  $\sim^*$  [t  $\circ$   $\Lambda$ .Src u]
proof –
  have map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t)  $\sim^*$  map ( $\lambda a. a \circ \Lambda$ .Src u) [t]
    using t u 1 ind1  $\Lambda$ .Ide-Src Ide-iff-standard-development-empty cong-map-App2
    by (metis  $\Lambda$ .Arr.simps(4))
  also have map ( $\lambda a. a \circ \Lambda$ .Src u) [t] = [t  $\circ$   $\Lambda$ .Src u]
    by simp
  finally show ?thesis by blast
qed
moreover have map ( $\lambda b. \Lambda$ .Trg t  $\circ$  b) (standard-development u)  $\sim^*$  [ $\Lambda$ .Trg t  $\circ$  u]
  using t u 1 ind2  $\Lambda$ .Ide-Trg Ide-iff-standard-development-empty cong-map-App1
  by (metis (mono-tags, opaque-lifting)  $\Lambda$ .Arr.simps(4) list.simps(8,9))
moreover have seq (map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t))
  (map ( $\lambda b. \Lambda$ .Trg t  $\circ$  b) (standard-development u))
proof
  show Arr (map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t))
    by (metis Con-implies-Arr(1) Ide.simps(1) calculation(2) ide-char)
  show Arr (map (( $\circ$ ) ( $\Lambda$ .Trg t)) (standard-development u))
    by (metis Con-implies-Arr(1) Ide.simps(1) calculation(3) ide-char)
  show  $\Lambda$ .Trg (last (map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t))) =
   $\Lambda$ .Src (hd (map (( $\circ$ ) ( $\Lambda$ .Trg t)) (standard-development u)))
    using 1 Src-hd-eqI Trg-last-eqI calculation(2) calculation(3) by auto
qed
ultimately have standard-development (t  $\circ$  u)  $\sim^*$  [t  $\circ$   $\Lambda$ .Src u] @ [ $\Lambda$ .Trg t  $\circ$  u]
  using cong-append [of map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t)
  map ( $\lambda b. \Lambda$ .Trg t  $\circ$  b) (standard-development u)]

```

```

                                [t ◦ Λ.Src u] [Λ.Trig t ◦ u]]
  by simp
  moreover have [t ◦ Λ.Src u] @ [Λ.Trig t ◦ u] *~* [t ◦ u]
    using 1 Λ.Ide-Trig Λ.resid-Arr-Src Λ.resid-Arr-self Λ.null-char
          ide-char Λ.Arr-not-Nil
  by simp
  ultimately show ?thesis
    using cong-transitive by blast
qed
qed
qed
show  $\bigwedge t u. (\Lambda.Arr\ t \wedge \Lambda.Arr\ u \implies$ 
   $\Lambda.Arr\ (\Lambda.subst\ u\ t) \wedge \neg \Lambda.Ide\ (\Lambda.subst\ u\ t)$ 
   $\longrightarrow standard-development\ (\Lambda.subst\ u\ t) *~* [\Lambda.subst\ u\ t]) \implies$ 
   $\Lambda.Arr\ (\lambda[t] \bullet u) \wedge \neg \Lambda.Ide\ (\lambda[t] \bullet u) \longrightarrow$ 
   $standard-development\ (\lambda[t] \bullet u) *~* [\lambda[t] \bullet u]$ 
proof
  fix t u
  assume 1:  $\Lambda.Arr\ (\lambda[t] \bullet u) \wedge \neg \Lambda.Ide\ (\lambda[t] \bullet u)$ 
  assume ind:  $\Lambda.Arr\ t \wedge \Lambda.Arr\ u \implies$ 
     $\Lambda.Arr\ (\Lambda.subst\ u\ t) \wedge \neg \Lambda.Ide\ (\Lambda.subst\ u\ t)$ 
     $\longrightarrow standard-development\ (\Lambda.subst\ u\ t) *~* [\Lambda.subst\ u\ t]$ 
  show  $standard-development\ (\lambda[t] \bullet u) *~* [\lambda[t] \bullet u]$ 
  proof (cases  $\Lambda.Ide\ (\Lambda.subst\ u\ t)$ )
    assume 2:  $\Lambda.Ide\ (\Lambda.subst\ u\ t)$ 
    have  $standard-development\ (\lambda[t] \bullet u) = [\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u]$ 
      using 1 2 Ide-iff-standard-development-empty [of  $\Lambda.subst\ u\ t$ ]  $\Lambda.Arr-Subst$ 
      by simp
    also have  $[\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u] *~* [\lambda[t] \bullet u]$ 
      using 1 2  $\Lambda.Ide-Src\ \Lambda.Ide-implies-Arr\ ide-char\ \Lambda.resid-Arr-Ide$ 
      apply (intro conjI)
      apply simp-all
      apply (metis  $\Lambda.Ide.simps(1)\ \Lambda.Ide-Subst-iff\ \Lambda.Ide-Trig$ )
      by fastforce
    finally show ?thesis by blast
  next
    assume 2:  $\neg \Lambda.Ide\ (\Lambda.subst\ u\ t)$ 
    have  $standard-development\ (\lambda[t] \bullet u) =$ 
       $[\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u] @ standard-development\ (\Lambda.subst\ u\ t)$ 
      using 1 by auto
    also have  $[\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u] @ standard-development\ (\Lambda.subst\ u\ t) *~*$ 
       $[\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u] @ [\Lambda.subst\ u\ t]$ 
  proof (intro cong-append)
    show  $seq\ [\Lambda.Beta\ (\Lambda.Src\ t)\ (\Lambda.Src\ u)]\ (standard-development\ (\Lambda.subst\ u\ t))$ 
      using 1 2 ind arr-char ide-implies-arr  $\Lambda.Arr-Subst\ Con-implies-Arr(1)\ Src-hd-eqI$ 
      apply (intro seqIAP)
      apply simp-all
      by (metis  $Arr.simps(1)$ )
    show  $[\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u] *~* [\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u]$ 

```

**using** 1  
**by** (*metis*  $\Lambda$ .Arr.simps(5)  $\Lambda$ .Ide-*Src*  $\Lambda$ .Ide-*implies-Arr* Arr.simps(2) *Resid-Arr-self*  
*ide-char*  $\Lambda$ .arr-*char*)  
**show** *standard-development* ( $\Lambda$ .subst  $u$   $t$ )  $\sim^*$  [ $\Lambda$ .subst  $u$   $t$ ]  
**using** 1 2  $\Lambda$ .Arr-*Subst ind* **by** *simp*  
**qed**  
**also have** [ $\lambda$ [ $\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ] @ [ $\Lambda$ .subst  $u$   $t$ ]  $\sim^*$  [ $\lambda$ [ $t$ ]  $\bullet$   $u$ ]  
**proof**  
**show** [ $\lambda$ [ $\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ] @ [ $\Lambda$ .subst  $u$   $t$ ]  $\lesssim^*$  [ $\lambda$ [ $t$ ]  $\bullet$   $u$ ]  
**proof** –  
**have**  $t \setminus \Lambda$ .*Src*  $t \neq \# \wedge u \setminus \Lambda$ .*Src*  $u \neq \#$   
**by** (*metis* 1  $\Lambda$ .Arr.simps(5)  $\Lambda$ .*Coinitial-iff-Con*  $\Lambda$ .Ide-*Src*  $\Lambda$ .Ide-*iff-Src-self*  
 $\Lambda$ .Ide-*implies-Arr*)  
**moreover have**  $\Lambda$ .con ( $\lambda$ [ $\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ) ( $\lambda$ [ $t$ ]  $\bullet$   $u$ )  
**by** (*metis* 1  $\Lambda$ .head-*redex.simps*(9)  $\Lambda$ .*prfx-implies-con*  $\Lambda$ .*subs-head-redex*  
 $\Lambda$ .*subs-implies-prfx*)  
**ultimately have** ( $[\lambda[\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ] @ [ $\Lambda$ .subst  $u$   $t$ ])  $\setminus^*$  [ $\lambda$ [ $t$ ]  $\bullet$   $u$ ] =  
 $[\lambda[\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ]  $\setminus^*$  [ $\lambda$ [ $t$ ]  $\bullet$   $u$ ] @  
 $[\Lambda$ .subst  $u$   $t$ ]  $\setminus^*$  ( $[\lambda$ [ $t$ ]  $\bullet$   $u$ ]  $\setminus^*$  [ $\lambda$ [ $\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ])  
**using** *Resid-append*(1)  
 $[of$  [ $\lambda$ [ $\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ] [ $\Lambda$ .subst  $u$   $t$ ] [ $\lambda$ [ $t$ ]  $\bullet$   $u$ ]]  
**apply** *simp*  
**by** (*metis*  $\Lambda$ .Arr-*Subst*  $\Lambda$ .*Coinitial-iff-Con*  $\Lambda$ .Ide-*Src*  $\Lambda$ .*resid-Arr-Ide*)  
**also have** ... = [ $\Lambda$ .subst ( $\Lambda$ .*Trg*  $u$ ) ( $\Lambda$ .*Trg*  $t$ )] @ ( $[\Lambda$ .subst  $u$   $t$ ]  $\setminus^*$  [ $\Lambda$ .subst  $u$   $t$ ])  
**proof** –  
**have**  $t \setminus \Lambda$ .*Src*  $t \neq \# \wedge u \setminus \Lambda$ .*Src*  $u \neq \#$   
**by** (*metis* 1  $\Lambda$ .Arr.simps(5)  $\Lambda$ .*Coinitial-iff-Con*  $\Lambda$ .Ide-*Src*  
 $\Lambda$ .Ide-*iff-Src-self*  $\Lambda$ .Ide-*implies-Arr*)  
**moreover have**  $\Lambda$ .*Src*  $t \setminus t \neq \# \wedge \Lambda$ .*Src*  $u \setminus u \neq \#$   
**using**  $\Lambda$ .Con-*sym calculation*(1) **by** *presburger*  
**moreover have**  $\Lambda$ .con ( $\Lambda$ .subst  $u$   $t$ ) ( $\Lambda$ .subst  $u$   $t$ )  
**by** (*meson*  $\Lambda$ .Arr-*Subst*  $\Lambda$ .Con-*implies-Arr2*  $\Lambda$ .arr-*char*  $\Lambda$ .arr-*def calculation*(2))  
**moreover have**  $\Lambda$ .con ( $\lambda$ [ $t$ ]  $\bullet$   $u$ ) ( $\lambda$ [ $\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ )  
**using**  $\langle \Lambda$ .con ( $\lambda$ [ $\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ) ( $\lambda$ [ $t$ ]  $\bullet$   $u$ )  $\rangle$   $\Lambda$ .con-*sym* **by** *blast*  
**moreover have**  $\Lambda$ .con ( $\lambda$ [ $\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ) ( $\lambda$ [ $t$ ]  $\bullet$   $u$ )  
**using**  $\langle \Lambda$ .con ( $\lambda$ [ $\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ) ( $\lambda$ [ $t$ ]  $\bullet$   $u$ )  $\rangle$  **by** *blast*  
**moreover have**  $\Lambda$ .con ( $\Lambda$ .subst  $u$   $t$ ) ( $\Lambda$ .subst ( $u \setminus \Lambda$ .*Src*  $u$ ) ( $t \setminus \Lambda$ .*Src*  $t$ ))  
**by** (*metis*  $\Lambda$ .*Coinitial-iff-Con*  $\Lambda$ .Ide-*Src calculation*(1–3)  $\Lambda$ .*resid-Arr-Ide*)  
**ultimately show** *?thesis*  
**using** 1 **by** *auto*  
**qed**  
**finally have** ( $[\lambda[\Lambda$ .*Src*  $t$ ]  $\bullet$   $\Lambda$ .*Src*  $u$ ] @ [ $\Lambda$ .subst  $u$   $t$ ])  $\setminus^*$  [ $\lambda$ [ $t$ ]  $\bullet$   $u$ ] =  
 $[\Lambda$ .subst ( $\Lambda$ .*Trg*  $u$ ) ( $\Lambda$ .*Trg*  $t$ )] @ [ $\Lambda$ .subst  $u$   $t$ ]  $\setminus^*$  [ $\Lambda$ .subst  $u$   $t$ ]  
**by** *blast*  
**moreover have** *Ide* ...  
**by** (*metis* 1 2  $\Lambda$ .Arr.simps(5)  $\Lambda$ .Arr-*Subst*  $\Lambda$ .Ide-*Subst*  $\Lambda$ .Ide-*Trg*  
 $\Lambda$ .Nil-*is-append-conv* Arr-*append-iff*<sub>PWE</sub> Con-*implies-Arr*(2) *Ide.simps*(1–2)  
 $\Lambda$ .Ide-*append*<sub>PWE</sub> *Resid-Arr-self ide-char calculation*  $\Lambda$ .ide-*char ind*  
 $\Lambda$ .Con-*imp-Arr-Resid*)

**ultimately show** *?thesis*  
**using** *ide-char* **by** *presburger*  
**qed**  
**show**  $[\lambda[t] \bullet u] \approx^* [\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u] @ [\Lambda.subst\ u\ t]$   
**proof** –  
**have**  $[\lambda[t] \bullet u] \approx^* ([\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u] @ [\Lambda.subst\ u\ t]) =$   
 $([\lambda[t] \bullet u] \approx^* [\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u]) \approx^* [\Lambda.subst\ u\ t]$   
**by** *fastforce*  
**also have**  $\dots = [\Lambda.subst\ u\ t] \approx^* [\Lambda.subst\ u\ t]$   
**proof** –  
**have**  $t \setminus \Lambda.Src\ t \neq \# \wedge u \setminus \Lambda.Src\ u \neq \#$   
**by** (*metis 1*  $\Lambda.Arr.simps(5)$   $\Lambda.Coinitial\text{-}iff\text{-}Con$   $\Lambda.Ide\text{-}Src$   
 $\Lambda.Ide\text{-}iff\text{-}Src\text{-}self$   $\Lambda.Ide\text{-}implies\text{-}Arr$ )  
**moreover have**  $\Lambda.con\ (\Lambda.subst\ u\ t)\ (\Lambda.subst\ u\ t)$   
**by** (*metis 1*  $\Lambda.Arr.simps(5)$   $\Lambda.Arr\text{-}Subst$   $\Lambda.Coinitial\text{-}iff\text{-}Con$   
 $\Lambda.con\text{-}def$   $\Lambda.null\text{-}char$ )  
**moreover have**  $\Lambda.con\ (\lambda[t] \bullet u)\ (\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u)$   
**by** (*metis 1*  $\Lambda.Con\text{-}sym$   $\Lambda.con\text{-}def$   $\Lambda.head\text{-}redex.simps(9)$   $\Lambda.null\text{-}char$   
 $\Lambda.prfx\text{-}implies\text{-}con$   $\Lambda.subs\text{-}head\text{-}redex$   $\Lambda.subs\text{-}implies\text{-}prfx$ )  
**moreover have**  $\Lambda.con\ (\Lambda.subst\ (u \setminus \Lambda.Src\ u)\ (t \setminus \Lambda.Src\ t))\ (\Lambda.subst\ u\ t)$   
**by** (*metis*  $\Lambda.Coinitial\text{-}iff\text{-}Con$   $\Lambda.Ide\text{-}Src$  *calculation(1)* *calculation(2)*  
 $\Lambda.resid\text{-}Arr\text{-}Ide$ )  
**ultimately show** *?thesis*  
**using**  $\Lambda.resid\text{-}Arr\text{-}Ide$   
**apply** *simp*  
**by** (*metis*  $\Lambda.Coinitial\text{-}iff\text{-}Con$   $\Lambda.Ide\text{-}Src$ )  
**qed**  
**finally have**  $[\lambda[t] \bullet u] \approx^* ([\lambda[\Lambda.Src\ t] \bullet \Lambda.Src\ u] @ [\Lambda.subst\ u\ t]) =$   
 $[\Lambda.subst\ u\ t] \approx^* [\Lambda.subst\ u\ t]$   
**by** *blast*  
**moreover have** *Ide* ...  
**by** (*metis 1 2*  $\Lambda.Arr.simps(5)$   $\Lambda.Arr\text{-}Subst$  *Con-implies-Arr(2)* *Resid-Arr-self*  
*ind ide-char*)  
**ultimately show** *?thesis*  
**using** *ide-char* **by** *presburger*  
**qed**  
**qed**  
**finally show** *?thesis* **by** *blast*  
**qed**  
**qed**  
**qed**

**lemma** *Src-hd-standard-development:*  
**assumes**  $\Lambda.Arr\ t$  **and**  $\neg \Lambda.Ide\ t$   
**shows**  $\Lambda.Src\ (hd\ (standard\text{-}development\ t)) = \Lambda.Src\ t$   
**by** (*metis* *assms Src-hd-eqI cong-standard-development list.sel(1)*)

**lemma** *Trg-last-standard-development:*  
**assumes**  $\Lambda.Arr\ t$  **and**  $\neg \Lambda.Ide\ t$

**shows**  $\Lambda.Trg (last (standard-development t)) = \Lambda.Trg t$   
**by** (*metis* *assms* *Trg-last-eqI* *cong-standard-development* *last-ConsL*)

**lemma** *Srcs-standard-development*:

**shows**  $\llbracket \Lambda.Arr t; standard-development t \neq \square \rrbracket$   
 $\implies Srcs (standard-development t) = \{\Lambda.Src t\}$   
**by** (*metis* *Con-implies-Arr(1)* *Ide.simps(1)* *Ide-iff-standard-development-empty*  
*Src-hd-standard-development* *Srcs-simp $\Lambda P$*  *cong-standard-development* *ide-char*)

**lemma** *Trgs-standard-development*:

**shows**  $\llbracket \Lambda.Arr t; standard-development t \neq \square \rrbracket$   
 $\implies Trgs (standard-development t) = \{\Lambda.Trg t\}$   
**by** (*metis* *Con-implies-Arr(2)* *Ide.simps(1)* *Ide-iff-standard-development-empty*  
*Trg-last-standard-development* *Trgs-simp $\Lambda P$*  *cong-standard-development* *ide-char*)

**lemma** *development-standard-development*:

**shows**  $\Lambda.Arr t \longrightarrow development t (standard-development t)$   
**apply** (*rule* *standard-development.induct*)  
**apply** *blast*  
**apply** *simp*  
**apply** (*simp* *add*: *development-map-Lam*)

**proof**

**fix** *t1 t2*  
**assume** *ind1*:  $\Lambda.Arr t1 \wedge \Lambda.Arr t2$   
 $\implies \Lambda.Arr t1 \longrightarrow development t1 (standard-development t1)$   
**assume** *ind2*:  $\Lambda.Arr t1 \wedge \Lambda.Arr t2$   
 $\implies \Lambda.Arr t2 \longrightarrow development t2 (standard-development t2)$   
**assume** *t*:  $\Lambda.Arr (t1 \circ t2)$   
**show** *development (t1  $\circ$  t2) (standard-development (t1  $\circ$  t2))*  
**proof** (*cases* *standard-development t1 =  $\square$* )  
**show** *standard-development t1 =  $\square$*   
 $\implies development (t1 \circ t2) (standard-development (t1 \circ t2))$   
**using** *t ind2*  $\Lambda.Ide-Src$   $\Lambda.Ide-Trg$   $\Lambda.Ide-iff-Src-self$   $\Lambda.Ide-iff-Trg-self$   
*Ide-iff-standard-development-empty*  
*development-map-App-2* [*of*  $\Lambda.Src t1 t2 standard-development t2$ ]  
**by** *fastforce*  
**assume** *t1*: *standard-development t1  $\neq$   $\square$*   
**show** *development (t1  $\circ$  t2) (standard-development (t1  $\circ$  t2))*  
**proof** (*cases* *standard-development t2 =  $\square$* )  
**assume** *t2*: *standard-development t2 =  $\square$*   
**show** *?thesis*  
**using** *t t2 ind1* *Ide-iff-standard-development-empty* *development-map-App-1* **by** *simp*  
**next**  
**assume** *t2*: *standard-development t2  $\neq$   $\square$*   
**have** *development (t1  $\circ$  t2) (map ( $\lambda a. a \circ \Lambda.Src t2$ ) (standard-development t1))*  
**using**  $\Lambda.Arr.simps(4)$  *development-map-App-1* *ind1 t* **by** *presburger*  
**moreover** **have** *development ((t1  $\circ$  t2)<sup>1\\*</sup>*  
 $map (\lambda a. a \circ \Lambda.Src t2) (standard-development t1))$   
 $(map (\lambda a. \Lambda.Trg t1 \circ a) (standard-development t2))$

```

proof –
  have  $\Lambda.App\ t1\ t2\ ^1 \setminus^* \text{map } (\lambda a. a \circ \Lambda.Src\ t2) (\text{standard-development } t1) =$ 
     $\Lambda.Trq\ t1 \circ t2$ 
  proof –
    have  $\text{map } (\lambda a. a \circ \Lambda.Src\ t2) (\text{standard-development } t1) \sim^* [t1 \circ \Lambda.Src\ t2]$ 
    proof –
      have  $\text{map } (\lambda a. a \circ \Lambda.Src\ t2) (\text{standard-development } t1) =$ 
         $\text{standard-development } (t1 \circ \Lambda.Src\ t2)$ 
      by (metis  $\Lambda.Arr.simps(4)$   $\Lambda.Ide-Src$   $\Lambda.Ide-iff-Src-self$ 
         $Ide-iff-standard-development-empty$   $\Lambda.Ide-implies-Arr$   $Nil-is-map-conv$ 
         $append-Nil2$   $\text{standard-development.simps}(4)$   $t$ )
      also have  $\text{standard-development } (t1 \circ \Lambda.Src\ t2) \sim^* [t1 \circ \Lambda.Src\ t2]$ 
      by (metis  $\Lambda.Arr.simps(4)$   $\Lambda.Ide.simps(4)$   $\Lambda.Ide-Src$   $\Lambda.Ide-implies-Arr$ 
         $cong-standard-development$   $development-Ide$   $ind1$   $t1$ )
      finally show ?thesis by blast
    qed
  hence  $[t1 \circ t2] \setminus^* \text{map } (\lambda a. a \circ \Lambda.Src\ t2) (\text{standard-development } t1) =$ 
     $[t1 \circ t2] \setminus^* [t1 \circ \Lambda.Src\ t2]$ 
  by (metis  $Resid-parallel$   $con-imp-coinitial$   $prfx-implies-con$   $calculation$ 
     $development-implies-map-is-Nil-conv$   $t1$ )
  also have  $[t1 \circ t2] \setminus^* [t1 \circ \Lambda.Src\ t2] = [\Lambda.Trq\ t1 \circ t2]$ 
  using  $t$   $\Lambda.arr-resid-iff-con$   $\Lambda.resid-Arr-self$ 
  by simp force
  finally have  $[t1 \circ t2] \setminus^* \text{map } (\lambda a. a \circ \Lambda.Src\ t2) (\text{standard-development } t1) =$ 
     $[\Lambda.Trq\ t1 \circ t2]$ 
  by blast
  thus ?thesis
  by (simp add: Resid1x-as-Resid')
  qed
thus ?thesis
  by (metis  $ind2$   $\Lambda.Arr.simps(4)$   $\Lambda.Ide-Trq$   $\Lambda.Ide-iff-Src-self$   $development-map-App-2$ 
     $\Lambda.reduction-strategy-def$   $\Lambda.head-strategy-is-reduction-strategy$   $t$ )
  qed
ultimately show ?thesis
  using  $t$   $development-append$  [of  $t1 \circ t2$ 
     $\text{map } (\lambda a. a \circ \Lambda.Src\ t2) (\text{standard-development } t1)$ 
     $\text{map } (\lambda b. \Lambda.Trq\ t1 \circ b) (\text{standard-development } t2)$ ]
  by auto
  qed
qed
next
fix  $t1\ t2$ 
assume  $ind: \Lambda.Arr\ t1 \wedge \Lambda.Arr\ t2 \implies$ 
   $\Lambda.Arr\ (\Lambda.subst\ t2\ t1)$ 
   $\implies development\ (\Lambda.subst\ t2\ t1)\ (\text{standard-development}\ (\Lambda.subst\ t2\ t1))$ 
show  $\Lambda.Arr\ (\lambda[t1] \bullet t2) \implies development\ (\lambda[t1] \bullet t2)\ (\text{standard-development}\ (\lambda[t1] \bullet t2))$ 
proof
  assume  $1: \Lambda.Arr\ (\lambda[t1] \bullet t2)$ 
  have  $development\ (\Lambda.subst\ t2\ t1)\ (\text{standard-development}\ (\Lambda.subst\ t2\ t1))$ 

```

```

    using 1 ind by (simp add:  $\Lambda$ .Arr-Subst)
  thus development ( $\lambda[t1] \bullet t2$ ) (standard-development ( $\lambda[t1] \bullet t2$ ))
    using 1  $\Lambda$ .Ide-Src  $\Lambda$ .subs-Ide by auto
qed
qed

lemma Std-standard-development:
shows Std (standard-development t)
  apply (rule standard-development.induct)
    apply simp-all
  using Std-map-Lam
    apply blast
proof
fix t u
assume t:  $\Lambda$ .Arr t  $\wedge$   $\Lambda$ .Arr u  $\implies$  Std (standard-development t)
assume u:  $\Lambda$ .Arr t  $\wedge$   $\Lambda$ .Arr u  $\implies$  Std (standard-development u)
assume 0:  $\Lambda$ .Arr t  $\wedge$   $\Lambda$ .Arr u
show Std (map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t) @
  map ( $\lambda b. \Lambda$ .Trg t  $\circ$  b) (standard-development u))
proof (cases  $\Lambda$ .Ide t)
show  $\Lambda$ .Ide t  $\implies$  ?thesis
  using 0  $\Lambda$ .Ide-iff-Trg-self Ide-iff-standard-development-empty u Std-map-App2
  by fastforce
assume 1:  $\neg \Lambda$ .Ide t
show ?thesis
proof (cases  $\Lambda$ .Ide u)
show  $\Lambda$ .Ide u  $\implies$  ?thesis
  using t u 0 1 Std-map-App1 [of  $\Lambda$ .Src u standard-development t]  $\Lambda$ .Ide-Src
  by (metis Ide-iff-standard-development-empty append-Nil2 list.simps(8))
assume 2:  $\neg \Lambda$ .Ide u
show ?thesis
proof (intro Std-append)
show 3: Std (map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t))
  using t 0 Std-map-App1  $\Lambda$ .Ide-Src by blast
show Std (map ( $\lambda b. \Lambda$ .Trg t  $\circ$  b) (standard-development u))
  using u 0 Std-map-App2  $\Lambda$ .Ide-Trg by simp
show map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t) = []  $\vee$ 
  map ( $\lambda b. \Lambda$ .Trg t  $\circ$  b) (standard-development u) = []  $\vee$ 
   $\Lambda$ .sseq (last (map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t)))
    (hd (map ( $\lambda b. \Lambda$ .Trg t  $\circ$  b) (standard-development u)))
proof -
have  $\Lambda$ .sseq (last (map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t)))
  (hd (map ( $\lambda b. \Lambda$ .Trg t  $\circ$  b) (standard-development u)))
proof -
obtain x where x: last (map ( $\lambda a. a \circ \Lambda$ .Src u) (standard-development t)) =
  x  $\circ$   $\Lambda$ .Src u
  using 0 1 Ide-iff-standard-development-empty last-map by auto
obtain y where y: hd (map ( $\lambda b. \Lambda$ .Trg t  $\circ$  b) (standard-development u)) =
   $\Lambda$ .Trg t  $\circ$  y

```

```

    using 0 2 Ide-iff-standard-development-empty list.map-sel(1) by auto
  have  $\Lambda$ .elementary-reduction x
  proof -
    have  $\Lambda$ .elementary-reduction (x  $\circ$   $\Lambda$ .Src u)
      using x
    by (metis 0 1 3 Ide-iff-standard-development-empty Nil-is-map-conv Std.simps(2)
        Std-imp-sseq-last-hd append-butlast-last-id append-self-conv2 list.discI
        list.sel(1)  $\Lambda$ .sseq-imp-elementary-reduction2)
    thus ?thesis
      using 0  $\Lambda$ .Ide-Src  $\Lambda$ .elementary-reduction-not-ide by auto
  qed
  moreover have  $\Lambda$ .elementary-reduction y
  proof -
    have  $\Lambda$ .elementary-reduction ( $\Lambda$ .Trg t  $\circ$  y)
      using y
    by (metis 0 2  $\Lambda$ .Ide-Trg Ide-iff-standard-development-empty
        u Std.elims(2)  $\Lambda$ .elementary-reduction.simps(4) list.map-sel(1) list.sel(1)
         $\Lambda$ .sseq-imp-elementary-reduction1)
    thus ?thesis
      using 0  $\Lambda$ .Ide-Trg  $\Lambda$ .elementary-reduction-not-ide by auto
  qed
  moreover have  $\Lambda$ .Trg t =  $\Lambda$ .Trg x
    by (metis 0 1 Ide-iff-standard-development-empty Trg-last-standard-development
        x  $\Lambda$ .lambda.inject(3) last-map)
  moreover have  $\Lambda$ .Src u =  $\Lambda$ .Src y
    using y
  by (metis 0 2  $\Lambda$ .Arr-not-Nil  $\Lambda$ .Cointial-iff-Con
      Ide-iff-standard-development-empty development.elims(2) development-imp-Arr
      development-standard-development  $\Lambda$ .lambda.inject(3) list.map-sel(1)
      list.sel(1))
  ultimately show ?thesis
    using x y by simp
  qed
  thus ?thesis by blast
  qed
  qed
  qed
  next
  fix t u
  assume ind:  $\Lambda$ .Arr t  $\wedge$   $\Lambda$ .Arr u  $\implies$  Std (standard-development ( $\Lambda$ .subst u t))
  show  $\Lambda$ .Arr t  $\wedge$   $\Lambda$ .Arr u
     $\implies$  Std (( $\lambda$ [ $\Lambda$ .Src t]  $\bullet$   $\Lambda$ .Src u) # standard-development ( $\Lambda$ .subst u t))
  proof
    assume 1:  $\Lambda$ .Arr t  $\wedge$   $\Lambda$ .Arr u
    show Std (( $\lambda$ [ $\Lambda$ .Src t]  $\bullet$   $\Lambda$ .Src u) # standard-development ( $\Lambda$ .subst u t))
  proof (cases  $\Lambda$ .Ide ( $\Lambda$ .subst u t))
    show  $\Lambda$ .Ide ( $\Lambda$ .subst u t)
       $\implies$  Std (( $\lambda$ [ $\Lambda$ .Src t]  $\bullet$   $\Lambda$ .Src u) # standard-development ( $\Lambda$ .subst u t))
  qed
  qed

```

```

    using 1  $\Lambda$ .Arr-Subst  $\Lambda$ .Ide-Src Ide-iff-standard-development-empty by simp
  assume 2:  $\neg \Lambda$ .Ide ( $\Lambda$ .subst u t)
  show Std (( $\lambda[\Lambda$ .Src t]  $\bullet$   $\Lambda$ .Src u) # standard-development ( $\Lambda$ .subst u t))
  proof -
    have  $\Lambda$ .sseq ( $\lambda[\Lambda$ .Src t]  $\bullet$   $\Lambda$ .Src u) (hd (standard-development ( $\Lambda$ .subst u t)))
    proof -
      have  $\Lambda$ .elementary-reduction (hd (standard-development ( $\Lambda$ .subst u t)))
      using ind
      by (metis 1 2  $\Lambda$ .Arr-Subst Ide-iff-standard-development-empty
          Std.elims(2) list.sel(1)  $\Lambda$ .sseq-imp-elementary-reduction1)
    moreover have  $\Lambda$ .seq ( $\lambda[\Lambda$ .Src t]  $\bullet$   $\Lambda$ .Src u)
      (hd (standard-development ( $\Lambda$ .subst u t)))
    using 1 2 Src-hd-standard-development calculation  $\Lambda$ .Arr.simps(5)
       $\Lambda$ .Arr-Src  $\Lambda$ .Arr-Subst  $\Lambda$ .Src-Subst  $\Lambda$ .Trg.simps(4)  $\Lambda$ .Trg-Src  $\Lambda$ .arr-char
       $\Lambda$ .elementary-reduction-is-arr  $\Lambda$ .seq-char
    by presburger
    ultimately show ?thesis
    using 1  $\Lambda$ .Ide-Src  $\Lambda$ .sseq-Beta by auto
  qed
  moreover have Std (standard-development ( $\Lambda$ .subst u t))
  using 1 ind by blast
  ultimately show ?thesis
  by (metis 1 2  $\Lambda$ .Arr-Subst Ide-iff-standard-development-empty Std.simps(3)
      list.collapse)
  qed
  qed
  qed
  qed

```

### 3.6.3 Standardization

In this section, we define and prove correct a function that takes an arbitrary reduction path and produces a standard reduction path congruent to it. The method is roughly analogous to insertion sort: given a path, recursively standardize the tail and then “insert” the head into to the result. A complication is that in general the head may be a parallel reduction instead of an elementary reduction, and in any case elementary reductions are not preserved under residuation so we need to be able to handle the parallel reductions that arise from permuting elementary reductions. In general, this means that parallel reduction steps have to be decomposed into factors, and then each factor has to be inserted at its proper position. Another issue is that reductions don’t all happen at the top level of a term, so we need to be able to descend recursively into terms during the insertion procedure. The key idea here is: in a standard reduction, once a step has occurred that is not a head reduction, then all subsequent terms will have *App* as their top-level constructor. So, once we have passed a step that is not a head reduction, we can recursively descend into the subsequent applications and treat the “rator” and the “rand” parts independently.

The following function performs the core insertion part of the standardization al-

gorithm. It assumes that it is given an arbitrary parallel reduction  $t$  and an already-standard reduction path  $U$ , and it inserts  $t$  into  $U$ , producing a standard reduction path that is congruent to  $t \# U$ . A somewhat elaborate case analysis is required to determine whether  $t$  needs to be factored and whether part of it might need to be permuted with the head of  $U$ . The recursion is complicated by the need to make sure that the second argument  $U$  is always a standard reduction path. This is so that it is possible to decide when the rest of the steps will be applications and it is therefore possible to recurse into them. This constrains what recursive calls we can make, since we are not able to make a recursive call in which an identity has been prepended to  $U$ . Also, if  $t \# U$  consists completely of identities, then its standardization is the empty list  $[]$ , which is not a path and cannot be congruent to  $t \# U$ . So in order to be able to apply the induction hypotheses in the correctness proof, we need to make sure that we don't make recursive calls when  $U$  itself would consist entirely of identities. Finally, when we descend through an application, the step  $t$  and the path  $U$  are projected to their "rator" and "rand" components, which are treated separately and the results concatenated. However, the projection operations can introduce identities and therefore do not preserve elementary reductions. To handle this, we need to filter out identities after projection but before the recursive call.

Ensuring termination also involves some care: we make recursive calls in which the length of the second argument is increased, but the "height" of the first argument is decreased. So we use a lexicographic order that makes the height of the first argument more significant and the length of the second argument secondary. The base cases either discard paths that consist entirely of identities, or else they expand a single parallel reduction  $t$  into a standard development.

```

function (sequential) stdz-insert
where stdz-insert t [] = standard-development t
  | stdz-insert «-» U = stdz-insert (hd U) (tl U)
  | stdz-insert λ[t] U =
    (if λ.Ide t then
      stdz-insert (hd U) (tl U)
    else
      map λ.Lam (stdz-insert t (map λ.un-Lam U)))
  | stdz-insert (λ[t] ◦ u) ((λ[-] • -) # U) = stdz-insert (λ[t] • u) U
  | stdz-insert (t ◦ u) U =
    (if λ.Ide (t ◦ u) then
      stdz-insert (hd U) (tl U)
    else if λ.seq (t ◦ u) (hd U) then
      if λ.contains-head-reduction (t ◦ u) then
        if λ.Ide ((t ◦ u) \ λ.head-redex (t ◦ u)) then
          λ.head-redex (t ◦ u) # stdz-insert (hd U) (tl U)
        else
          λ.head-redex (t ◦ u) # stdz-insert ((t ◦ u) \ λ.head-redex (t ◦ u)) U
      else if λ.contains-head-reduction (hd U) then
        if λ.Ide ((t ◦ u) \ λ.head-strategy (t ◦ u)) then
          stdz-insert (λ.head-strategy (t ◦ u)) (tl U)
        else

```

```

       $\Lambda.head\text{-strategy } (t \circ u) \# \text{stdz-insert } ((t \circ u) \setminus \Lambda.head\text{-strategy } (t \circ u)) (tl U)$ 
    else
      map ( $\lambda a. a \circ \Lambda.Src u$ )
        (stdz-insert t (filter notIde (map  $\Lambda.un\text{-App1 } U$ ))) @
      map ( $\lambda b. \Lambda.Trq (\Lambda.un\text{-App1 } (last U)) \circ b$ )
        (stdz-insert u (filter notIde (map  $\Lambda.un\text{-App2 } U$ )))
    else []
  | stdz-insert ( $\lambda [t] \bullet u$ ) U =
    (if  $\Lambda.Arr t \wedge \Lambda.Arr u$  then
      ( $\lambda [\Lambda.Src t] \bullet \Lambda.Src u$ ) # stdz-insert ( $\Lambda.subst u t$ ) U
    else [])
  | stdz-insert - - = []
by pat-completeness auto

```

```

fun standardize
where standardize [] = []
  | standardize U = stdz-insert (hd U) (standardize (tl U))

```

```

abbreviation stdzins-rel
where stdzins-rel  $\equiv$  mlex-prod (length o snd) (inv-image (mlex-prod  $\Lambda.hgt \Lambda.subterm\text{-rel}$ )
fst)

```

```

termination stdz-insert
using  $\Lambda.subterm.intrros(2-3)$   $\Lambda.hgt\text{-Subst less-Suc-eq-le}$   $\Lambda.elementary\text{-reduction-decreases-hgt}$ 
 $\Lambda.elementary\text{-reduction-head-redex}$   $\Lambda.contains\text{-head-reduction-iff}$ 
 $\Lambda.elementary\text{-reduction-is-arr}$   $\Lambda.Src\text{-head-redex}$   $\Lambda.App\text{-Var-contains-no-head-reduction}$ 
 $\Lambda.hgt\text{-resid-App-head-redex}$   $\Lambda.seq\text{-char}$ 
apply (relation stdzins-rel)
apply (auto simp add: wf-mlex  $\Lambda.wf\text{-subterm-rel}$  mlex-iff mlex-less  $\Lambda.subterm\text{-lemmas}(1)$ )
by (meson dual-order.eq-iff length-filter-le not-less-eq-eq)+

```

```

lemma stdz-insert-Ide:
shows  $Ide (t \# U) \implies \text{stdz-insert } t U = []$ 
proof (induct U arbitrary: t)
  show  $\bigwedge t. Ide [t] \implies \text{stdz-insert } t [] = []$ 
    by (metis Ide-iff-standard-development-empty  $\Lambda.Ide\text{-implies-Arr}$  Ide.simps(2)
 $\Lambda.ide\text{-char stdz-insert.simps}(1)$ )
  show  $\bigwedge U. [\bigwedge t. Ide (t \# U) \implies \text{stdz-insert } t U = []; Ide (t \# u \# U)]$ 
     $\implies \text{stdz-insert } t (u \# U) = []$ 
    for t u
    using  $\Lambda.ide\text{-char}$ 
    apply (cases t; cases u)
    apply simp-all
    by fastforce
qed

```

```

lemma stdz-insert-Ide-Std:

```

```

shows  $\llbracket \Lambda.Ide\ u; seq\ [u]\ U; Std\ U \rrbracket \implies stdz-insert\ u\ U = stdz-insert\ (hd\ U)\ (tl\ U)$ 
proof (induct U arbitrary: u)
  show  $\bigwedge u. \llbracket \Lambda.Ide\ u; seq\ [u]\ []; Std\ [] \rrbracket \implies stdz-insert\ u\ [] = stdz-insert\ (hd\ [])\ (tl\ [])$ 
    by (simp add: seq-char)
  fix u v U
  assume u:  $\Lambda.Ide\ u$ 
  assume seq:  $seq\ [u]\ (v\ \#)\ U$ 
  assume Std:  $Std\ (v\ \#)\ U$ 
  assume ind:  $\bigwedge u. \llbracket \Lambda.Ide\ u; seq\ [u]\ U; Std\ U \rrbracket \implies stdz-insert\ u\ U = stdz-insert\ (hd\ U)\ (tl\ U)$ 
show  $stdz-insert\ u\ (v\ \#)\ U = stdz-insert\ (hd\ (v\ \#)\ U)\ (tl\ (v\ \#)\ U)$ 
  using u ind stdz-insert-Ide Ide-implies-Arr
  apply (cases u; cases v)
    apply simp-all
proof –
  fix x y a b
  assume xy:  $\Lambda.Ide\ x \wedge \Lambda.Ide\ y$ 
  assume u':  $u = x \circ y$ 
  assume v':  $v = \lambda[a] \bullet b$ 
  have ab:  $\Lambda.Ide\ a \wedge \Lambda.Ide\ b$ 
    using Std  $\langle v = \lambda[a] \bullet b \rangle Std.elims(2)\ \Lambda.sseq-Beta$ 
    by (metis Std-consE  $\Lambda.elementary-reduction.simps(5)\ Std.simps(2)$ )
  have  $x = \lambda[a] \wedge y = b$ 
    using xy ab u u' v' seq seq-char
    by (metis  $\Lambda.Ide-iff-Src-self\ \Lambda.Ide-iff-Trg-self\ \Lambda.Ide-implies-Arr\ \Lambda.Src.simps(5)\$ 
       $Srcs-simp_{\Lambda P}\ Trgs.simps(2)\ \Lambda.lambda.inject(3)\ list.sel(1)\ singleton-insert-inj-eq\$ 
       $\Lambda.targets-char_{\Lambda}$ )
  thus  $stdz-insert\ (x \circ y)\ ((\lambda[a] \bullet b)\ \#)\ U = stdz-insert\ (\lambda[a] \bullet b)\ U$ 
    using u u' stdz-insert.simps(4) by presburger
  qed
qed

```

Insertion of a term with *Beta* as its top-level constructor always leaves such a term at the head of the result. Stated another way, *Beta* at the top-level must always come first in a standard reduction path.

```

lemma stdz-insert-Beta-ind:
shows  $\llbracket \Lambda.hgt\ t + length\ U \leq n; \Lambda.is-Beta\ t; seq\ [t]\ U \rrbracket \implies \Lambda.is-Beta\ (hd\ (stdz-insert\ t\ U))$ 
proof (induct n arbitrary: t U)
  show  $\bigwedge t\ U. \llbracket \Lambda.hgt\ t + length\ U \leq 0; \Lambda.is-Beta\ t; seq\ [t]\ U \rrbracket \implies \Lambda.is-Beta\ (hd\ (stdz-insert\ t\ U))$ 
    using Arr.simps(1) seq-char by blast
  fix n t U
  assume ind:  $\bigwedge t\ U. \llbracket \Lambda.hgt\ t + length\ U \leq n; \Lambda.is-Beta\ t; seq\ [t]\ U \rrbracket \implies \Lambda.is-Beta\ (hd\ (stdz-insert\ t\ U))$ 
  assume seq:  $seq\ [t]\ U$ 
  assume n:  $\Lambda.hgt\ t + length\ U \leq Suc\ n$ 
  assume t:  $\Lambda.is-Beta\ t$ 
  show  $\Lambda.is-Beta\ (hd\ (stdz-insert\ t\ U))$ 

```

```

using t seq seq-char
by (cases U; cases t; cases hd U) auto
qed

```

```

lemma stdz-insert-Beta:
assumes  $\Lambda.is\text{-}Beta\ t$  and seq [t] U
shows  $\Lambda.is\text{-}Beta\ (hd\ (stdz\text{-}insert\ t\ U))$ 
using assms stdz-insert-Beta-ind by blast

```

This is the correctness lemma for insertion: Given a term  $t$  and standard reduction path  $U$  sequential with it, the result of insertion is a standard reduction path which is congruent to  $t \# U$  unless  $t \# U$  consists entirely of identities.

The proof is very long. Its structure parallels that of the definition of the function *stdz-insert*. For really understanding the details, I strongly suggest viewing the proof in Isabelle/JEdit and using the code folding feature to unfold the proof a little bit at a time.

```

lemma stdz-insert-correctness:
shows seq [t] U  $\wedge$  Std U  $\longrightarrow$ 
      Std (stdz-insert t U)  $\wedge$  ( $\neg$  Ide (t # U)  $\longrightarrow$  cong (stdz-insert t U) (t # U))
      (is ?P t U)
proof (rule stdz-insert.induct [of ?P])
  show  $\bigwedge t. ?P\ t\ []$ 
    using seq-char by simp
  show  $\bigwedge u\ U. ?P\ \#(u\ \# U)$ 
    using seq-char not-arr-null null-char by auto
  show  $\bigwedge x\ u\ U. ?P\ (hd\ (u\ \# U))\ (tl\ (u\ \# U)) \Longrightarrow ?P\ \langle\langle x \rangle\rangle\ (u\ \# U)$ 
  proof –
    fix x u U
    assume ind: ?P (hd (u # U)) (tl (u # U))
    show ?P  $\langle\langle x \rangle\rangle\ (u\ \# U)$ 
    proof (intro impI, elim conjE, intro conjI)
      assume seq: seq [⟨x⟩] (u # U)
      assume Std: Std (u # U)
      have 1: stdz-insert ⟨x⟩ (u # U) = stdz-insert u U
        by simp
      have 2: U  $\neq [] \Longrightarrow seq\ [u]\ U$ 
        using Std Std-imp-Arr
        by (simp add: arrI_P arr-append-imp-seq)
      show Std (stdz-insert ⟨x⟩ (u # U))
        using ind
        by (metis 1 2 Std Std-standard-development list.exhaust-sel list.sel(1) list.sel(3)
          reduction-paths.Std.simps(3) reduction-paths.stdz-insert.simps(1))
      show  $\neg\ Ide\ (\langle\langle x \rangle\rangle\ \# u\ \# U) \longrightarrow stdz\text{-}insert\ \langle\langle x \rangle\rangle\ (u\ \# U) \sim^* \langle\langle x \rangle\rangle\ \# u\ \# U$ 
      proof (cases U = [])
        show U = []  $\Longrightarrow ?thesis$ 
          using cong-standard-development cong-cons-ideI(1)
          apply simp
          by (metis Arr.simps(1–2) Arr-iff-Con-self Con-rec(3)  $\Lambda.in\text{-}sourcesI$  con-char
            cong-transitive ideE  $\Lambda.Ide.simps(2)$   $\Lambda.arr\text{-}char$   $\Lambda.ide\text{-}char$  seq)

```

```

assume  $U: U \neq []$ 
show ?thesis
  using 1 2 ind seq seq-char cong-cons-ideI(1)
  apply simp
  by (metis Std Std-consE U  $\Lambda$ .Arr.simps(2)  $\Lambda$ .Ide.simps(2)  $\Lambda$ .targets-simps(2)
      prfx-transitive)
qed
qed
qed
show  $\bigwedge M u U. [\Lambda.Ide M \implies ?P (hd (u \# U)) (tl (u \# U));$ 
       $\neg \Lambda.Ide M \implies ?P M (map \Lambda.un-Lam (u \# U))]$ 
       $\implies ?P \lambda[M] (u \# U)$ 
proof –
  fix  $M u U$ 
  assume ind1:  $\Lambda.Ide M \implies ?P (hd (u \# U)) (tl (u \# U))$ 
  assume ind2:  $\neg \Lambda.Ide M \implies ?P M (map \Lambda.un-Lam (u \# U))$ 
  show  $?P \lambda[M] (u \# U)$ 
  proof (intro impI, elim conjE)
    assume seq:  $seq [\lambda[M]] (u \# U)$ 
    assume Std:  $Std (u \# U)$ 
    have  $u: \Lambda.is-Lam u$ 
    using seq
    by (metis insert-subset  $\Lambda$ .lambda.disc(8) list.simps(15) mem-Collect-eq
        seq-Lam-Arr-implies)
    have  $U: set U \subseteq Collect \Lambda.is-Lam$ 
    using  $u seq$ 
    by (metis insert-subset  $\Lambda$ .lambda.disc(8) list.simps(15) seq-Lam-Arr-implies)
    show  $Std (stdz-insert \lambda[M] (u \# U)) \wedge$ 
       $(\neg Ide (\lambda[M] \# u \# U) \longrightarrow stdz-insert \lambda[M] (u \# U) \text{ *~* } \lambda[M] \# u \# U)$ 
    proof (cases  $\Lambda.Ide M$ )
      assume  $M: \Lambda.Ide M$ 
      have 1:  $stdz-insert \lambda[M] (u \# U) = stdz-insert u U$ 
      using  $M$  by simp
      show ?thesis
      proof (cases Ide (u \# U))
        show  $Ide (u \# U) \implies ?thesis$ 
        using 1 Std-standard-development Ide-iff-standard-development-empty
        by (metis Ide-imp-Ide-hd Std Std-implies-set-subset-elementary-reduction
             $\Lambda$ .elementary-reduction-not-ide list.sel(1) list.set-intros(1)
            mem-Collect-eq subset-code(1))
        assume 2:  $\neg Ide (u \# U)$ 
        show ?thesis
        proof (cases U = [])
          assume 3:  $U = []$ 
          have 4: standard-development u *~*  $[\lambda[M]] @ [u]$ 
          using  $M$  2 3 seq ide-char cong-standard-development [of u]
            cong-append-ideI(1) [of  $[\lambda[M]] [u]$ 
          by (metis Arr-imp-arr-hd Ide.simps(2) Std Std-imp-Arr cong-transitive
               $\Lambda$ .Ide.simps(3)  $\Lambda$ .arr-char  $\Lambda$ .ide-char list.sel(1) not-Cons-self2)

```

```

show ?thesis
  using 1 3 4 Std-standard-development by force
next
assume 3:  $U \neq []$ 
have stdz-insert  $\lambda[M]$  ( $u \# U$ ) = stdz-insert  $u$   $U$ 
  using M 3 by simp
have 5:  $\Lambda.Arr\ u \wedge \neg \Lambda.Ide\ u$ 
  by (meson 3 Std Std-consE  $\Lambda.elementary-reduction-not-ide$   $\Lambda.ide-char$ 
       $\Lambda.sseq-imp-elementary-reduction1$ )
have 4: standard-development  $u$  @  $U$   $\sim^*$  ( $\lambda[M]$  @  $[u]$ ) @  $U$ 
proof (intro cong-append seqI $_{\Lambda P}$ )
  show Arr (standard-development  $u$ )
using 5 Std-standard-development Std-imp-Arr Ide-iff-standard-development-empty
  by force
show Arr  $U$ 
  using Std 3 by auto
show  $\Lambda.Trg$  (last (standard-development  $u$ )) =  $\Lambda.Src$  (hd  $U$ )
  by (metis 3 5 Std Std-consE Trg-last-standard-development  $\Lambda.seq-char$ 
       $\Lambda.sseq-imp-seq$ )
show standard-development  $u$   $\sim^*$   $\lambda[M]$  @  $[u]$ 
  using M 5 Std Std-imp-Arr cong-standard-development [of  $u$ ]
      cong-append-ideI(3) [of  $\lambda[M]$   $[u]$ ]
  by (metis (no-types, lifting) Arr.simps(2) Ide.simps(2) arr-char ide-char
       $\Lambda.Ide.simps(3)$   $\Lambda.arr-char$   $\Lambda.ide-char$  prfx-transitive seq seq-def
      sources-cons)
show  $U \sim^* U$ 
  by (simp add:  $\langle Arr\ U \rangle$  arr-char prfx-reflexive)
qed
show ?thesis
proof (intro conjI)
  show Std (stdz-insert  $\lambda[M]$  ( $u \# U$ ))
    by (metis (no-types, lifting) 1 3 M Std Std-consE append-Cons
        append-eq-append-conv2 append-self-conv arr-append-imp-seq ind1
        list.sel(1) list.sel(3) not-Cons-self2 seq seq-def)
  show  $\neg Ide$  ( $\lambda[M]$  #  $u$  #  $U$ )  $\longrightarrow$  stdz-insert  $\lambda[M]$  ( $u \# U$ )  $\sim^*$   $\lambda[M]$  #  $u$  #  $U$ 
proof
  have seq  $[u]$   $U \wedge Std\ U$ 
    using 2 3 Std
    by (metis Cons-eq-appendI Std-consE arr-append-imp-seq neq-Nil-conv
        self-append-conv2 seq seqE)
  thus stdz-insert  $\lambda[M]$  ( $u \# U$ )  $\sim^*$   $\lambda[M]$  #  $u$  #  $U$ 
    using M 1 2 3 4 ind1 cong-cons-ideI(1) [of  $\lambda[M]$   $u$  #  $U$ ]
    apply simp
    by (meson cong-transitive seq)
qed
qed
qed
qed
next

```

```

assume  $M: \neg \Lambda.Ide\ M$ 
have  $1: stdz-insert\ \lambda[M]\ (u\ \# \ U) =$ 
       $map\ \Lambda.Lam\ (stdz-insert\ M\ (\Lambda.un-Lam\ u\ \# \ map\ \Lambda.un-Lam\ U))$ 
using  $M$  by simp
show ?thesis
proof (intro conjI)
  show  $Std\ (stdz-insert\ \lambda[M]\ (u\ \# \ U))$ 
    by (metis 1 M Std Std-map-Lam Std-map-un-Lam ind2 \Lambda.lambda.disc(8))
      list.simps(9) seq seq-Lam-Arr-implies seq-map-un-Lam
  show  $\neg\ Ide\ (\lambda[M]\ \# \ u\ \# \ U) \longrightarrow stdz-insert\ \lambda[M]\ (u\ \# \ U) \sim^* \lambda[M]\ \# \ u\ \# \ U$ 
proof
  have  $map\ \Lambda.Lam\ (stdz-insert\ M\ (\Lambda.un-Lam\ u\ \# \ map\ \Lambda.un-Lam\ U)) \sim^*$ 
     $\lambda[M]\ \# \ u\ \# \ U$ 
proof  $-$ 
  have  $map\ \Lambda.Lam\ (stdz-insert\ M\ (\Lambda.un-Lam\ u\ \# \ map\ \Lambda.un-Lam\ U)) \sim^*$ 
     $map\ \Lambda.Lam\ (M\ \# \ \Lambda.un-Lam\ u\ \# \ map\ \Lambda.un-Lam\ U)$ 
by (metis (mono-tags, opaque-lifting) Ide-imp-Ide-hd M Std Std-map-un-Lam)
      cong-map-Lam ind2 \Lambda.ide-char \Lambda.lambda.discI(2)
      list.sel(1) list.simps(9) seq seq-Lam-Arr-implies seq-map-un-Lam
thus ?thesis
using  $u\ U$ 
by (simp add: map-idI subset-code(1))
qed
thus  $stdz-insert\ \lambda[M]\ (u\ \# \ U) \sim^* \lambda[M]\ \# \ u\ \# \ U$ 
using  $1$  by presburger
qed
qed
qed
qed
show  $\bigwedge M\ N\ A\ B\ U. ?P\ (\lambda[M]\ \bullet\ N)\ U \implies ?P\ (\lambda[M]\ \circ\ N)\ ((\lambda[A]\ \bullet\ B)\ \# \ U)$ 
proof  $-$ 
fix  $M\ N\ A\ B\ U$ 
assume ind: ?P (\lambda[M] \bullet N) U
show  $?P\ (\lambda[M]\ \circ\ N)\ ((\lambda[A]\ \bullet\ B)\ \# \ U)$ 
proof (intro impI, elim conjE)
  assume seq: seq [\lambda[M] \circ N] ((\lambda[A] \bullet B) \# U)
  assume Std: Std ((\lambda[A] \bullet B) \# U)
  have  $MN: \Lambda.Arr\ M \wedge \Lambda.Arr\ N$ 
using seq
by (simp add: seq-char)
have  $AB: \Lambda.Trq\ M = A \wedge \Lambda.Trq\ N = B$ 
proof  $-$ 
have  $1: \Lambda.Ide\ A \wedge \Lambda.Ide\ B$ 
using Std
by (metis Std.simps(2) Std.simps(3) \Lambda.elementary-reduction.simps(5))
      list.exhaust-sel \Lambda.sseq-Beta
moreover have  $Trgs\ [\lambda[M]\ \circ\ N] = Srcs\ [\lambda[A]\ \bullet\ B]$ 
using  $1$  by seq seq-char

```

by (*simp add:  $\Lambda$ .Ide-implies-Arr Srcs-simp $_{\Lambda P}$* )  
 ultimately show *?thesis*  
 by (*metis  $\Lambda$ .Ide-iff-Src-self  $\Lambda$ .Ide-implies-Arr  $\Lambda$ .Src.simps(5) Srcs-simp $_{\Lambda P}$   
 $\Lambda$ .Trg.simps(2-3) Trgs-simp $_{\Lambda P}$   $\Lambda$ .lambda.inject(2)  $\Lambda$ .lambda.sel(3-4)  
 last.simps list.sel(1) seq-char seq the-elem-eq*)

**qed**  
**have 1:** *stdz-insert* ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) #  $U$ ) = *stdz-insert* ( $\lambda[M] \bullet N$ )  $U$   
 by *auto*  
**show** *Std* (*stdz-insert* ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) #  $U$ ))  $\wedge$   
 ( $\neg$  *Ide* (( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) #  $U$ )  $\longrightarrow$   
*stdz-insert* ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) #  $U$ )  $\sim^*$  ( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) #  $U$ )

**proof** (*cases*  $U = []$ )  
 assume  $U: U = []$   
**have 1:** *stdz-insert* ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) #  $U$ ) =  
*standard-development* ( $\lambda[M] \bullet N$ )  
 using  $U$  by *simp*  
**show** *?thesis*  
**proof** (*intro conjI*)  
**show** *Std* (*stdz-insert* ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) #  $U$ ))  
 using 1 *Std-standard-development* by *presburger*  
**show**  $\neg$  *Ide* (( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) #  $U$ )  $\longrightarrow$   
*stdz-insert* ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) #  $U$ )  $\sim^*$  ( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) #  $U$

**proof** (*intro impI*)  
 assume 2:  $\neg$  *Ide* (( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) #  $U$ )  
**have** *stdz-insert* ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) #  $U$ ) =  
 ( $\lambda[\Lambda$ .Src  $M$ ]  $\bullet$   $\Lambda$ .Src  $N$ ) # *standard-development* ( $\Lambda$ .subst  $N$   $M$ )  
 using 1 *MN* by *simp*  
**also have** ...  $\sim^*$  [ $\lambda[M] \bullet N$ ]  
 using *MN AB cong-standard-development*  
 by (*metis* 1 *calculation*  $\Lambda$ .Arr.simps(5)  $\Lambda$ .Ide.simps(5))  
**also have** [ $\lambda[M] \bullet N$ ]  $\sim^*$  ( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) #  $U$   
 using *AB MN U Beta-decomp(2)* [*of M N*] by *simp*  
**finally show** *stdz-insert* ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) #  $U$ )  $\sim^*$   
 ( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) #  $U$   
 by *blast*

**qed**  
**qed**  
**next**  
 assume  $U: U \neq []$   
**have 1:** *stdz-insert* ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) #  $U$ ) = *stdz-insert* ( $\lambda[M] \bullet N$ )  $U$   
 using  $U$  by *simp*  
**have 2:** *seq* [ $\lambda[M] \bullet N$ ]  $U$   
 using *MN AB U Std  $\Lambda$ .sseq-imp-seq*  
**apply** (*intro seqI $_{\Lambda P}$* )  
**apply** *auto*  
 by *fastforce*  
**have 3:** *Std*  $U$   
 using *Std* by *fastforce*  
**show** *?thesis*

```

proof (intro conjI)
  show Std (stdz-insert ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) # U))
    using 2 3 ind by simp
  show  $\neg$  Ide (( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) # U)  $\longrightarrow$ 
    stdz-insert ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) # U)  $\sim^*$  ( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) # U
  proof
    have stdz-insert ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) # U)  $\sim^*$  [ $\lambda[M] \bullet N$ ] @ U
      by (metis 1 2 3  $\Lambda$ .Ide.simps(5) U Ide.simps(3) append.left-neutral
        append-Cons  $\Lambda$ .ide-char ind list.exhaust)
    also have [ $\lambda[M] \bullet N$ ] @ U  $\sim^*$  (( $\lambda[M] \circ N$ ) @ [ $\lambda[A] \bullet B$ ]) @ U
      using MN AB Beta-decomp
      by (meson 2 cong-append cong-reflexive seqE)
    also have (( $\lambda[M] \circ N$ ) @ [ $\lambda[A] \bullet B$ ]) @ U = ( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) # U
      by simp
    finally show stdz-insert ( $\lambda[M] \circ N$ ) (( $\lambda[A] \bullet B$ ) # U)  $\sim^*$ 
      ( $\lambda[M] \circ N$ ) # ( $\lambda[A] \bullet B$ ) # U
      by argo
  qed
qed
qed
qed
qed
show  $\bigwedge M N u U. (\Lambda$ .Arr M  $\wedge$   $\Lambda$ .Arr N  $\implies$  ?P ( $\Lambda$ .subst N M) (u # U))
   $\implies$  ?P ( $\lambda[M] \bullet N$ ) (u # U)
proof –
  fix M N u U
  assume ind:  $\Lambda$ .Arr M  $\wedge$   $\Lambda$ .Arr N  $\implies$  ?P ( $\Lambda$ .subst N M) (u # U)
  show ?P ( $\lambda[M] \bullet N$ ) (u # U)
  proof (intro impI, elim conjE)
    assume seq: seq [ $\lambda[M] \bullet N$ ] (u # U)
    assume Std: Std (u # U)
    have MN:  $\Lambda$ .Arr M  $\wedge$   $\Lambda$ .Arr N
      using seq seq-char by simp
    show Std (stdz-insert ( $\lambda[M] \bullet N$ ) (u # U))  $\wedge$ 
      ( $\neg$  Ide ( $\Lambda$ .Beta M N # u # U))  $\longrightarrow$ 
      cong (stdz-insert ( $\lambda[M] \bullet N$ ) (u # U)) (( $\lambda[M] \bullet N$ ) # u # U)
  proof (cases  $\Lambda$ .Ide ( $\Lambda$ .subst N M))
    assume 1:  $\Lambda$ .Ide ( $\Lambda$ .subst N M)
    have 2:  $\neg$  Ide (u # U)
      using Std Std-implies-set-subset-elementary-reduction  $\Lambda$ .elementary-reduction-not-ide
      by force
    have 3: stdz-insert ( $\lambda[M] \bullet N$ ) (u # U) = ( $\lambda[\Lambda$ .Src M]  $\bullet$   $\Lambda$ .Src N) # stdz-insert u U
      using MN 1 seq seq-char Std stdz-insert-Ide-Std [of  $\Lambda$ .subst N M u # U]
       $\Lambda$ .Ide-implies-Arr
    by (cases U = []) auto
  show ?thesis
  proof (cases U = [])
    assume U: U = []
    have 3: stdz-insert ( $\lambda[M] \bullet N$ ) (u # U) =

```

$(\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N) \# \text{standard-development } u$   
**using** 2 3  $U$  **by force**  
**have** 4:  $\Lambda.\text{seq } (\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N) (\text{hd } (\text{standard-development } u))$   
**proof**  
**show**  $\Lambda.\text{Arr } (\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N)$   
**using**  $MN$  **by simp**  
**show**  $\Lambda.\text{Arr } (\text{hd } (\text{standard-development } u))$   
**by** (*metis* 2 *Arr-imp-arr-hd Ide.simps(2) Ide-iff-standard-development-empty Std Std-consE Std-imp-Arr Std-standard-development U  $\Lambda.\text{arr-char}$   $\Lambda.\text{ide-char}$* )  
**show**  $\Lambda.\text{Trg } (\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N) = \Lambda.\text{Src } (\text{hd } (\text{standard-development } u))$   
**by** (*metis* 1 2 *Ide.simps(2) MN Src-hd-standard-development Std Std-consE Trg-last-Src-hd-eqI U  $\Lambda.\text{Ide-iff-Src-self} \Lambda.\text{Ide-implies-Arr} \Lambda.\text{Src-Subst} \Lambda.\text{Trg.simps(4)} \Lambda.\text{Trg-Src} \Lambda.\text{Trg-Subst} \Lambda.\text{ide-char last-ConsL list.sel(1) seq}$* )  
**qed**  
**show** ?thesis  
**proof** (*intro conjI*)  
**show**  $\text{Std } (\text{stdz-insert } (\lambda[M] \bullet N) (u \# U))$   
**proof** –  
**have**  $\Lambda.\text{sseq } (\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N) (\text{hd } (\text{standard-development } u))$   
**using**  $MN$  2 4  $U$   $\Lambda.\text{Ide-Src}$   
**apply** (*intro  $\Lambda.\text{sseq-BetaI}$* )  
**apply** *auto*  
**by** (*metis Ide.simps(1) Resid.simps(2) Std Std-consE Std-standard-development cong-standard-development hd-Cons-tl ide-char  $\Lambda.\text{sseq-imp-elementary-reduction1 Std.simps(2)}$* )  
**thus** ?thesis  
**by** (*metis* 3 *Std.simps(2-3) Std-standard-development hd-Cons-tl  $\Lambda.\text{sseq-imp-elementary-reduction1}$* )  
**qed**  
**show**  $\neg \text{Ide } ((\lambda[M] \bullet N) \# u \# U)$   
 $\rightarrow \text{stdz-insert } (\lambda[M] \bullet N) (u \# U) \text{ *~* } (\lambda[M] \bullet N) \# u \# U$   
**proof**  
**have**  $\text{stdz-insert } (\lambda[M] \bullet N) (u \# U) =$   
 $[\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ \text{standard-development } u$   
**using** 3 **by simp**  
**also have** 5:  $[\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ \text{standard-development } u \text{ *~* }$   
 $[\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ [u]$   
**proof** (*intro cong-append*)  
**show**  $\text{seq } [\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] (\text{standard-development } u)$   
**by** (*metis* 2 3 *Ide.simps(2) Ide-iff-standard-development-empty Std Std-consE Std-imp-Arr U  $\langle \text{Std } (\text{stdz-insert } (\Lambda.\text{Beta } M N) (u \# U)) \rangle$  arr-append-imp-seq arr-char calculation  $\Lambda.\text{ide-char neq-Nil-conv}$* )  
**thus**  $[\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] \text{ *~* } [\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N]$   
**using** *cong-reflexive by blast*  
**show**  $\text{standard-development } u \text{ *~* } [u]$   
**by** (*metis* 2 *Arr.simps(2) Ide.simps(2) Std Std-imp-Arr U cong-standard-development  $\Lambda.\text{arr-char} \Lambda.\text{ide-char not-Cons-self2}$* )  
**qed**

**also have**  $[\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ [u] \sim^*$   
 $([\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ [\Lambda.\text{subst } N M]) @ [u]$

**proof** (*intro cong-append*)

**show**  $\text{seq } [\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] [u]$   
**by** (*metis 5 Con-implies-Arr(1) Ide.simps(1) arr-append-imp-seq*  
*arr-char ide-char not-Cons-self2*)

**show**  $[\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] \sim^* [\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ [\Lambda.\text{subst } N M]$   
**by** (*metis (full-types) 1 MN Ide-iff-standard-development-empty*  
*cong-standard-development cong-transitive \Lambda.Arr.simps(5) \Lambda.Arr-Subst*  
*\Lambda.Ide.simps(5) Beta-decomp(1) standard-development.simps(5)*)

**show**  $[u] \sim^* [u]$   
**using** *Resid-Arr-self Std Std-imp-Arr U ide-char* **by** *blast*

**qed**

**also have**  $([\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ [\Lambda.\text{subst } N M]) @ [u] \sim^* [\lambda[M] \bullet N] @ [u]$   
**by** (*metis Beta-decomp(1) MN U Resid-Arr-self cong-append*  
*ide-char seq-char seq*)

**also have**  $[\lambda[M] \bullet N] @ [u] = (\lambda[M] \bullet N) \# u \# U$   
**using** *U* **by** *simp*

**finally show**  $\text{stdz-insert } (\lambda[M] \bullet N) (u \# U) \sim^* (\lambda[M] \bullet N) \# u \# U$   
**by** *blast*

**qed**

**qed**

**next**

**assume**  $U: U \neq []$

**have**  $4: \text{seq } [u] U$   
**by** (*simp add: Std U arrI\_P arr-append-imp-seq*)

**have**  $5: \text{Std } U$   
**using** *Std* **by** *auto*

**have**  $6: \text{Std } (\text{stdz-insert } u U) \wedge$   
 $\text{set } (\text{stdz-insert } u U) \subseteq \{a. \Lambda.\text{elementary-reduction } a\} \wedge$   
 $(\neg \text{Ide } (u \# U) \longrightarrow$   
 $\text{cong } (\text{stdz-insert } u U) (u \# U))$

**proof** –

**have**  $\text{seq } [\Lambda.\text{subst } N M] (u \# U) \wedge \text{Std } (u \# U)$   
**using** *MN Std Std-imp-Arr \Lambda.Arr-Subst*

**apply** (*intro conjI seqI\_{\Lambda P}*)

**apply** *simp-all*

**by** (*metis Trg-last-Src-hd-eqI \Lambda.Trg.simps(4) last-ConsL list.sel(1) seq*)

**thus** *?thesis*

**using** *MN 1 2 3 4 5 ind Std-implies-set-subset-elementary-reduction*  
*stdz-insert-Ide-Std*

**apply** *simp*

**by** (*meson cong-cons-ideI(1) cong-transitive lambda-calculus.ide-char*)

**qed**

**have**  $7: \Lambda.\text{seq } (\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N) (\text{hd } (\text{stdz-insert } u U))$   
**using** *MN 1 2 6 Arr-imp-arr-hd Con-implies-Arr(2) ide-char \Lambda.arr-char*  
*Ide-iff-standard-development-empty Src-hd-eqI Trg-last-Src-hd-eqI*  
*Trg-last-standard-development \Lambda.Ide-implies-Arr seq*

**apply** (*intro \Lambda.seqI\_{\Lambda}*)

```

    apply simp
    apply (metis Ide.simps(1))
  by (metis  $\Lambda$ .Arr.simps(5)  $\Lambda$ .Ide.simps(5) last.simps standard-development.simps(5))
  have 8: seq  $[\lambda[\Lambda$ .Src M]  $\bullet$   $\Lambda$ .Src N] (stdz-insert u U)
    by (metis 2 6 7 seqI $_{\Lambda P}$  Arr.simps(2) Con-implies-Arr(2)
        Ide.simps(1) ide-char last.simps  $\Lambda$ .seqE  $\Lambda$ .seq-char)
  show ?thesis
  proof (intro conjI)
    show Std (stdz-insert  $(\lambda[M] \bullet N)$  (u # U))
    proof -
      have  $\Lambda$ .sseq  $(\lambda[\Lambda$ .Src M]  $\bullet$   $\Lambda$ .Src N) (hd (stdz-insert u U))
        by (metis MN 2 6 7  $\Lambda$ .Ide-Src Std.elims(2) Ide.simps(1)
            Resid.simps(2) ide-char list.sel(1)  $\Lambda$ .sseq-BetaI
             $\Lambda$ .sseq-imp-elementary-reduction1)
      thus ?thesis
        by (metis 2 3 6 Std.simps(3) Resid.simps(1) con-char prfx-implies-con
            list.exhaust-sel)
    qed
  show  $\neg$  Ide  $((\lambda[M] \bullet N) \# u \# U)$ 
     $\rightarrow$  stdz-insert  $(\lambda[M] \bullet N)$  (u # U)  $^{*\sim*}$   $(\lambda[M] \bullet N) \# u \# U$ 
  proof
    have stdz-insert  $(\lambda[M] \bullet N)$  (u # U) =  $[\lambda[\Lambda$ .Src M]  $\bullet$   $\Lambda$ .Src N] @ stdz-insert u U
      using 3 by simp
    also have ...  $^{*\sim*}$   $[\lambda[\Lambda$ .Src M]  $\bullet$   $\Lambda$ .Src N] @ u # U
      using MN 2 3 6 8 cong-append
      by (meson cong-reflexive seqE)
    also have  $[\lambda[\Lambda$ .Src M]  $\bullet$   $\Lambda$ .Src N] @ u # U  $^{*\sim*}$ 
       $([\lambda[\Lambda$ .Src M]  $\bullet$   $\Lambda$ .Src N] @  $[\Lambda$ .subst N M]) @ u # U
      using MN 1 2 6 8 Beta-decomp(1) Std Src-hd-eqI Trg-last-Src-hd-eqI
       $\Lambda$ .Arr-Subst  $\Lambda$ .ide-char ide-char
    apply (intro cong-append cong-append-ideI seqI $_{\Lambda P}$ )
    apply auto[2]
    apply metis
    apply auto[4]
    by (metis cong-transitive)
    also have  $([\lambda[\Lambda$ .Src M]  $\bullet$   $\Lambda$ .Src N] @  $[\Lambda$ .subst N M]) @ u # U  $^{*\sim*}$ 
       $[\lambda[M] \bullet N] @ u \# U$ 
      by (meson MN 2 6 Beta-decomp(1) cong-append prfx-transitive seq)
    also have  $[\lambda[M] \bullet N] @ u \# U = (\lambda[M] \bullet N) \# u \# U$ 
      by simp
    finally show stdz-insert  $(\lambda[M] \bullet N)$  (u # U)  $^{*\sim*}$   $(\lambda[M] \bullet N) \# u \# U$ 
      by simp
  qed
  qed
  qed
  next
  assume 1:  $\neg$   $\Lambda$ .Ide  $(\Lambda$ .subst N M)
  have 2: stdz-insert  $(\lambda[M] \bullet N)$  (u # U) =
     $(\lambda[\Lambda$ .Src M]  $\bullet$   $\Lambda$ .Src N) # stdz-insert  $(\Lambda$ .subst N M) (u # U)

```

**using** 1 *MN* **by** *simp*  
**have** 3:  $\text{seq} [\Lambda.\text{subst } N M] (u \# U)$   
**using**  $\Lambda.\text{Arr-Subst } MN \text{ seq-char seq}$  **by** *force*  
**have** 4:  $\text{Std} (\text{stdz-insert } (\Lambda.\text{subst } N M) (u \# U)) \wedge$   
 $\text{set} (\text{stdz-insert } (\Lambda.\text{subst } N M) (u \# U)) \subseteq \{a. \Lambda.\text{elementary-reduction } a\} \wedge$   
 $\text{stdz-insert } (\Lambda.\text{Subst } 0 N M) (u \# U) \sim^* \Lambda.\text{subst } N M \# u \# U$   
**using** 1 3 *Std ind MN Ide.simps(3)  $\Lambda.\text{ide-char}$*   
 $\text{Std-implies-set-subset-elementary-reduction}$   
**by** *presburger*  
**have** 5:  $\Lambda.\text{seq} (\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N) (\text{hd} (\text{stdz-insert } (\Lambda.\text{subst } N M) (u \# U)))$   
**using** *MN 4*  
**apply** (*intro  $\Lambda.\text{seqI}_\Lambda$* )  
**apply** *simp*  
**apply** (*metis Arr-imp-arr-hd Con-implies-Arr(1) Ide.simps(1) ide-char  $\Lambda.\text{arr-char}$* )  
**using** *Src-hd-eqI*  
**by** *force*  
**show** *?thesis*  
**proof** (*intro conjI*)  
**show**  $\text{Std} (\text{stdz-insert } (\lambda[M] \bullet N) (u \# U))$   
**proof** –  
**have**  $\Lambda.\text{sseq} (\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N) (\text{hd} (\text{stdz-insert } (\Lambda.\text{subst } N M) (u \# U)))$   
**using** 5  
**by** (*metis 4 MN  $\Lambda.\text{Ide-Src Std.elims(2) Ide.simps(1) Resid.simps(2)$*   
 $\text{ide-char list.sel(1)  $\Lambda.\text{sseq-BetaI} \Lambda.\text{sseq-imp-elementary-reduction1}$$ )  
**thus** *?thesis*  
**by** (*metis 2 4 Std.simps(3) Arr.simps(1) Con-implies-Arr(2)*  
 $\text{Ide.simps(1) ide-char list.exhaust-sel}$ )  
**qed**  
**show**  $\neg \text{Ide} ((\lambda[M] \bullet N) \# u \# U)$   
 $\longrightarrow \text{stdz-insert } (\lambda[M] \bullet N) (u \# U) \sim^* (\lambda[M] \bullet N) \# u \# U$   
**proof**  
**have**  $\text{stdz-insert } (\lambda[M] \bullet N) (u \# U) =$   
 $[\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ \text{stdz-insert } (\Lambda.\text{subst } N M) (u \# U)$   
**using** 2 **by** *simp*  
**also have**  $\dots \sim^* [\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ \Lambda.\text{subst } N M \# u \# U$   
**proof** (*intro cong-append*)  
**show**  $\text{seq} [\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] (\text{stdz-insert } (\Lambda.\text{subst } N M) (u \# U))$   
**by** (*metis 4 5 Arr.simps(2) Con-implies-Arr(1) Ide.simps(1) ide-char*  
 $\Lambda.\text{arr-char} \Lambda.\text{seq-char last-ConsL seqI}_{\Lambda P}$ )  
**show**  $[\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] \sim^* [\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N]$   
**by** (*meson MN cong-transitive  $\Lambda.\text{Arr-Src Beta-decomp(1)}$* )  
**show**  $\text{stdz-insert } (\Lambda.\text{subst } N M) (u \# U) \sim^* \Lambda.\text{subst } N M \# u \# U$   
**using** 4 **by** *fastforce*  
**qed**  
**also have**  $[\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ \Lambda.\text{subst } N M \# u \# U =$   
 $([\lambda[\Lambda.\text{Src } M] \bullet \Lambda.\text{Src } N] @ [\Lambda.\text{subst } N M]) @ u \# U$   
**by** *simp*  
**also have**  $\dots \sim^* [\lambda[M] \bullet N] @ u \# U$   
**by** (*meson Beta-decomp(1) MN cong-append cong-reflexive seqE seq*)

```

also have  $[\lambda[M] \bullet N] @ u \# U = (\lambda[M] \bullet N) \# u \# U$ 
  by simp
finally show stdz-insert  $(\lambda[M] \bullet N) (u \# U) \sim^* (\lambda[M] \bullet N) \# u \# U$ 
  by blast
qed
qed
qed
qed
qed

```

Because of the way the function package processes the pattern matching in the definition of *stdz-insert*, it produces eight separate subgoals for the remainder of the proof, even though these subgoals are all simple consequences of a single, more general fact. We first prove this fact, then use it to discharge the eight subgoals.

```

have *:  $\bigwedge M N u U.$ 
   $\llbracket \neg (\Lambda.is-Lam M \wedge \Lambda.is-Beta u);$ 
   $\Lambda.Ide (M \circ N) \implies ?P (hd (u \# U)) (tl (u \# U));$ 
   $\llbracket \neg \Lambda.Ide (M \circ N);$ 
   $\Lambda.seq (M \circ N) (hd (u \# U));$ 
   $\Lambda.contains-head-reduction (M \circ N);$ 
   $\Lambda.Ide (\Lambda.resid (M \circ N) (\Lambda.head-redex (M \circ N))) \rrbracket$ 
   $\implies ?P (hd (u \# U)) (tl (u \# U));$ 
   $\llbracket \neg \Lambda.Ide (M \circ N);$ 
   $\Lambda.seq (M \circ N) (hd (u \# U));$ 
   $\Lambda.contains-head-reduction (M \circ N);$ 
   $\neg \Lambda.Ide (\Lambda.resid (M \circ N) (\Lambda.head-redex (M \circ N))) \rrbracket$ 
   $\implies ?P (\Lambda.resid (M \circ N) (\Lambda.head-redex (M \circ N))) (u \# U);$ 
   $\llbracket \neg \Lambda.Ide (M \circ N);$ 
   $\Lambda.seq (M \circ N) (hd (u \# U));$ 
   $\neg \Lambda.contains-head-reduction (M \circ N);$ 
   $\Lambda.contains-head-reduction (hd (u \# U));$ 
   $\Lambda.Ide (\Lambda.resid (M \circ N) (\Lambda.head-strategy (M \circ N))) \rrbracket$ 
   $\implies ?P (\Lambda.head-strategy (M \circ N)) (tl (u \# U));$ 
   $\llbracket \neg \Lambda.Ide (M \circ N);$ 
   $\Lambda.seq (M \circ N) (hd (u \# U));$ 
   $\neg \Lambda.contains-head-reduction (M \circ N);$ 
   $\Lambda.contains-head-reduction (hd (u \# U));$ 
   $\neg \Lambda.Ide (\Lambda.resid (M \circ N) (\Lambda.head-strategy (M \circ N))) \rrbracket$ 
   $\implies ?P (\Lambda.resid (M \circ N) (\Lambda.head-strategy (M \circ N))) (tl (u \# U));$ 
   $\llbracket \neg \Lambda.Ide (M \circ N);$ 
   $\Lambda.seq (M \circ N) (hd (u \# U));$ 
   $\neg \Lambda.contains-head-reduction (M \circ N);$ 
   $\neg \Lambda.contains-head-reduction (hd (u \# U)) \rrbracket$ 
   $\implies ?P M (filter notIde (map \Lambda.un-App1 (u \# U)));$ 
   $\llbracket \neg \Lambda.Ide (M \circ N);$ 
   $\Lambda.seq (M \circ N) (hd (u \# U));$ 
   $\neg \Lambda.contains-head-reduction (M \circ N);$ 
   $\neg \Lambda.contains-head-reduction (hd (u \# U)) \rrbracket$ 
   $\implies ?P N (filter notIde (map \Lambda.un-App2 (u \# U))) \rrbracket$ 

```

```

    ⇒ ?P (M ◦ N) (u # U)
proof –
  fix M N u U
  assume ind1: Λ.Ide (M ◦ N) ⇒ ?P (hd (u # U)) (tl (u # U))
  assume ind2: [¬ Λ.Ide (M ◦ N);
    Λ.seq (M ◦ N) (hd (u # U));
    Λ.contains-head-reduction (M ◦ N);
    Λ.Ide (Λ.resid (M ◦ N) (Λ.head-redex (M ◦ N)))]
    ⇒ ?P (hd (u # U)) (tl (u # U))
  assume ind3: [¬ Λ.Ide (M ◦ N);
    Λ.seq (M ◦ N) (hd (u # U));
    Λ.contains-head-reduction (M ◦ N);
    ¬ Λ.Ide (Λ.resid (M ◦ N) (Λ.head-redex (M ◦ N)))]
    ⇒ ?P (Λ.resid (M ◦ N) (Λ.head-redex (M ◦ N))) (u # U)
  assume ind4: [¬ Λ.Ide (M ◦ N);
    Λ.seq (M ◦ N) (hd (u # U));
    ¬ Λ.contains-head-reduction (M ◦ N);
    Λ.contains-head-reduction (hd (u # U));
    Λ.Ide (Λ.resid (M ◦ N) (Λ.head-strategy (M ◦ N)))]
    ⇒ ?P (Λ.head-strategy (M ◦ N)) (tl (u # U))
  assume ind5: [¬ Λ.Ide (M ◦ N);
    Λ.seq (M ◦ N) (hd (u # U));
    ¬ Λ.contains-head-reduction (M ◦ N);
    Λ.contains-head-reduction (hd (u # U));
    ¬ Λ.Ide (Λ.resid (M ◦ N) (Λ.head-strategy (M ◦ N)))]
    ⇒ ?P (Λ.resid (M ◦ N) (Λ.head-strategy (M ◦ N))) (tl (u # U))
  assume ind7: [¬ Λ.Ide (M ◦ N);
    Λ.seq (M ◦ N) (hd (u # U));
    ¬ Λ.contains-head-reduction (M ◦ N);
    ¬ Λ.contains-head-reduction (hd (u # U))]
    ⇒ ?P M (filter notIde (map Λ.un-App1 (u # U)))
  assume ind8: [¬ Λ.Ide (M ◦ N);
    Λ.seq (M ◦ N) (hd (u # U));
    ¬ Λ.contains-head-reduction (M ◦ N);
    ¬ Λ.contains-head-reduction (hd (u # U))]
    ⇒ ?P N (filter notIde (map Λ.un-App2 (u # U)))
  assume *: ¬ (Λ.is-Lam M ∧ Λ.is-Beta u)
  show ?P (M ◦ N) (u # U)
  proof (intro impI, elim conjE)
    assume seq: seq [M ◦ N] (u # U)
    assume Std: Std (u # U)
    have MN: Λ.Arr M ∧ Λ.Arr N
      using seq-char seq by force
    have u: Λ.Arr u
      using Std
      by (meson Std-imp-Arr Arr.simps(2) Con-Arr-self Con-implies-Arr(1)
        Con-initial-left Λ.arr-char list.simps(3))
    have U ≠ [] ⇒ Arr U
      using Std Std-imp-Arr Arr.simps(3)

```

```

by (metis Arr.elims(3) list.discI)
have  $\Lambda.is\text{-}App\ u \vee \Lambda.is\text{-}Beta\ u$ 
  using * seq MN u seq-char  $\Lambda.arr\text{-}char\ Srcs\text{-}simp_{\Lambda P}\ \Lambda.targets\text{-}char_{\Lambda}$ 
  by (cases M; cases u) auto
have **:  $\Lambda.seq\ (M \circ N)\ u$ 
  using Srcs-simp $_{\Lambda P}$  seq-char seq  $\Lambda.seq\text{-}def\ u$  by force
show Std (stdz-insert (M  $\circ$  N) (u # U))  $\wedge$ 
  ( $\neg\ Ide\ ((M \circ N)\ \# u\ \# U)$ 
    $\longrightarrow\ cong\ (stdz\text{-}insert\ (M \circ N)\ (u\ \# U))\ ((M \circ N)\ \# u\ \# U)$ )
proof (cases  $\Lambda.Ide\ (M \circ N)$ )
  assume 1:  $\Lambda.Ide\ (M \circ N)$ 
  have MN:  $\Lambda.Arr\ M \wedge \Lambda.Arr\ N \wedge \Lambda.Ide\ M \wedge \Lambda.Ide\ N$ 
  using MN 1 by simp
  have 2: stdz-insert (M  $\circ$  N) (u # U) = stdz-insert u U
  using MN 1
  by (simp add: Std seq stdz-insert-Ide-Std)
show ?thesis
proof (cases  $U = []$ )
  assume U:  $U = []$ 
  have 2: stdz-insert (M  $\circ$  N) (u # U) = standard-development u
  using 1 2 U by simp
  show ?thesis
proof (intro conjI)
  show Std (stdz-insert (M  $\circ$  N) (u # U))
  using 2 Std-standard-development by presburger
  show  $\neg\ Ide\ ((M \circ N)\ \# u\ \# U) \longrightarrow$ 
    stdz-insert (M  $\circ$  N) (u # U)  $\sim^*$  (M  $\circ$  N) # u # U
  by (metis 1 2 Ide.simps(2) U cong-cons-ideI(1) cong-standard-development
    ide-backward-stable ide-char  $\Lambda.ide\text{-}char\ prfx\text{-}transitive\ seq\ u$ )
qed
next
assume U:  $U \neq []$ 
have 2: stdz-insert (M  $\circ$  N) (u # U) = stdz-insert u U
  using 1 2 U by simp
have 3: seq [u] U
  by (simp add: Std U arrI $_P$  arr-append-imp-seq)
have 4: Std (stdz-insert u U)  $\wedge$ 
  set (stdz-insert u U)  $\subseteq \{a.\ \Lambda.elementary\text{-}reduction\ a\} \wedge$ 
  ( $\neg\ Ide\ (u\ \# U) \longrightarrow\ cong\ (stdz\text{-}insert\ u\ U)\ (u\ \# U)$ )
  using MN 3 Std ind1 Std-implies-set-subset-elementary-reduction
  by (metis 1 Std.simps(3) U list.sel(1) list.sel(3) standardize.cases)
show ?thesis
proof (intro conjI)
  show Std (stdz-insert (M  $\circ$  N) (u # U))
  by (metis 1 2 3 Std Std.simps(3) U ind1 list.exhaust-sel list.sel(1,3))
  show  $\neg\ Ide\ ((M \circ N)\ \# u\ \# U) \longrightarrow$ 
    stdz-insert (M  $\circ$  N) (u # U)  $\sim^*$  (M  $\circ$  N) # u # U
proof
  assume 5:  $\neg\ Ide\ ((M \circ N)\ \# u\ \# U)$ 

```

```

have stdz-insert (M ◦ N) (u # U) *~* u # U
  using 1 2 4 5 seq-char seq by force
also have u # U *~* [M ◦ N] @ u # U
  using 1 Ide.simps(2) cong-append-ideI(1) ide-char seq by blast
also have [M ◦ N] @ (u # U) = (M ◦ N) # u # U
  by simp
finally show stdz-insert (M ◦ N) (u # U) *~* (M ◦ N) # u # U
  by blast
qed
qed
qed
next
assume 1: ¬ Λ.Ide (M ◦ N)
show ?thesis
proof (cases Λ.contains-head-reduction (M ◦ N))
  assume 2: Λ.contains-head-reduction (M ◦ N)
  show ?thesis
proof (cases Λ.Ide ((M ◦ N) \ Λ.head-redex (M ◦ N)))
  assume 3: Λ.Ide ((M ◦ N) \ Λ.head-redex (M ◦ N))
  have 4: ¬ Ide (u # U)
    by (metis Std Std-implies-set-subset-elementary-reduction in-mono
      Λ.elementary-reduction-not-ide list.set-intros(1) mem-Collect-eq
      set-Ide-subset-ide)
  have 5: stdz-insert (M ◦ N) (u # U) = Λ.head-redex (M ◦ N) # stdz-insert u U
    using MN 1 2 3 4 ** by auto
  show ?thesis
proof (cases U = [])
  assume U: U = []
  have u: Λ.Arr u ∧ ¬ Λ.Ide u
    using 4 U u by force
  have 5: stdz-insert (M ◦ N) (u # U) =
    Λ.head-redex (M ◦ N) # standard-development u
    using 5 U by simp
  show ?thesis
proof (intro conjI)
  show Std (stdz-insert (M ◦ N) (u # U))
  proof –
    have Λ.sseq (Λ.head-redex (M ◦ N)) (hd (standard-development u))
    proof –
      have Λ.seq (Λ.head-redex (M ◦ N)) (hd (standard-development u))
      proof
        show Λ.Arr (Λ.head-redex (M ◦ N))
          using MN Λ.Arr.simps(4) Λ.Arr-head-redex by presburger
        show Λ.Arr (hd (standard-development u))
          using Arr-imp-arr-hd Ide-iff-standard-development-empty
            Std-standard-development u
        by force
      show Λ.Trq (Λ.head-redex (M ◦ N)) = Λ.Src (hd (standard-development u))
      proof –

```

```

have  $\Lambda.$ Trg ( $\Lambda.$ head-redex ( $M \circ N$ )) =
   $\Lambda.$ Trg ( $(M \circ N) \setminus \Lambda.$ head-redex ( $M \circ N$ ))
by (metis 3 MN  $\Lambda.$ Con-Arr-head-redex  $\Lambda.$ Src-resid
   $\Lambda.$ Arr.simps(4)  $\Lambda.$ Ide-iff-Src-self  $\Lambda.$ Ide-iff-Trg-self
   $\Lambda.$ Ide-implies-Arr)
also have ... =  $\Lambda.$ Src u
using MN
by (metis Trg-last-Src-hd-eqI Trg-last-eqI head-redex-decomp
   $\Lambda.$ Arr.simps(4) last-ConsL last-appendR list.sel(1)
  not-Cons-self2 seq)
also have ... =  $\Lambda.$ Src (hd (standard-development u))
using ** 2 3 u MN Src-hd-standard-development [of u] by metis
finally show ?thesis by blast
qed
qed
thus ?thesis
by (metis 2 u MN  $\Lambda.$ Arr.simps(4) Ide-iff-standard-development-empty
  development.simps(2) development-standard-development
   $\Lambda.$ head-redex-is-head-reduction list.exhaust-sel
   $\Lambda.$ sseq-head-reductionI)
qed
thus ?thesis
by (metis 5 Ide-iff-standard-development-empty Std.simps(3)
  Std-standard-development list.exhaust u)
qed
show  $\neg$  Ide ( $(M \circ N) \# u \# U$ )  $\longrightarrow$ 
  stdz-insert ( $M \circ N$ ) ( $u \# U$ )  $*\sim*$  ( $M \circ N$ )  $\# u \# U$ 
proof
have stdz-insert ( $M \circ N$ ) ( $u \# U$ ) =
  [ $\Lambda.$ head-redex ( $M \circ N$ )] @ standard-development u
using 5 by simp
also have ...  $*\sim*$  [ $\Lambda.$ head-redex ( $M \circ N$ )] @ [u]
using u cong-standard-development [of u] cong-append
by (metis 2 5 Ide-iff-standard-development-empty Std-imp-Arr
   $\langle$ Std (stdz-insert ( $M \circ N$ ) ( $u \# U$ )) $\rangle$ 
  arr-append-imp-seq arr-char calculation cong-standard-development
  cong-transitive  $\Lambda.$ Arr-head-redex  $\Lambda.$ contains-head-reduction-iff
  list.distinct(1))
also have [ $\Lambda.$ head-redex ( $M \circ N$ )] @ [u]  $*\sim*$ 
  ([ $\Lambda.$ head-redex ( $M \circ N$ )] @ [( $M \circ N$ ) \  $\Lambda.$ head-redex ( $M \circ N$ )]]) @ [u]
proof –
have [ $\Lambda.$ head-redex ( $M \circ N$ )]  $*\sim*$ 
  [ $\Lambda.$ head-redex ( $M \circ N$ )] @ [( $M \circ N$ ) \  $\Lambda.$ head-redex ( $M \circ N$ )]
by (metis (no-types, lifting) 1 3 MN Arr-iff-Con-self Ide.simps(2)
  Resid.simps(2) arr-append-imp-seq arr-char cong-append-ideI(4)
  cong-transitive head-redex-decomp ide-backward-stable ide-char
   $\Lambda.$ Arr.simps(4)  $\Lambda.$ ide-char not-Cons-self2)
thus ?thesis
using MN U u seq

```

by (*meson cong-append head-redex-decomp*  $\Lambda$ .Arr.simps(4) *prfx-transitive*)  
 qed  
 also have  $([\Lambda$ .head-redex  $(M \circ N)] @$   
 $[(M \circ N) \setminus \Lambda$ .head-redex  $(M \circ N)]) @ [u] \sim^*$   
 $[M \circ N] @ [u]$   
 by (*metis*  $\Lambda$ .Arr.simps(4) *MN U Resid-Arr-self cong-append ide-char*  
*seq-char head-redex-decomp seq*)  
 also have  $[M \circ N] @ [u] = (M \circ N) \# u \# U$   
 using *U* by *simp*  
 finally show  $stdz$ -insert  $(M \circ N) (u \# U) \sim^* (M \circ N) \# u \# U$   
 by *blast*  
 qed  
 qed  
 next  
 assume *U*:  $U \neq []$   
 have 6: *Std* ( $stdz$ -insert *u U*)  $\wedge$   
 $set (stdz$ -insert *u U*)  $\subseteq \{a. \Lambda$ .elementary-reduction *a*  $\}$   $\wedge$   
 $cong (stdz$ -insert *u U*) ( $u \# U$ )  
 proof –  
 have *seq* [*u*] *U*  
 by (*simp add*: *Std U arrI<sub>P</sub> arr-append-imp-seq*)  
 moreover have *Std U*  
 using *Std Std.elims*(2) *U* by *blast*  
 ultimately show *?thesis*  
 using *ind2 \*\* 1 2 3 4 Std-implies-set-subset-elementary-reduction*  
 by *force*  
 qed  
 show *?thesis*  
 proof (*intro conjI*)  
 show *Std* ( $stdz$ -insert  $(M \circ N) (u \# U)$ )  
 proof –  
 have  $\Lambda$ .sseq  $(\Lambda$ .head-redex  $(M \circ N)) (hd (stdz$ -insert *u U*)  
 proof –  
 have  $\Lambda$ .seq  $(\Lambda$ .head-redex  $(M \circ N)) (hd (stdz$ -insert *u U*)  
 proof  
 show  $\Lambda$ .Arr  $(\Lambda$ .head-redex  $(M \circ N))$   
 using *MN*  $\Lambda$ .Arr-head-redex by *force*  
 show  $\Lambda$ .Arr  $(hd (stdz$ -insert *u U*)  
 using 6  
 by (*metis* *Arr-imp-arr-hd Con-implies-Arr*(2) *Ide.simps*(1) *ide-char*  
 $\Lambda$ .arr-char)  
 show  $\Lambda$ .Trg  $(\Lambda$ .head-redex  $(M \circ N)) = \Lambda$ .Src  $(hd (stdz$ -insert *u U*)  
 proof –  
 have  $\Lambda$ .Trg  $(\Lambda$ .head-redex  $(M \circ N)) =$   
 $\Lambda$ .Trg  $((M \circ N) \setminus \Lambda$ .head-redex  $(M \circ N))$   
 by (*metis* 3  $\Lambda$ .Arr-not-Nil  $\Lambda$ .Ide-iff-*Src-self*  
 $\Lambda$ .Ide-iff-*Trg-self*  $\Lambda$ .Ide-implies-Arr  $\Lambda$ .Src-resid)  
 also have ... =  $\Lambda$ .Trg  $(M \circ N)$   
 by (*metis* 1 *MN Trg-last-eqI Trg-last-standard-development*)

$\text{cong-standard-development head-redex-decomp } \Lambda.\text{Arr.simps}(4)$   
 $\text{last-snoc}$   
**also have**  $\dots = \Lambda.\text{Src} (\text{hd} (\text{stdz-insert } u \ U))$   
**by**  $(\text{metis} \ ** \ 6 \ \text{Src-hd-eqI} \ \Lambda.\text{seqE}_\Lambda \ \text{list.sel}(1))$   
**finally show**  $?thesis$  **by**  $\text{blast}$   
**qed**  
**qed**  
**thus**  $?thesis$   
**by**  $(\text{metis} \ 2 \ 6 \ MN \ \Lambda.\text{Arr.simps}(4) \ \text{Std.elims}(1) \ \text{Ide.simps}(1)$   
 $\text{Resid.simps}(2) \ \text{ide-char} \ \Lambda.\text{head-redex-is-head-reduction}$   
 $\text{list.sel}(1) \ \Lambda.\text{sseq-head-reductionI} \ \Lambda.\text{sseq-imp-elementary-reduction1})$   
**qed**  
**thus**  $?thesis$   
**by**  $(\text{metis} \ 5 \ 6 \ \text{Std.simps}(3) \ \text{Arr.simps}(1) \ \text{Con-implies-Arr}(1)$   
 $\text{con-char} \ \text{prfx-implies-con} \ \text{list.exhaust-sel})$   
**qed**  
**show**  $\neg \text{Ide} ((M \circ N) \ \# \ u \ \# \ U) \longrightarrow$   
 $\text{stdz-insert} (M \circ N) (u \ \# \ U) \ * \sim \ * (M \circ N) \ \# \ u \ \# \ U$   
**proof**  
**have**  $\text{stdz-insert} (M \circ N) (u \ \# \ U) =$   
 $[\Lambda.\text{head-redex} (M \circ N)] \ @ \ \text{stdz-insert } u \ U$   
**using**  $5$  **by**  $\text{simp}$   
**also have**  $7: [\Lambda.\text{head-redex} (M \circ N)] \ @ \ \text{stdz-insert } u \ U \ * \sim \ *$   
 $[\Lambda.\text{head-redex} (M \circ N)] \ @ \ u \ \# \ U$   
**using**  $6$   $\text{cong-append} [\text{of } [\Lambda.\text{head-redex} (M \circ N)] \ \text{stdz-insert } u \ U$   
 $[\Lambda.\text{head-redex} (M \circ N)] \ u \ \# \ U]$   
**by**  $(\text{metis} \ 2 \ 5 \ \text{Arr.simps}(1) \ \text{Resid.simps}(2) \ \text{Std-imp-Arr}$   
 $\langle \text{Std} (\text{stdz-insert} (M \circ N) (u \ \# \ U)) \rangle$   
 $\text{arr-append-imp-seq} \ \text{arr-char} \ \text{calculation} \ \text{cong-standard-development}$   
 $\text{cong-transitive} \ \text{ide-implies-arr} \ \Lambda.\text{Arr-head-redex}$   
 $\Lambda.\text{contains-head-reduction-iff} \ \text{list.distinct}(1))$   
**also have**  $[\Lambda.\text{head-redex} (M \circ N)] \ @ \ u \ \# \ U \ * \sim \ *$   
 $([\Lambda.\text{head-redex} (M \circ N)] \ @$   
 $[(M \circ N) \setminus \Lambda.\text{head-redex} (M \circ N)]) \ @ \ u \ \# \ U$   
**proof**  $-$   
**have**  $[\Lambda.\text{head-redex} (M \circ N)] \ * \sim \ *$   
 $[\Lambda.\text{head-redex} (M \circ N)] \ @ \ [(M \circ N) \setminus \Lambda.\text{head-redex} (M \circ N)]$   
**by**  $(\text{metis} \ 2 \ 3 \ \text{head-redex-decomp} \ \Lambda.\text{Arr-head-redex}$   
 $\Lambda.\text{Con-Arr-head-redex} \ \Lambda.\text{Ide-iff-Src-self} \ \Lambda.\text{Ide-implies-Arr}$   
 $\Lambda.\text{Src-resid} \ \Lambda.\text{contains-head-reduction-iff} \ \Lambda.\text{resid-Arr-self}$   
 $\text{prfx-decomp} \ \text{prfx-transitive})$   
**moreover have**  $\text{seq} [\Lambda.\text{head-redex} (M \circ N)] (u \ \# \ U)$   
**by**  $(\text{metis} \ 7 \ \text{arr-append-imp-seq} \ \text{cong-implies-coterminal} \ \text{coterminalE}$   
 $\text{list.distinct}(1))$   
**ultimately show**  $?thesis$   
**using**  $3$   $\text{ide-char} \ \text{cong-symmetric} \ \text{cong-append}$   
**by**  $(\text{meson} \ 6 \ \text{prfx-transitive})$   
**qed**  
**also have**  $([\Lambda.\text{head-redex} (M \circ N)] \ @$

$$[(M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N)] @ u \# U * \sim^*$$

$$[M \circ N] @ u \# U$$
**by** (*meson 6 MN  $\Lambda$ .Arr.simps(4) cong-append prfx-transitive head-redex-decomp seq*)

**also have**  $[M \circ N] @ (u \# U) = (M \circ N) \# u \# U$   
**by** *simp*

**finally show**  $\text{stdz-insert } (M \circ N) (u \# U) * \sim^* (M \circ N) \# u \# U$   
**by** *blast*

**qed**  
**qed**  
**qed**  
**next**

**assume**  $3: \neg \Lambda.\text{Ide } ((M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N))$

**have**  $4: \text{stdz-insert } (M \circ N) (u \# U) =$   
 $\Lambda.\text{head-redex } (M \circ N) \#$   
 $\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N)) (u \# U)$

**using** *MN 1 2 3 \*\* by auto*

**have**  $5: \text{Std } (\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N)) (u \# U)) \wedge$   
 $\text{set } (\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N)) (u \# U))$   
 $\subseteq \{a. \Lambda.\text{elementary-reduction } a\} \wedge$   
 $\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N)) (u \# U) * \sim^*$   
 $(M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N) \# u \# U$

**proof** –

**have**  $\text{seq } [(M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N)] (u \# U)$   
**by** (*metis (full-types) MN arr-append-imp-seq cong-implies-coterminal coterminalE head-redex-decomp  $\Lambda$ .Arr.simps(4) not-Cons-self2 seq seq-def targets-append*)

**thus** *?thesis*  
**using** *ind3 1 2 3 \*\* Std Std-implies-set-subset-elementary-reduction by auto*

**qed**

**show** *?thesis*

**proof** (*intro conjI*)

**show**  $\text{Std } (\text{stdz-insert } (M \circ N) (u \# U))$

**proof** –

**have**  $\Lambda.\text{sseq } (\Lambda.\text{head-redex } (M \circ N))$   
 $(\text{hd } (\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N)) (u \# U)))$

**proof** –

**have**  $\Lambda.\text{seq } (\Lambda.\text{head-redex } (M \circ N))$   
 $(\text{hd } (\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N)) (u \# U)))$

**using** *MN 5  $\Lambda$ .Arr-head-redex*

**by** (*metis (no-types, lifting) Arr-imp-arr-hd Con-implies-Arr(2) Ide.simps(1) Src-hd-eqI ide-char  $\Lambda$ .Arr.simps(4)  $\Lambda$ .Arr-head-redex  $\Lambda$ .Con-Arr-head-redex  $\Lambda$ .Src-resid  $\Lambda$ .arr-char  $\Lambda$ .seq-char list.sel(1)*)

**moreover have**  $\Lambda.\text{elementary-reduction}$   
 $(\text{hd } (\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-redex } (M \circ N)) (u \# U)))$

**using** *5*

**by** (*metis Arr.simps(1) Con-implies-Arr(2) Ide.simps(1) hd-in-set*)

```

      ide-char mem-Collect-eq subset-code(1))
    ultimately show ?thesis
      using MN 2  $\Lambda$ .head-redex-is-head-reduction  $\Lambda$ .sseq-head-reductionI
      by simp
    qed
  thus ?thesis
    by (metis 4 5 Std.simps(3) Arr.simps(1) Con-implies-Arr(2)
        Ide.simps(1) ide-char list.exhaust-sel)
  qed
  show  $\neg$  Ide  $((M \circ N) \# u \# U) \longrightarrow$ 
    stdz-insert  $(M \circ N) (u \# U) \sim^* (M \circ N) \# u \# U$ 
  proof
    have stdz-insert  $(M \circ N) (u \# U) =$ 
       $[\Lambda$ .head-redex  $(M \circ N)] @$ 
      stdz-insert  $((M \circ N) \setminus \Lambda$ .head-redex  $(M \circ N)) (u \# U)$ 
    using 4 by simp
    also have ...  $\sim^* [\Lambda$ .head-redex  $(M \circ N)] @$ 
       $((M \circ N) \setminus \Lambda$ .head-redex  $(M \circ N) \# u \# U)$ 
  proof (intro cong-append)
    show seq  $[\Lambda$ .head-redex  $(M \circ N)]$ 
       $(stdz-insert ((M \circ N) \setminus \Lambda$ .head-redex  $(M \circ N)) (u \# U))$ 
    by (metis 4 5 Ide.simps(1) Resid.simps(1) Std-imp-Arr
         $\langle$ Std  $(stdz-insert (M \circ N) (u \# U))\rangle$  arrIP arr-append-imp-seq
        calculation ide-char list.discI)
    show  $[\Lambda$ .head-redex  $(M \circ N)] \sim^* [\Lambda$ .head-redex  $(M \circ N)]$ 
      using MN  $\Lambda$ .cong-reflexive ide-char  $\Lambda$ .Arr-head-redex by force
    show stdz-insert  $((M \circ N) \setminus \Lambda$ .head-redex  $(M \circ N)) (u \# U) \sim^* (M \circ N) \setminus$ 
       $\Lambda$ .head-redex  $(M \circ N) \# u \# U$ 
      using 5 by fastforce
  qed
  also have  $([\Lambda$ .head-redex  $(M \circ N)] @$ 
     $((M \circ N) \setminus \Lambda$ .head-redex  $(M \circ N) \# u \# U)) =$ 
     $([\Lambda$ .head-redex  $(M \circ N)] @$ 
     $[(M \circ N) \setminus \Lambda$ .head-redex  $(M \circ N)]) @ (u \# U)$ 
    by simp
  also have  $([\Lambda$ .head-redex  $(M \circ N)] @$ 
     $[(M \circ N) \setminus \Lambda$ .head-redex  $(M \circ N)]) @ u \# U \sim^*$ 
     $[M \circ N] @ u \# U$ 
    by (meson ** cong-append cong-reflexive seqE head-redex-decomp
        seq  $\Lambda$ .seq-char)
  also have  $[M \circ N] @ (u \# U) = (M \circ N) \# u \# U$ 
    by simp
  finally show stdz-insert  $(M \circ N) (u \# U) \sim^* (M \circ N) \# u \# U$ 
    by blast
  qed
  qed
  next
  assume 2:  $\neg \Lambda$ .contains-head-reduction  $(M \circ N)$ 

```

**show** *?thesis*  
**proof** (*cases*  $\Lambda$ .contains-head-reduction  $u$ )  
**assume**  $\exists$ :  $\Lambda$ .contains-head-reduction  $u$   
**have**  $B$ : [ $\Lambda$ .head-strategy ( $M \circ N$ )] @ [( $M \circ N$ ) \  $\Lambda$ .head-strategy ( $M \circ N$ )]  $\sim^*$   
 $[M \circ N]$  @ [ $u$ ]  
**proof** –  
**have** [ $M \circ N$ ] @ [ $u$ ]  $\sim^*$  [ $\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))  $\sqcup$   $M \circ N$ ]  
**proof** –  
**have**  $\Lambda$ .is-internal-reduction ( $M \circ N$ )  
**using**  $\mathcal{2}$  \*\*  $\Lambda$ .is-internal-reduction-iff **by** *blast*  
**moreover have**  $\Lambda$ .is-head-reduction  $u$   
**proof** –  
**have**  $\Lambda$ .elementary-reduction  $u$   
**by** (*metis* *Std* *lambda-calculus.sseq-imp-elementary-reduction1*  
*list.discI* *list.sel(1)* *reduction-paths.Std.elims(2)*)  
**thus** *?thesis*  
**using**  $\Lambda$ .is-head-reduction-if  $\exists$  **by** *force*  
**qed**  
**moreover have**  $\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ )) \ ( $M \circ N$ ) =  $u$   
**using**  $\Lambda$ .resid-head-strategy-Src(1) \*\* *calculation(1-2)* **by** *fastforce*  
**moreover have** [ $M \circ N$ ]  $\lesssim^*$  [ $\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))  $\sqcup$   $M \circ N$ ]  
**using**  $MN$   $\Lambda$ .prfx-implies-con *ide-char*  $\Lambda$ .Arr-head-strategy  
 $\Lambda$ .Src-head-strategy  $\Lambda$ .prfx-Join  
**by** *force*  
**ultimately show** *?thesis*  
**using**  $u$   $\Lambda$ .Coinitial-iff-Con  $\Lambda$ .Arr-not-Nil  $\Lambda$ .resid-Join  
*prfx-decomp* [*of*  $M \circ N$   $\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))  $\sqcup$   $M \circ N$ ]  
**by** *simp*  
**qed**  
**also have** [ $\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))  $\sqcup$   $M \circ N$ ]  $\sim^*$   
 $[\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))]  
@  
 $[(M \circ N) \setminus \Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))]  
**proof** –  
**have**  $\exists$ :  $\Lambda$ .composite-of  
 $(\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ )))  
 $((M \circ N) \setminus \Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ )))  
 $(\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))  $\sqcup$   $M \circ N$ )  
**using**  $\Lambda$ .Arr-head-strategy  $MN$   $\Lambda$ .Src-head-strategy  $\Lambda$ .join-of-Join  
 $\Lambda$ .join-of-def  
**by** *force*  
**hence** *composite-of*  
 $[\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))]  
 $[(M \circ N) \setminus \Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))]  
 $[\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))  $\sqcup$   $M \circ N$ ]  
**using** *composite-of-single-single*  
**by** (*metis* (*no-types*, *lifting*)  $\Lambda$ .Con-sym *Ide.simps(2)* *Resid.simps(3)*  
*composite-ofI*  $\Lambda$ .composite-ofE  $\Lambda$ .con-char *ide-char*  $\Lambda$ .prfx-implies-con)  
**hence** [ $\Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))]  
@  
 $[(M \circ N) \setminus \Lambda$ .head-strategy ( $\Lambda$ .Src ( $M \circ N$ ))]  
 $\sim^*$

```

    [Λ.head-strategy (Λ.Src (M ◦ N)) ⊔ M ◦ N]
  using Λ.resid-Join
  by (meson 3 composite-of-single-single composite-of-unq-upto-cong)
  thus ?thesis by blast
qed
also have [Λ.head-strategy (Λ.Src (M ◦ N))] @
  [(M ◦ N) \ Λ.head-strategy (Λ.Src (M ◦ N))] *~*
  [Λ.head-strategy (M ◦ N)] @
  [(M ◦ N) \ Λ.head-strategy (M ◦ N)]
  by (metis (full-types) Λ.Arr.simps(4) MN prfx-transitive calculation
    Λ.head-strategy-Src)
  finally show ?thesis by blast
qed
show ?thesis
proof (cases Λ.Ide ((M ◦ N) \ Λ.head-strategy (M ◦ N)))
  assume 4: Λ.Ide ((M ◦ N) \ Λ.head-strategy (M ◦ N))
  have A: [Λ.head-strategy (M ◦ N)] *~*
    [Λ.head-strategy (M ◦ N)] @ [(M ◦ N) \ Λ.head-strategy (M ◦ N)]
  by (meson 4 B Con-implies-Arr(1) Ide.simps(2) arr-append-imp-seq arr-char
    con-char cong-append-ideI(2) ide-char Λ.ide-char not-Cons-self2
    prfx-implies-con)
  have 5: ¬ Ide (u # U)
  by (meson 3 Ide-consE Λ.ide-backward-stable Λ.subs-head-redex
    Λ.subs-implies-prfx Λ.contains-head-reduction-iff
    Λ.elementary-reduction-head-redex Λ.elementary-reduction-not-ide)
  have 6: stdz-insert (M ◦ N) (u # U) =
    stdz-insert (Λ.head-strategy (M ◦ N)) U
  using 1 2 3 4 5 * ** ‹Λ.is-App u ∨ Λ.is-Beta u›
  apply (cases u)
  apply simp-all
  apply blast
  by (cases M) auto
show ?thesis
proof (cases U = [])
  assume U: U = []
  have u: ¬ Λ.Ide u
  using 5 U by simp
  have 6: stdz-insert (M ◦ N) (u # U) =
    standard-development (Λ.head-strategy (M ◦ N))
  using 6 U by simp
show ?thesis
proof (intro conjI)
  show Std (stdz-insert (M ◦ N) (u # U))
  using 6 Std-standard-development by presburger
  show ¬ Ide ((M ◦ N) # u # U) →
    stdz-insert (M ◦ N) (u # U) *~* (M ◦ N) # u # U
proof
  have stdz-insert (M ◦ N) (u # U) *~* [Λ.head-strategy (M ◦ N)]
  using 4 6 cong-standard-development ** 1 2 3 Λ.Arr.simps(4)

```

```

       $\Lambda$ .Arr-head-strategy MN  $\Lambda$ .ide-backward-stable  $\Lambda$ .ide-char
    by metis
  also have [ $\Lambda$ .head-strategy (M  $\circ$  N)]  $\ast\sim\ast$  [M  $\circ$  N] @ [u]
    by (meson A B prfx-transitive)
  also have [M  $\circ$  N] @ [u] = (M  $\circ$  N) # u # U
    using U by auto
  finally show stdz-insert (M  $\circ$  N) (u # U)  $\ast\sim\ast$  (M  $\circ$  N) # u # U
    by blast
qed
qed
next
assume U: U  $\neq$  []
have  $\gamma$ : seq [ $\Lambda$ .head-strategy (M  $\circ$  N)] U
proof
  show Arr [ $\Lambda$ .head-strategy (M  $\circ$  N)]
    by (meson A Con-implies-Arr(1) con-char prfx-implies-con)
  show Arr U
    using U  $\langle$  U  $\neq$  []  $\implies$  Arr U  $\rangle$  by presburger
  show  $\Lambda$ .Trg (last [ $\Lambda$ .head-strategy (M  $\circ$  N)]) =  $\Lambda$ .Src (hd U)
    by (metis A B Std Std-consE Trg-last-eqI U  $\Lambda$ .seqE $\Lambda$   $\Lambda$ .sseq-imp-seq last-snoc)
qed
have 8: Std (stdz-insert ( $\Lambda$ .head-strategy (M  $\circ$  N)) U)  $\wedge$ 
  set (stdz-insert ( $\Lambda$ .head-strategy (M  $\circ$  N)) U)
     $\subseteq$  {a.  $\Lambda$ .elementary-reduction a}  $\wedge$ 
  stdz-insert ( $\Lambda$ .head-strategy (M  $\circ$  N)) U  $\ast\sim\ast$ 
   $\Lambda$ .head-strategy (M  $\circ$  N) # U
proof -
  have Std U
    by (metis Std Std.simps(3) U list.exhaust-sel)
  moreover have  $\neg$  Ide ( $\Lambda$ .head-strategy (M  $\circ$  N) # tl (u # U))
    using 1 4  $\Lambda$ .ide-backward-stable by blast
  ultimately show ?thesis
    using ind4  $\ast\ast$  1 2 3 4 7 Std-implies-set-subset-elementary-reduction
    by force
qed
show ?thesis
proof (intro conjI)
  show Std (stdz-insert (M  $\circ$  N) (u # U))
    using 6 8 by presburger
  show  $\neg$  Ide ((M  $\circ$  N) # u # U)  $\longrightarrow$ 
    stdz-insert (M  $\circ$  N) (u # U)  $\ast\sim\ast$  (M  $\circ$  N) # u # U
proof
  have stdz-insert (M  $\circ$  N) (u # U) =
    stdz-insert ( $\Lambda$ .head-strategy (M  $\circ$  N)) U
  using 6 by simp
  also have ...  $\ast\sim\ast$  [ $\Lambda$ .head-strategy (M  $\circ$  N)] @ U
  using 8 by simp
  also have [ $\Lambda$ .head-strategy (M  $\circ$  N)] @ U  $\ast\sim\ast$  ([M  $\circ$  N] @ [u]) @ U
    by (meson A B U 7 Resid-Arr-self cong-append ide-char

```

```

      prfx-transitive ⟨ $U \neq [] \implies \text{Arr } U$ ⟩
also have ( $[M \circ N] @ [u]$ ) @  $U = (M \circ N) \# u \# U$ 
  by simp
finally show stdz-insert ( $M \circ N$ ) ( $u \# U$ )  $\sim^*$  ( $M \circ N$ )  $\# u \# U$ 
  by blast
qed
qed
qed
next
assume  $\not\vdash \Lambda.\text{Ide } ((M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N))$ 
show ?thesis
proof (cases  $U = []$ )
  assume  $U: U = []$ 
  have  $5: \text{stdz-insert } (M \circ N) (u \# U) =$ 
     $\Lambda.\text{head-strategy } (M \circ N) \#$ 
     $\text{standard-development } ((M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N))$ 
  using  $1\ 2\ 3\ 4\ U\ * \ ** \ \langle \Lambda.\text{is-App } u \vee \Lambda.\text{is-Beta } u \rangle$ 
  apply (cases  $u$ )
  apply simp-all
  apply blast
  apply (cases  $M$ )
  apply simp-all
  by blast+
show ?thesis
proof (intro conjI)
  show Std (stdz-insert ( $M \circ N$ ) ( $u \# U$ ))
  proof –
  have  $\Lambda.\text{sseq } (\Lambda.\text{head-strategy } (M \circ N))$ 
    (hd (standard-development
      ( $(M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N)$ )))
  proof –
  have  $\Lambda.\text{seq } (\Lambda.\text{head-strategy } (M \circ N))$ 
    (hd (standard-development
      ( $(M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N)$ )))
  using  $MN\ **\ 4\ \Lambda.\text{Arr-head-strategy}\ \text{Arr-imp-arr-hd}$ 
     $\text{Ide-iff-standard-development-empty}\ \text{Src-hd-standard-development}$ 
     $\text{Std-imp-Arr}\ \text{Std-standard-development}\ \Lambda.\text{Arr-resid}$ 
     $\Lambda.\text{Src-head-strategy}\ \Lambda.\text{Src-resid}$ 
  by force
moreover have  $\Lambda.\text{elementary-reduction}$ 
    (hd (standard-development
      ( $(M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N)$ )))
  by (metis  $4\ \text{Ide-iff-standard-development-empty}\ MN\ \text{Std-consE}$ 
     $\text{Std-standard-development}\ \text{hd-Cons-tl}\ \Lambda.\text{Arr.simps}(4)$ 
     $\Lambda.\text{Arr-resid}\ \Lambda.\text{Con-head-strategy}$ 
     $\Lambda.\text{sseq-imp-elementary-reduction1}\ \text{Std.simps}(2)$ )
ultimately show ?thesis
using  $\Lambda.\text{sseq-head-reductionI}\ \text{Std-standard-development}$ 
by (metis  $**\ 2\ 3\ \text{Std}\ U\ \Lambda.\text{internal-reduction-preserves-no-head-redex}$ )

```

$\Lambda.is\text{-internal-reduction-iff}$   $\Lambda.Src\text{-head-strategy}$   
 $\Lambda.elementary\text{-reduction-not-ide}$   $\Lambda.head\text{-strategy-}Src$   
 $\Lambda.head\text{-strategy-is-elementary}$   $\Lambda.ide\text{-char}$   $\Lambda.is\text{-head-reduction-char}$   
 $\Lambda.is\text{-head-reduction-if}$   $\Lambda.seqE_\Lambda$   $Std.simps(2)$

**qed**

**thus** *?thesis*

**by** (*metis 4 5 MN Ide-iff-standard-development-empty*  
 $Std\text{-standard-development}$   $\Lambda.Arr.simps(4)$   $\Lambda.Arr\text{-resid}$   
 $\Lambda.Con\text{-head-strategy}$   $list.exhaust\text{-sel}$   $Std.simps(3)$ )

**qed**

**show**  $\neg Ide ((M \circ N) \# u \# U) \longrightarrow$   
 $stdz\text{-insert} (M \circ N) (u \# U) \text{ *~* } (M \circ N) \# u \# U$

**proof**

**have**  $stdz\text{-insert} (M \circ N) (u \# U) =$   
 $[\Lambda.head\text{-strategy} (M \circ N)] @$   
 $standard\text{-development} ((M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N))$

**using 5 by simp**

**also have**  $\dots \text{ *~* } [\Lambda.head\text{-strategy} (M \circ N)] @$   
 $[(M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N)]$

**proof** (*intro cong-append*)

**show 6:**  $seq [\Lambda.head\text{-strategy} (M \circ N)]$   
 $(standard\text{-development}$   
 $((M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N)))$

**using 4** *Ide-iff-standard-development-empty MN*  
 $\langle Std (stdz\text{-insert} (M \circ N) (u \# U)) \rangle$   
 $arr\text{-append-imp-seq}$   $arr\text{-char}$   $calculation$   $\Lambda.Arr\text{-head-strategy}$   
 $\Lambda.Arr\text{-resid}$   $lambda\text{-calculus.Src-head-strategy}$

**by force**

**show**  $[\Lambda.head\text{-strategy} (M \circ N)] \text{ *~* } [\Lambda.head\text{-strategy} (M \circ N)]$

**by** (*meson MN 6 cong-reflexive seqE*)

**show**  $standard\text{-development} ((M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N)) \text{ *~* }$   
 $[(M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N)]$

**using 4** *MN cong-standard-development*  $\Lambda.Arr.simps(4)$   
 $\Lambda.Arr\text{-resid}$   $\Lambda.Con\text{-head-strategy}$

**by presburger**

**qed**

**also have**  $[\Lambda.head\text{-strategy} (M \circ N)] @$   
 $[(M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N)] \text{ *~* }$   
 $[M \circ N] @ [u]$

**using B by blast**

**also have**  $[M \circ N] @ [u] = (M \circ N) \# u \# U$

**using U by simp**

**finally show**  $stdz\text{-insert} (M \circ N) (u \# U) \text{ *~* } (M \circ N) \# u \# U$   
**by blast**

**qed**

**qed**

**next**

**assume**  $U: U \neq []$

**have 5:**  $stdz\text{-insert} (M \circ N) (u \# U) =$

```

       $\Lambda$ .head-strategy (M  $\circ$  N) #
      stdz-insert ( $\Lambda$ .resid (M  $\circ$  N) ( $\Lambda$ .head-strategy (M  $\circ$  N))) U
using 1 2 3 4 U * **  $\langle \Lambda$ .is-App u  $\vee$   $\Lambda$ .is-Beta u  $\rangle$ 
apply (cases u)
      apply simp-all
      apply blast
apply (cases M)
      apply simp-all
by blast+
have 6: Std (stdz-insert ((M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N)) U)  $\wedge$ 
      set (stdz-insert ((M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N)) U)
       $\subseteq$  {a.  $\Lambda$ .elementary-reduction a}  $\wedge$ 
      stdz-insert ((M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N)) U  $\sim^*$ 
      (M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N) # U
proof -
      have seq [(M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N)] U
      proof
        show Arr [(M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N)]
          by (simp add: MN  $\Lambda$ .Arr-resid  $\Lambda$ .Con-head-strategy)
        show Arr U
          using U  $\langle U \neq [] \implies$  Arr U  $\rangle$  by blast
        show  $\Lambda$ .Trg (last [(M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N)]) =  $\Lambda$ .Src (hd U)
          by (metis (mono-tags, lifting) B U Std Std-consE Trg-last-eqI
             $\Lambda$ .seq-char  $\Lambda$ .sseq-imp-seq last-ConsL last-snoc)
      qed
thus ?thesis
      using ind5 Std-implies-set-subset-elementary-reduction
      by (metis ** 1 2 3 4 Std Std.simps(3) Arr-iff-Con-self Ide.simps(3)
        Resid.simps(1) seq-char  $\Lambda$ .ide-char list.exhaust-sel list.sel(1,3))
qed
show ?thesis
proof (intro conjI)
      show Std (stdz-insert (M  $\circ$  N) (u # U))
      proof -
        have  $\Lambda$ .sseq ( $\Lambda$ .head-strategy (M  $\circ$  N))
          (hd (stdz-insert ((M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N)) U))
      proof -
        have  $\Lambda$ .seq ( $\Lambda$ .head-strategy (M  $\circ$  N))
          (hd (stdz-insert ((M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N)) U))
      proof
        show  $\Lambda$ .Arr ( $\Lambda$ .head-strategy (M  $\circ$  N))
          using MN  $\Lambda$ .Arr-head-strategy by force
        show  $\Lambda$ .Arr (hd (stdz-insert ((M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N)) U))
          using 6
          by (metis Ide.simps(1) Resid.simps(2) Std-consE hd-Cons-tl ide-char)
        show  $\Lambda$ .Trg ( $\Lambda$ .head-strategy (M  $\circ$  N)) =
           $\Lambda$ .Src (hd (stdz-insert ((M  $\circ$  N)  $\setminus$   $\Lambda$ .head-strategy (M  $\circ$  N)) U))
          using 6
          by (metis MN Src-hd-eqI  $\Lambda$ .Arr.simps(4)  $\Lambda$ .Con-head-strategy)

```

$\Lambda.\text{Src-resid list.sel}(1)$

**qed**

**moreover have**  $\Lambda.\text{is-head-reduction } (\Lambda.\text{head-strategy } (M \circ N))$

**using**  $** 1 2 3 \Lambda.\text{Src-head-strategy } \Lambda.\text{head-strategy-is-elementary}$   
 $\Lambda.\text{head-strategy-Src } \Lambda.\text{is-head-reduction-char } \Lambda.\text{seq-char}$

**by**  $(\text{metis } \Lambda.\text{Src-head-redex } \Lambda.\text{contains-head-reduction-iff}$   
 $\Lambda.\text{head-redex-is-head-reduction}$   
 $\Lambda.\text{internal-reduction-preserves-no-head-redex}$   
 $\Lambda.\text{is-internal-reduction-iff})$

**moreover have**  $\Lambda.\text{elementary-reduction}$   
 $(\text{hd } (\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N)) U))$

**by**  $(\text{metis } 6 \text{ Ide.simps}(1) \text{ Resid.simps}(2) \text{ ide-char hd-in-set}$   
 $\text{in-mono mem-Collect-eq})$

**ultimately show**  $?thesis$

**using**  $\Lambda.\text{sseq-head-reductionI}$  **by**  $\text{blast}$

**qed**

**thus**  $?thesis$

**by**  $(\text{metis } 5 6 \text{ Std.simps}(3) \text{ Arr.simps}(1) \text{ Con-implies-Arr}(1)$   
 $\text{con-char prfx-implies-con list.exhaust-sel})$

**qed**

**show**  $\neg \text{Ide } ((M \circ N) \# u \# U) \longrightarrow$   
 $\text{stdz-insert } (M \circ N) (u \# U) * \sim * (M \circ N) \# u \# U$

**proof**

**have**  $\text{stdz-insert } (M \circ N) (u \# U) =$   
 $[\Lambda.\text{head-strategy } (M \circ N)] @$   
 $\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N)) U$

**using**  $5$  **by**  $\text{simp}$

**also have**  $10: \dots * \sim * [\Lambda.\text{head-strategy } (M \circ N)] @$   
 $((M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N) \# U)$

**proof**  $(\text{intro cong-append})$

**show**  $10: \text{seq } [\Lambda.\text{head-strategy } (M \circ N)]$   
 $(\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N)) U)$

**by**  $(\text{metis } 5 6 \text{ Ide.simps}(1) \text{ Resid.simps}(1) \text{ Std-imp-Arr}$   
 $\langle \text{Std } (\text{stdz-insert } (M \circ N) (u \# U)) \rangle \text{arr-append-imp-seq}$   
 $\text{arr-char calculation ide-char list.distinct}(1))$

**show**  $[\Lambda.\text{head-strategy } (M \circ N)] * \sim * [\Lambda.\text{head-strategy } (M \circ N)]$

**using**  $MN 10 \text{ cong-reflexive}$  **by**  $\text{blast}$

**show**  $\text{stdz-insert } ((M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N)) U * \sim *$   
 $(M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N) \# U$

**using**  $6$  **by**  $\text{auto}$

**qed**

**also have**  $11: [\Lambda.\text{head-strategy } (M \circ N)] @$   
 $((M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N) \# U) =$   
 $([\Lambda.\text{head-strategy } (M \circ N)] @$   
 $[(M \circ N) \setminus \Lambda.\text{head-strategy } (M \circ N)]) @ U$

**by**  $\text{simp}$

**also have**  $\dots * \sim * (([M \circ N] @ [u]) @ U)$

**proof**  $-$

**have**  $\text{seq } ([\Lambda.\text{head-strategy } (M \circ N)] @$

```

      [(M ◦ N) \ Λ.head-strategy (M ◦ N)] U
    by (metis U 10 11 append-is-Nil-conv arr-append-imp-seq
        cong-implies-coterminal coterminalE not-Cons-self2)
    thus ?thesis
      using B cong-append cong-reflexive by blast
    qed
    also have (([M ◦ N] @ [u]) @ U = (M ◦ N) # u # U)
      by simp
    finally show stdz-insert (M ◦ N) (u # U) *~* (M ◦ N) # u # U
      by blast
    qed
  qed
  qed
  next
  assume 3: ¬ Λ.contains-head-reduction u
  have u: Λ.Arr u ∧ Λ.is-App u ∧ ¬ Λ.contains-head-reduction u
    using 3 ‹Λ.is-App u ∨ Λ.is-Beta u› Λ.is-Beta-def u by force
  have 5: ¬ Λ.Ide u
    by (metis Std Std.simps(2) Std.simps(3) Λ.elementary-reduction-not-ide
        Λ.ide-char neq-Nil-conv Λ.sseq-imp-elementary-reduction1)
  show ?thesis
  proof –
    have 4: stdz-insert (M ◦ N) (u # U) =
      map (λX. Λ.App X (Λ.Src N))
        (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) @
        map (Λ.App (Λ.Trq (Λ.un-App1 (last (u # U))))
          (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))
      using u MN 1 2 3 5 * ** ‹Λ.is-App u ∨ Λ.is-Beta u›
    apply (cases u)
      apply simp-all
    apply (cases U = [])
      apply simp-all
    by blast+
    have **: set U ⊆ Collect Λ.is-App
      using u 5 Std seq-App-Std-implies by blast
    have X: Std (filter notIde (map Λ.un-App1 (u # U)))
      by (metis ** Std Std-filter-map-un-App1 insert-subset list.simps(15)
          mem-Collect-eq u)
    have Y: Std (filter notIde (map Λ.un-App2 (u # U)))
      by (metis ** u Std Std-filter-map-un-App2 insert-subset list.simps(15)
          mem-Collect-eq)
    have A: ¬ Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide ⇒
      Std (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) ∧
      set (stdz-insert M (filter notIde (map Λ.un-App1 (u # U))))
        ⊆ {a. Λ.elementary-reduction a} ∧
      stdz-insert M (filter notIde (map Λ.un-App1 (u # U))) *~*
      M # filter notIde (map Λ.un-App1 (u # U))
  proof –

```

```

assume *:  $\neg \Lambda.un-App1 \text{ ' set } (u \# U) \subseteq Collect \Lambda.Ide$ 
have seq [M] (filter notIde (map  $\Lambda.un-App1$  (u # U)))
proof
  show Arr [M]
    using MN by simp
  show Arr (filter notIde (map  $\Lambda.un-App1$  (u # U)))
    by (metis (mono-tags, lifting) * Std-imp-Arr X empty-filter-conv
      list.set-map mem-Collect-eq subset-code(1))
  show  $\Lambda.Trq$  (last [M]) =  $\Lambda.Src$  (hd (filter notIde (map  $\Lambda.un-App1$  (u # U))))
  proof –
    have  $\Lambda.Trq$  (last [M]) =  $\Lambda.Src$  (hd (map  $\Lambda.un-App1$  (u # U)))
      using ** u by fastforce
    also have ... =  $\Lambda.Src$  (hd (filter notIde (map  $\Lambda.un-App1$  (u # U))))
    proof –
      have Arr (map  $\Lambda.un-App1$  (u # U))
        using u ***
      by (metis Arr-map-un-App1 Std Std-imp-Arr insert-subset
        list.simps(15) mem-Collect-eq neq-Nil-conv)
    moreover have  $\neg Ide$  (map  $\Lambda.un-App1$  (u # U))
      by (metis * Collect-cong  $\Lambda.ide-char$  list.set-map set-Ide-subset-ide)
    ultimately show ?thesis
      using Src-hd-eqI cong-filter-notIde by blast
    qed
  finally show ?thesis by blast
  qed
moreover have  $\neg Ide$  (M # filter notIde (map  $\Lambda.un-App1$  (u # U)))
  using *
  by (metis (no-types, lifting) *** Arr-map-un-App1 Std Std-imp-Arr
    Arr.simps(1) Ide.elims(2) Resid-Arr-Ide-ind ide-char
    seq-char calculation(1) cong-filter-notIde filter-notIde-Ide
    insert-subset list.discI list.sel(3) list.simps(15) mem-Collect-eq u)
  ultimately show ?thesis
  by (metis X 1 2 3 ** ind7 Std-implies-set-subset-elementary-reduction
    list.sel(1))
qed
have B:  $\neg \Lambda.un-App2 \text{ ' set } (u \# U) \subseteq Collect \Lambda.Ide \implies$ 
  Std (stdz-insert N (filter notIde (map  $\Lambda.un-App2$  (u # U))))  $\wedge$ 
  set (stdz-insert N (filter notIde (map  $\Lambda.un-App2$  (u # U))))
   $\subseteq \{a. \Lambda.elementary-reduction a\} \wedge$ 
  stdz-insert N (filter notIde (map  $\Lambda.un-App2$  (u # U)))  $\sim^*$ 
  N # filter notIde (map  $\Lambda.un-App2$  (u # U))
proof –
  assume **:  $\neg \Lambda.un-App2 \text{ ' set } (u \# U) \subseteq Collect \Lambda.Ide$ 
  have seq [N] (filter notIde (map  $\Lambda.un-App2$  (u # U)))
  proof
    show Arr [N]
      using MN by simp
    show Arr (filter ( $\lambda u. \neg \Lambda.Ide$  u) (map  $\Lambda.un-App2$  (u # U)))

```

```

    by (metis (mono-tags, lifting) ** Std-imp-Arr Y empty-filter-conv
        list.set-map mem-Collect-eq subset-code(1))
  show  $\Lambda.Trg$  (last [N]) =  $\Lambda.Src$  (hd (filter notIde (map  $\Lambda.un-App2$  (u # U))))
  proof -
    have  $\Lambda.Trg$  (last [N]) =  $\Lambda.Src$  (hd (map  $\Lambda.un-App2$  (u # U)))
    by (metis u seq Trg-last-Src-hd-eqI  $\Lambda.Src.simps(4)$ 
         $\Lambda.Trg.simps(3)$   $\Lambda.is-App-def$   $\Lambda.lambda.sel(4)$  last-ConsL
        list.discI list.map-sel(1) list.sel(1))
    also have ... =  $\Lambda.Src$  (hd (filter notIde (map  $\Lambda.un-App2$  (u # U))))
    proof -
      have Arr (map  $\Lambda.un-App2$  (u # U))
      using u ***
      by (metis Arr-map-un-App2 Std Std-imp-Arr list.distinct(1)
          mem-Collect-eq set-ConsD subset-code(1))
      moreover have  $\neg$  Ide (map  $\Lambda.un-App2$  (u # U))
      by (metis ** Collect-cong  $\Lambda.ide-char$  list.set-map set-Ide-subset-ide)
      ultimately show ?thesis
      using Src-hd-eqI cong-filter-notIde by blast
    qed
    finally show ?thesis by blast
  qed
  moreover have  $\Lambda.seq$  (M  $\circ$  N) u
  by (metis u Srcs-simp $\Lambda_P$  Arr.simps(2) Trgs.simps(2) seq-char
      list.sel(1) seq  $\Lambda.seqI(1)$   $\Lambda.sources-char_\Lambda$ )
  moreover have  $\neg$  Ide (N # filter notIde (map  $\Lambda.un-App2$  (u # U)))
  using u *
  by (metis (no-types, lifting) *** Arr-map-un-App2 Std Std-imp-Arr
      Arr.simps(1) Ide.elims(2) Resid-Arr-Ide-ind ide-char
      seq-char calculation(1) cong-filter-notIde filter-notIde-Ide
      insert-subset list.discI list.sel(3) list.simps(15) mem-Collect-eq)
  ultimately show ?thesis
  using * 1 2 3 Y ind8 Std-implies-set-subset-elementary-reduction
  by simp
  qed
  show ?thesis
  proof (cases  $\Lambda.un-App1$  ' set (u # U)  $\subseteq$  Collect  $\Lambda.Ide$ ;
      cases  $\Lambda.un-App2$  ' set (u # U)  $\subseteq$  Collect  $\Lambda.Ide$ )
  show  $\llbracket \Lambda.un-App1$  ' set (u # U)  $\subseteq$  Collect  $\Lambda.Ide$ ;
       $\Lambda.un-App2$  ' set (u # U)  $\subseteq$  Collect  $\Lambda.Ide$   $\rrbracket$ 
       $\implies$  ?thesis
  proof -
    assume *:  $\Lambda.un-App1$  ' set (u # U)  $\subseteq$  Collect  $\Lambda.Ide$ 
    assume **:  $\Lambda.un-App2$  ' set (u # U)  $\subseteq$  Collect  $\Lambda.Ide$ 
    have False
    using u 5 * ** Ide-iff-standard-development-empty
    by (metis  $\Lambda.Ide.simps(4)$  image-subset-iff  $\Lambda.lambda.collapse(3)$ 
        list.set-intros(1) mem-Collect-eq)
  thus ?thesis by blast

```

```

qed
show  $\llbracket \Lambda.un\text{-}App1 \text{ ' set } (u \# U) \subseteq Collect \ \Lambda.Ide;$ 
       $\neg \Lambda.un\text{-}App2 \text{ ' set } (u \# U) \subseteq Collect \ \Lambda.Ide \rrbracket$ 
       $\implies ?thesis$ 
proof –
  assume *:  $\Lambda.un\text{-}App1 \text{ ' set } (u \# U) \subseteq Collect \ \Lambda.Ide$ 
  assume **:  $\neg \Lambda.un\text{-}App2 \text{ ' set } (u \# U) \subseteq Collect \ \Lambda.Ide$ 
  have 6:  $\Lambda.Trq (\Lambda.un\text{-}App1 (last (u \# U))) = \Lambda.Trq M$ 
  proof –
    have  $\Lambda.Trq M = \Lambda.Src (hd (map \ \Lambda.un\text{-}App1 (u \# U)))$ 
      by (metis u seq Trg-last-Src-hd-eqI hd-map \ \Lambda.Src.simps(4) \ \Lambda.Trq.simps(3)
         $\Lambda.is\text{-}App\text{-}def \ \Lambda.lambda.sel(3) \ last\text{-}ConsL \ list.discI \ list.sel(1)$ )
    also have ... =  $\Lambda.Trq (last (map \ \Lambda.un\text{-}App1 (u \# U)))$ 
    proof –
      have 6:  $Ide (map \ \Lambda.un\text{-}App1 (u \# U))$ 
      using * *** u Std Std-imp-Arr Ide-char ide-char Arr-map-un-App1
      by (metis (mono-tags, lifting) Collect-cong insert-subset
         $\Lambda.ide\text{-}char \ list.distinct(1) \ list.set\text{-}map \ list.simps(15)$ 
        mem-Collect-eq)
      hence  $Src (map \ \Lambda.un\text{-}App1 (u \# U)) = Trq (map \ \Lambda.un\text{-}App1 (u \# U))$ 
      using Ide-imp-Src-eq-Trq by blast
      thus ?thesis
      using 6 Ide-implies-Arr by force
    qed
    also have ... =  $\Lambda.Trq (\Lambda.un\text{-}App1 (last (u \# U)))$ 
      by (simp add: last-map)
    finally show ?thesis by simp
  qed
have filter notIde ( $map \ \Lambda.un\text{-}App1 (u \# U)$ ) = []
  using * by (simp add: subset-eq)
hence 4: stdz-insert ( $M \circ N$ ) ( $u \# U$ ) =
   $map (\lambda X. X \circ \Lambda.Src \ N) (standard\text{-}development \ M) \ @$ 
   $map (\Lambda.App (\Lambda.Trq (\Lambda.un\text{-}App1 (last (u \# U))))$ 
   $(stdz\text{-}insert \ N (filter \ notIde (map \ \Lambda.un\text{-}App2 (u \# U))))$ 
  using u 4 5 * ** Ide-iff-standard-development-empty MN
  by simp
show ?thesis
proof (intro conjI)
  have Std ( $map (\lambda X. X \circ \Lambda.Src \ N) (standard\text{-}development \ M) \ @$ 
     $map (\Lambda.App (\Lambda.Trq (\Lambda.un\text{-}App1 (last (u \# U))))$ 
     $(stdz\text{-}insert \ N (filter \ notIde (map \ \Lambda.un\text{-}App2 (u \# U))))$ )
  proof (intro Std-append)
    show Std ( $map (\lambda X. X \circ \Lambda.Src \ N) (standard\text{-}development \ M)$ )
      using Std-map-App1 Std-standard-development MN \ \Lambda.Ide-Src
      by force
    show Std ( $map (\Lambda.App (\Lambda.Trq (\Lambda.un\text{-}App1 (last (u \# U))))$ 
       $(stdz\text{-}insert \ N (filter \ notIde (map \ \Lambda.un\text{-}App2 (u \# U))))$ )
      using ** B MN 6 Std-map-App2 \ \Lambda.Ide-Trq by presburger
    show  $map (\lambda X. X \circ \Lambda.Src \ N) (standard\text{-}development \ M) = [] \ \vee$ 

```

$$\begin{aligned} & \text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } (u \# U)))))) \\ & \quad (\text{stdz-insert } N \text{ (filter notIde (map } \Lambda.\text{un-App2 } (u \# U)))) = [] \vee \\ & \Lambda.\text{sseq } (\text{last } (\text{map } (\lambda X. X \circ \Lambda.\text{Src } N) (\text{standard-development } M))) \\ & \quad (\text{hd } (\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } (u \# U)))))) \\ & \quad \quad (\text{stdz-insert } N \text{ (filter notIde} \\ & \quad \quad \quad (\text{map } \Lambda.\text{un-App2 } (u \# U))))))
\end{aligned}$$

**proof** (*cases*  $\Lambda.\text{Ide } M$ )

**show**  $\Lambda.\text{Ide } M \implies ?thesis$

**using** *Ide-iff-standard-development-empty MN by blast*

**assume**  $M: \neg \Lambda.\text{Ide } M$

**have**  $\Lambda.\text{sseq } (\text{last } (\text{map } (\lambda X. X \circ \Lambda.\text{Src } N) (\text{standard-development } M)))$   
 $(\text{hd } (\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } (u \# U))))))$   
 $(\text{stdz-insert } N \text{ (filter notIde}$   
 $(\text{map } \Lambda.\text{un-App2 } (u \# U))))))$

**proof** –

**have**  $\text{last } (\text{map } (\lambda X. X \circ \Lambda.\text{Src } N) (\text{standard-development } M)) =$   
 $\Lambda.\text{App } (\text{last } (\text{standard-development } M)) (\Lambda.\text{Src } N)$

**using**  $M$

**by** (*simp add: Ide-iff-standard-development-empty MN last-map*)

**moreover have**  $\text{hd } (\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } (u \# U))))))$   
 $(\text{stdz-insert } N \text{ (filter notIde}$   
 $(\text{map } \Lambda.\text{un-App2 } (u \# U)))) =$   
 $\Lambda.\text{App } (\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } (u \# U))))$   
 $(\text{hd } (\text{stdz-insert } N \text{ (filter notIde}$   
 $(\text{map } \Lambda.\text{un-App2 } (u \# U))))))$

**by** (*metis \*\* B Ide.simps(1) Resid.simps(2) hd-map ide-char*)

**moreover**

**have**  $\Lambda.\text{sseq } (\Lambda.\text{App } (\text{last } (\text{standard-development } M)) (\Lambda.\text{Src } N))$

...

**proof** –

**have**  $\Lambda.\text{elementary-reduction } (\text{last } (\text{standard-development } M))$

**using**  $M \text{ MN Std-standard-development}$

*Ide-iff-standard-development-empty last-in-set*

*mem-Collect-eq set-standard-development subsetD*

**by** *metis*

**moreover have**  $\Lambda.\text{elementary-reduction}$

$(\text{hd } (\text{stdz-insert } N$

$(\text{filter notIde } (\text{map } \Lambda.\text{un-App2 } (u \# U))))))$

**using**  $** B$

**by** (*metis Arr.simps(1) Con-implies-Arr(2) Ide.simps(1)*

*ide-char in-mono list.set-sel(1) mem-Collect-eq*)

**moreover have**  $\Lambda.\text{Trg } (\text{last } (\text{standard-development } M)) =$

$\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } (u \# U)))$

**using**  $M \text{ MN 6 Trg-last-standard-development by presburger}$

**moreover have**  $\Lambda.\text{Src } N =$

$\Lambda.\text{Src } (\text{hd } (\text{stdz-insert } N$

$(\text{filter notIde } (\text{map } \Lambda.\text{un-App2 } (u \# U))))))$

**by** (*metis \*\* B Src-hd-eqI list.sel(1)*)

**ultimately show** *?thesis*

```

    by simp
  qed
  ultimately show ?thesis by simp
  qed
  thus ?thesis by blast
  qed
  qed
  thus Std (stdz-insert (M ∘ N) (u # U))
    using 4 by simp
  show ¬ Ide ((M ∘ N) # u # U) →
    stdz-insert (M ∘ N) (u # U) *~* (M ∘ N) # u # U
  proof
  show stdz-insert (M ∘ N) (u # U) *~* (M ∘ N) # u # U
  proof (cases Λ.Ide M)
  assume M: Λ.Ide M
  have stdz-insert (M ∘ N) (u # U) =
    map (Λ.App (Λ.Trq (Λ.un-App1 (last (u # U)))))
      (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))
  using 4 M MN Ide-iff-standard-development-empty by simp
  also have ... *~* (map (Λ.App (Λ.Trq (Λ.un-App1 (last (u # U)))))
    (N # filter notIde (map Λ.un-App2 (u # U))))
  proof -
  have Λ.Ide (Λ.Trq (Λ.un-App1 (last (u # U))))
    using M 6 Λ.Ide-Trq Λ.Ide-implies-Arr by fastforce
  thus ?thesis
    using ** *** B u cong-map-App1 by blast
  qed
  also have map (Λ.App (Λ.Trq (Λ.un-App1 (last (u # U)))))
    (N # filter notIde (map Λ.un-App2 (u # U))) =
    map (Λ.App (Λ.Trq (Λ.un-App1 (last (u # U)))))
      (filter notIde (N # map Λ.un-App2 (u # U)))
  using 1 M by force
  also have map (Λ.App (Λ.Trq (Λ.un-App1 (last (u # U)))))
    (filter notIde (N # map Λ.un-App2 (u # U))) *~*
    map (Λ.App (Λ.Trq (Λ.un-App1 (last (u # U)))))
      (N # map Λ.un-App2 (u # U))
  proof -
  have Arr (N # map Λ.un-App2 (u # U))
  proof
  show Λ.arr N
    using MN by blast
  show Arr (map Λ.un-App2 (u # U))
    using *** u Std Arr-map-un-App2
    by (metis Std-imp-Arr insert-subset list.distinct(1)
      list.simps(15) mem-Collect-eq)
  show Λ.trq N = Src (map Λ.un-App2 (u # U))
    using u ⟨Λ.seq (M ∘ N) u⟩ Λ.seq-char Λ.is-App-def by auto
  qed
  moreover have ¬ Ide (N # map Λ.un-App2 (u # U))

```

```

    using 1 M by force
  moreover have  $\Lambda.Ide (\Lambda.Trig (\Lambda.un-App1 (last (u \# U))))$ 
    using M 6  $\Lambda.Ide-Trig \Lambda.Ide-implies-Arr$  by presburger
  ultimately show ?thesis
    using cong-filter-notIde cong-map-App1 by blast
qed
also have map ( $\Lambda.App (\Lambda.Trig (\Lambda.un-App1 (last (u \# U))))$ )
  ( $N \# map \Lambda.un-App2 (u \# U)$ ) =
  map ( $\Lambda.App M$ ) ( $N \# map \Lambda.un-App2 (u \# U)$ )
  using M MN  $\langle \Lambda.Trig (\Lambda.un-App1 (last (u \# U))) = \Lambda.Trig M \rangle$ 
   $\Lambda.Ide-iff-Trig-self$ 
  by force
also have ... = ( $M \circ N$ )  $\# map (\Lambda.App M) (map \Lambda.un-App2 (u \# U))$ 
  by simp
also have ... = ( $M \circ N$ )  $\# u \# U$ 
proof -
  have Arr ( $u \# U$ )
    using Std Std-imp-Arr by blast
  moreover have set ( $u \# U$ )  $\subseteq Collect \Lambda.is-App$ 
    using *** u by simp
  moreover have  $\Lambda.un-App1 u = M$ 
  by (metis * u M seq Trig-last-Src-hd-eqI  $\Lambda.Ide-iff-Src-self$ 
     $\Lambda.Ide-iff-Trig-self \Lambda.Ide-implies-Arr \Lambda.Src.simps(4)$ 
     $\Lambda.Trig.simps(3) \Lambda.lambda.collapse(3) \Lambda.lambda.sel(3)$ 
    last.simps list.distinct(1) list.sel(1) list.set-intros(1)
    list.set-map list.simps(9) mem-Collect-eq standardize.cases
    subset-iff)
  moreover have  $\Lambda.un-App1 \text{ ` } set (u \# U) \subseteq \{M\}$ 
proof -
  have Ide ( $map \Lambda.un-App1 (u \# U)$ )
    using * *** Std Std-imp-Arr Arr-map-un-App1
  by (metis Collect-cong Ide-char calculation(1-2)  $\Lambda.ide-char$ 
    list.set-map)
  thus ?thesis
    by (metis calculation(3) hd-map list.discI list.sel(1)
    list.set-map set-Ide-subset-single-hd)
qed
ultimately show ?thesis
  using M map-App-map-un-App2 by blast
qed
finally show ?thesis by blast
next
assume M:  $\neg \Lambda.Ide M$ 
have stdz-insert ( $M \circ N$ ) ( $u \# U$ ) =
  map ( $\lambda X. X \circ \Lambda.Src N$ ) (standard-development M) @
  map ( $\lambda X. \Lambda.Trig M \circ X$ )
  (stdz-insert N (filter notIde (map  $\Lambda.un-App2 (u \# U)$ )))
  using 4 6 by simp
also have ...  $\sim^* [M \circ \Lambda.Src N] @ [\Lambda.Trig M \circ N] @$ 

```

```

      map (λX. Λ.Trg M ◦ X)
      (filter notIde (map Λ.un-App2 (u # U)))
proof (intro cong-append)
show map (λX. X ◦ Λ.Src N) (standard-development M) *~*
  [M ◦ Λ.Src N]
  using MN M cong-standard-development Λ.Ide-Src
    cong-map-App2 [of Λ.Src N standard-development M [M]]
  by simp
show map (λX. Λ.Trg M ◦ X)
  (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))) *~*
  [Λ.Trg M ◦ N] @
  map (λX. Λ.Trg M ◦ X)
  (filter notIde (map Λ.un-App2 (u # U)))
proof -
  have map (λX. Λ.Trg M ◦ X)
  (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))) *~*
  map (λX. Λ.Trg M ◦ X)
  (N # filter notIde (map Λ.un-App2 (u # U)))
  using ** B MN cong-map-App1 lambda-calculus.Ide-Trg
  by presburger
  also have map (λX. Λ.Trg M ◦ X)
  (N # filter notIde (map Λ.un-App2 (u # U))) =
  [Λ.Trg M ◦ N] @
  map (λX. Λ.Trg M ◦ X)
  (filter notIde (map Λ.un-App2 (u # U)))

  by simp
  finally show ?thesis by blast
qed
show seq (map (λX. X ◦ Λ.Src N) (standard-development M))
  (map (λX. Λ.Trg M ◦ X)
  (stdz-insert N (filter notIde
    (map Λ.un-App2 (u # U)))))
  using MN M ** B cong-standard-development [of M]
  by (metis Nil-is-append-conv Resid.simps(2) Std-imp-Arr
  ‹Std (stdz-insert (M ◦ N) (u # U))› arr-append-imp-seq
  arr-char calculation complete-development-Ide-iff
  complete-development-def list.map-disc-iff development.simps(1))
qed
also have [M ◦ Λ.Src N] @ [Λ.Trg M ◦ N] @
  map (λX. Λ.Trg M ◦ X)
  (filter notIde (map Λ.un-App2 (u # U))) =
  ([M ◦ Λ.Src N] @ [Λ.Trg M ◦ N]) @
  map (λX. Λ.Trg M ◦ X)
  (filter notIde (map Λ.un-App2 (u # U)))

  by simp
also have ([M ◦ Λ.Src N] @ [Λ.Trg M ◦ N]) @
  map (λX. Λ.Trg M ◦ X)
  (filter notIde (map Λ.un-App2 (u # U))) *~*
  ([M ◦ Λ.Src N] @ [Λ.Trg M ◦ N]) @

```

```

      map (λX. Λ.Trg M ∘ X) (map Λ.un-App2 (u # U))
proof (intro cong-append)
show seq ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])
      (map (λX. Λ.Trg M ∘ X)
        (filter notIde (map Λ.un-App2 (u # U))))
proof
show Arr ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])
      by (simp add: MN)
show 9: Arr (map (λX. Λ.Trg M ∘ X)
      (filter notIde (map Λ.un-App2 (u # U))))
proof –
      have Arr (map Λ.un-App2 (u # U))
        using *** u Arr-map-un-App2
      by (metis Std Std-imp-Arr list.distinct(1) mem-Collect-eq
        set-ConsD subset-code(1))
moreover have ¬ Ide (map Λ.un-App2 (u # U))
      using **
      by (metis Collect-cong Λ.ide-char list.set-map
        set-Ide-subset-ide)
ultimately show ?thesis
      using cong-filter-notIde
      by (metis Arr-map-App2 Con-implies-Arr(2) Ide.simps(1)
        MN ide-char Λ.Ide-Trg)
qed
show Λ.Trg (last ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])) =
      Λ.Src (hd (map (λX. Λ.Trg M ∘ X)
      (filter notIde (map Λ.un-App2 (u # U)))))
proof –
      have Λ.Trg (last ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])) =
        Λ.Trg M ∘ Λ.Trg N
      using MN by auto
also have ... = Λ.Src u
      using Trg-last-Src-hd-eqI seq by force
also have ... = Λ.Src (Λ.Trg M ∘ Λ.un-App2 u)
      using MN ⟨Λ.App (Λ.Trg M) (Λ.Trg N) = Λ.Src u⟩ u by auto
also have 8: ... = Λ.Trg M ∘ Λ.Src (Λ.un-App2 u)
      using MN by simp
also have 7: ... = Λ.Trg M ∘
      Λ.Src (hd (filter notIde
      (map Λ.un-App2 (u # U))))
      using u 5 list.simps(9) cong-filter-notIde
      ⟨filter notIde (map Λ.un-App1 (u # U)) = []⟩
      by auto
also have ... = Λ.Src (hd (map (λX. Λ.Trg M ∘ X)
      (filter notIde
      (map Λ.un-App2 (u # U)))))
      by (metis 7 8 9 Arr.simps(1) hd-map Λ.Src.simps(4)
        Λ.lambda.sel(4) list.simps(8))

```

```

finally show  $\Lambda.Trg$  ( $last$  ( $[M \circ \Lambda.Src\ N] @ [\Lambda.Trg\ M \circ N]$ )) =
 $\Lambda.Src$  ( $hd$  ( $map$  ( $\lambda X. \Lambda.Trg\ M \circ X$ )
( $filter\ notIde$ 
( $map\ \Lambda.un-App2$  ( $u \# U$ ))))))
  by blast
  qed
qed
show  $seq$  [ $M \circ \Lambda.Src\ N$ ] [ $\Lambda.Trg\ M \circ N$ ]
  using  $MN$  by force
show [ $M \circ \Lambda.Src\ N$ ]  $\sim^*$  [ $M \circ \Lambda.Src\ N$ ]
  using  $MN$ 
  by ( $meson$  head-redex-decomp  $\Lambda.Arr.simps(4)$   $\Lambda.Arr.Src$ 
prfx-transitive)
show [ $\Lambda.Trg\ M \circ N$ ]  $\sim^*$  [ $\Lambda.Trg\ M \circ N$ ]
  using  $MN$ 
  by ( $meson$   $\langle seq$  [ $M \circ \Lambda.Src\ N$ ] [ $\Lambda.Trg\ M \circ N$ ]  $\rangle$  cong-reflexive seqE)
show  $map$  ( $\lambda X. \Lambda.Trg\ M \circ X$ )
  ( $filter\ notIde$  ( $map\ \Lambda.un-App2$  ( $u \# U$ )))  $\sim^*$ 
   $map$  ( $\lambda X. \Lambda.Trg\ M \circ X$ ) ( $map\ \Lambda.un-App2$  ( $u \# U$ ))
proof –
  have  $Arr$  ( $map\ \Lambda.un-App2$  ( $u \# U$ ))
  using  $***$   $u$   $Arr-map-un-App2$ 
  by ( $metis$   $Std\ Std-imp-Arr\ list.distinct(1)$   $mem-Collect-eq$ 
set-ConsD subset-code(1))
  moreover have  $\neg Ide$  ( $map\ \Lambda.un-App2$  ( $u \# U$ ))
  using  $**$ 
  by ( $metis$   $Collect-cong\ \Lambda.ide-char\ list.set-map$ 
set-Ide-subset-ide)
  ultimately show ?thesis
  using  $M\ MN\ cong-filter-notIde\ cong-map-App1\ \Lambda.Ide-Trg$ 
  by presburger
qed
qed
also have ( $[M \circ \Lambda.Src\ N] @ [\Lambda.Trg\ M \circ N]$ ) @
   $map$  ( $\lambda X. \Lambda.Trg\ M \circ X$ ) ( $map\ \Lambda.un-App2$  ( $u \# U$ ))  $\sim^*$ 
  [ $M \circ N$ ] @  $u \# U$ 
proof (intro cong-append)
  show  $seq$  ( $[M \circ \Lambda.Src\ N] @ [\Lambda.Trg\ M \circ N]$ )
  ( $map$  ( $\lambda X. \Lambda.Trg\ M \circ X$ ) ( $map\ \Lambda.un-App2$  ( $u \# U$ )))
  by ( $metis$   $Nil-is-append-conv\ Nil-is-map-conv\ arr-append-imp-seq$ 
calculation cong-implies-coterminal coterminalE
list.distinct(1))
  show [ $M \circ \Lambda.Src\ N$ ] @ [ $\Lambda.Trg\ M \circ N$ ]  $\sim^*$  [ $M \circ N$ ]
  using  $MN\ \Lambda.resid-Arr-self\ \Lambda.Arr-not-Nil\ \Lambda.Ide-Trg\ ide-char$  by simp
show  $map$  ( $\lambda X. \Lambda.Trg\ M \circ X$ ) ( $map\ \Lambda.un-App2$  ( $u \# U$ ))  $\sim^*$   $u \# U$ 
proof –
  have  $map$  ( $\lambda X. \Lambda.Trg\ M \circ X$ ) ( $map\ \Lambda.un-App2$  ( $u \# U$ )) =  $u \# U$ 
  proof (intro map-App-map-un-App2)
  show  $Arr$  ( $u \# U$ )

```

```

    using Std Std-imp-Arr by blast
  show set (u # U) ⊆ Collect Λ.is-App
    using *** u by auto
  show Λ.Ide (Λ.Trig M)
    using MN Λ.Ide-Trig by blast
  show Λ.un-App1 ‘ set (u # U) ⊆ {Λ.Trig M}
  proof –
    have Λ.un-App1 u = Λ.Trig M
      using * u seq seq-char
    apply (cases u)
      apply simp-all
    by (metis Trig-last-Src-hd-eqI Λ.Ide-iff-Src-self
      Λ.Src-Src Λ.Src-Trig Λ.Src-eq-iff(2) Λ.Trig.simps(3)
      last-ConsL list.sel(1) seq u)
  moreover have Ide (map Λ.un-App1 (u # U))
    using * Std Std-imp-Arr Arr-map-un-App1
  by (metis Collect-cong Ide-char
    ⟨Arr (u # U)⟩ ⟨set (u # U) ⊆ Collect Λ.is-App⟩
    Λ.ide-char list.set-map)
  ultimately show ?thesis
    using set-Ide-subset-single-hd by force
  qed
  qed
  thus ?thesis
    by (simp add: Resid-Arr-self Std ide-char)
  qed
  qed
  also have [M ∘ N] @ u # U = (M ∘ N) # u # U
    by simp
  finally show ?thesis by blast
  qed
  qed
  qed
  show [¬ Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide;
    Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide]
    ⇒ ?thesis
  proof –
    assume *: ¬ Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide
    assume **: Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide
    have 10: filter notIde (map Λ.un-App2 (u # U)) = []
      using ** by (simp add: subset-eq)
    hence 4: stdz-insert (M ∘ N) (u # U) =
      map (λX. X ∘ Λ.Src N)
        (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) @
        map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))
          (standard-development N))
    using u 4 5 * ** Ide-iff-standard-development-empty MN
    by simp

```

```

have 6:  $\Lambda.Ide (\Lambda.Trq (\Lambda.un-App1 (last (u \# U))))$ 
using ***  $u Std Std-imp-Arr$ 
by ( $metis Arr-imp-arr-last in-mono \Lambda.Arr.simps(4) \Lambda.Ide-Trq \Lambda.arr-char$ 
 $\Lambda.lambda.collapse(3) last.simps last-in-set list.discI mem-Collect-eq$ )
show ?thesis
proof (intro conjI)
show  $Std (stdz-insert (M \circ N) (u \# U))$ 
proof –
have  $Std (map (\lambda X. X \circ \Lambda.Src N)$ 
 $(stdz-insert M (filter notIde (map \Lambda.un-App1 (u \# U)))) @$ 
 $map (\Lambda.App (\Lambda.Trq (\Lambda.un-App1 (last (u \# U))))$ 
 $(standard-development N))$ 
proof (intro Std-append)
show  $Std (map (\lambda X. X \circ \Lambda.Src N)$ 
 $(stdz-insert M (filter notIde$ 
 $(map \Lambda.un-App1 (u \# U))))$ 
using *  $A MN Std-map-App1 \Lambda.Ide-Src$  by presburger
show  $Std (map (\Lambda.App (\Lambda.Trq (\Lambda.un-App1 (last (u \# U))))$ 
 $(standard-development N))$ 
using  $MN 6 Std-map-App2 Std-standard-development$  by simp
show  $map (\lambda X. X \circ \Lambda.Src N)$ 
 $(stdz-insert M$ 
 $(filter notIde (map \Lambda.un-App1 (u \# U)))) = [] \vee$ 
 $map (\Lambda.App (\Lambda.Trq (\Lambda.un-App1 (last (u \# U))))$ 
 $(standard-development N) = [] \vee$ 
 $\Lambda.sseq (last (map (\lambda X. \Lambda.App X (\Lambda.Src N))$ 
 $(stdz-insert M$ 
 $(filter notIde (map \Lambda.un-App1 (u \# U))))$ 
 $(hd (map (\Lambda.App (\Lambda.Trq (\Lambda.un-App1 (last (u \# U))))$ 
 $(standard-development N))))$ 
proof (cases  $\Lambda.Ide N$ )
show  $\Lambda.Ide N \implies ?thesis$ 
using  $Ide-iff-standard-development-empty MN$  by blast
assume  $N: \neg \Lambda.Ide N$ 
have  $\Lambda.sseq (last (map (\lambda X. X \circ \Lambda.Src N)$ 
 $(stdz-insert M$ 
 $(filter notIde (map \Lambda.un-App1 (u \# U))))$ 
 $(hd (map (\Lambda.App (\Lambda.Trq (\Lambda.un-App1 (last (u \# U))))$ 
 $(standard-development N))))$ 
proof –
have  $hd (map (\Lambda.App (\Lambda.Trq (\Lambda.un-App1 (last (u \# U))))$ 
 $(standard-development N)) =$ 
 $\Lambda.App (\Lambda.Trq (\Lambda.un-App1 (last (u \# U))))$ 
 $(hd (standard-development N))$ 
by ( $meson Ide-iff-standard-development-empty MN N list.map-sel(1)$ )
moreover have  $last (map (\lambda X. X \circ \Lambda.Src N)$ 
 $(stdz-insert M$ 
 $(filter notIde (map \Lambda.un-App1 (u \# U)))) =$ 
 $\Lambda.App (last (stdz-insert M$ 

```

```

      (filter notIde
        (map  $\Lambda.un-App1$  (u # U))))))
    ( $\Lambda.Src$  N)
  by (metis * A Ide.simps(1) Resid.simps(1) ide-char last-map)
moreover have  $\Lambda.sseq$  ... ( $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))
  (hd (standard-development N)))
proof -
  have  $\gamma$ :  $\Lambda.elementary-reduction$ 
    (last (stdz-insert M (filter notIde
      (map  $\Lambda.un-App1$  (u # U))))))
    using * A
  by (metis Ide.simps(1) Resid.simps(2) ide-char last-in-set
    mem-Collect-eq subset-iff)
moreover
  have  $\Lambda.elementary-reduction$  (hd (standard-development N))
    using MN N hd-in-set set-standard-development
      Ide-iff-standard-development-empty
  by blast
moreover have  $\Lambda.Src$  N =  $\Lambda.Src$  (hd (standard-development N))
  using MN N Src-hd-standard-development by auto
moreover have  $\Lambda.Trq$  (last (stdz-insert M
  (filter notIde
    (map  $\Lambda.un-App1$  (u # U)))))) =
   $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U)))
proof -
  have [ $\Lambda.Trq$  (last (stdz-insert M
    (filter notIde
      (map  $\Lambda.un-App1$  (u # U))))))] =
    [ $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U)))]
proof -
  have  $\Lambda.Trq$  (last (stdz-insert M
    (filter notIde
      (map  $\Lambda.un-App1$  (u # U)))))) =
     $\Lambda.Trq$  (last (map  $\Lambda.un-App1$  (u # U)))
proof -
  have  $\Lambda.Trq$  (last (stdz-insert M
    (filter notIde (map  $\Lambda.un-App1$  (u # U)))))) =
     $\Lambda.Trq$  (last (M # filter notIde (map  $\Lambda.un-App1$  (u # U))))
    using * A Trq-last-eqI by blast
  also have ... =  $\Lambda.Trq$  (last ([M] @ filter notIde
    (map  $\Lambda.un-App1$  (u # U))))
    by simp
  also have ... =  $\Lambda.Trq$  (last (filter notIde
    (map  $\Lambda.un-App1$  (u # U))))
proof -
  have seq [M] (filter notIde (map  $\Lambda.un-App1$  (u # U)))
proof
  show Arr [M]
    using MN by simp

```

```

show Arr (filter notIde (map  $\Lambda.un-App1$  (u # U)))
  using * Std-imp-Arr
  by (metis (no-types, lifting)
    X empty-filter-conv list.set-map mem-Collect-eq subsetI)
show  $\Lambda.Trq$  (last [M]) =
   $\Lambda.Src$  (hd (filter notIde (map  $\Lambda.un-App1$  (u # U))))
proof -
  have  $\Lambda.Trq$  (last [M]) =  $\Lambda.Trq$  M
  using MN by simp
  also have ... =  $\Lambda.Src$  ( $\Lambda.un-App1$  u)
  by (metis Trg-last-Src-hd-eqI  $\Lambda.Src.simps(4)$ 
     $\Lambda.Trq.simps(3)$   $\Lambda.lambda.collapse(3)$ 
     $\Lambda.lambda.inject(3)$  last-ConsL list.sel(1) seq u)
  also have ... =  $\Lambda.Src$  (hd (map  $\Lambda.un-App1$  (u # U)))
  by auto
  also have ... =  $\Lambda.Src$  (hd (filter notIde
    (map  $\Lambda.un-App1$  (u # U))))
  using u 5 10 by force
  finally show ?thesis by blast
qed
qed
thus ?thesis
  by (metis Arr.simps(1) last-appendR seq-char)
qed
also have ... =  $\Lambda.Trq$  (last (map  $\Lambda.un-App1$  (u # U)))
proof -
  have filter ( $\lambda u. \neg \Lambda.Ide$  u) (map  $\Lambda.un-App1$  (u # U))  $\sim^*$ 
    map  $\Lambda.un-App1$  (u # U)
  using * *** u Std Std-imp-Arr Arr-map-un-App1 [of u # U]
    cong-filter-notIde
  by (metis (mono-tags, lifting) empty-filter-conv
    filter-notIde-Ide list.discI list.set-map
    mem-Collect-eq set-ConsD subset-code(1))
  thus ?thesis
  using cong-implies-coterminal Trg-last-eqI
  by presburger
qed
finally show ?thesis by blast
qed
thus ?thesis
  by (simp add: last-map)
qed
moreover
have  $\Lambda.Ide$  ( $\Lambda.Trq$  (last (stdz-insert M
  (filter notIde
    (map  $\Lambda.un-App1$  (u # U))))))
  using 7  $\Lambda.Ide-Trq$   $\Lambda.elementary-reduction-is-arr$  by blast
moreover have  $\Lambda.Ide$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))
  using 6 by blast

```

```

      ultimately show ?thesis by simp
    qed
    ultimately show ?thesis
      using  $\Lambda.sseq.simps(4)$  by blast
    qed
    ultimately show ?thesis by argo
  qed
  thus ?thesis by blast
qed
qed
thus ?thesis
  using 4 by simp
qed
show  $\neg Ide ((M \circ N) \# u \# U) \longrightarrow$ 
  stdz-insert  $(M \circ N) (u \# U) \sim^* (M \circ N) \# u \# U$ 
proof
show stdz-insert  $(M \circ N) (u \# U) \sim^* (M \circ N) \# u \# U$ 
proof (cases  $\Lambda.Ide N$ )
  assume  $N: \Lambda.Ide N$ 
  have stdz-insert  $(M \circ N) (u \# U) =$ 
    map  $(\lambda X. X \circ N)$ 
      (stdz-insert  $M (filter notIde$ 
        (map  $\Lambda.un-App1 (u \# U)))$ )
  using 4  $N MN Ide-iff-standard-development-empty \Lambda.Ide-iff-Src-self$ 
  by force
  also have ...  $\sim^* map (\lambda X. X \circ N)$ 
    (M # filter notIde
      (map  $\Lambda.un-App1 (u \# U)$ ))
  using *  $A MN N \Lambda.Ide-Src cong-map-App2 \Lambda.Ide-iff-Src-self$ 
  by blast
  also have map  $(\lambda X. X \circ N)$ 
    (M # filter notIde
      (map  $\Lambda.un-App1 (u \# U)$ )) =
    [M  $\circ N$ ] @
      map  $(\lambda X. \Lambda.App X N)$ 
        (filter notIde (map  $\Lambda.un-App1 (u \# U)$ ))
  by auto
  also have [M  $\circ N$ ] @
    map  $(\lambda X. X \circ N)$ 
      (filter notIde (map  $\Lambda.un-App1 (u \# U)$ ))  $\sim^*$ 
    [M  $\circ N$ ] @ map  $(\lambda X. X \circ N)$  (map  $\Lambda.un-App1 (u \# U)$ )
proof (intro cong-append)
  show seq [M  $\circ N$ ]
    (map  $(\lambda X. X \circ N)$ 
      (filter notIde (map  $\Lambda.un-App1 (u \# U)$ )))
proof
  have 20: Arr (map  $\Lambda.un-App1 (u \# U)$ )
  using ***  $u Std Arr-map-un-App1$ 
  by (metis Std-imp-Arr insert-subset list.discI list.simps(15))

```

```

      mem-Collect-eq)
show Arr [M ◦ N]
  using MN by auto
show 21: Arr (map (λX. X ◦ N)
              (filter notIde (map Λ.un-App1 (u # U))))
proof –
  have Arr (filter notIde (map Λ.un-App1 (u # U)))
    using u 20 cong-filter-notIde
  by (metis (no-types, lifting) * Std-imp-Arr
      ⟨Std (filter notIde (map Λ.un-App1 (u # U)))⟩
      empty-filter-conv list.set-map mem-Collect-eq subsetI)
  thus ?thesis
    using MN N Arr-map-App1 Λ.Ide-Src by presburger
qed
show Λ.Trq (last [M ◦ N]) =
  Λ.Src (hd (map (λX. X ◦ N)
                (filter notIde (map Λ.un-App1 (u # U)))))
proof –
  have Λ.Trq (last [M ◦ N]) = Λ.Trq M ◦ N
    using MN N Λ.Ide-iff-Trq-self by simp
  also have ... = Λ.Src (Λ.un-App1 u) ◦ N
    using MN u seq seq-char
  by (metis Trq-last-Src-hd-eqI calculation Λ.Src-Src Λ.Src-Trq
      Λ.Src-eq-iff(2) Λ.is-App-def Λ.lambda.sel(3) list.sel(1))
  also have ... = Λ.Src (Λ.un-App1 u ◦ N)
    using MN N Λ.Ide-iff-App-self by simp
  also have ... = Λ.Src (hd (map (λX. X ◦ N)
                                (map Λ.un-App1 (u # U))))
    by simp
  also have ... = Λ.Src (hd (map (λX. X ◦ N)
                                (filter notIde
                                  (map Λ.un-App1 (u # U)))))
    by simp
proof –
  have cong (map Λ.un-App1 (u # U))
    (filter notIde (map Λ.un-App1 (u # U)))
    using * 20 21 cong-filter-notIde
  by (metis Arr.simps(1) filter-notIde-Ide map-is-Nil-conv)
  thus ?thesis
    by (metis (no-types, lifting) Ide.simps(1) Resid.simps(2)
        Src-hd-eqI hd-map ide-char Λ.Src.simps(4)
        list.distinct(1) list.simps(9))
qed
finally show ?thesis by blast
qed
qed
show cong [M ◦ N] [M ◦ N]
  using MN
  by (meson head-redex-decomp Λ.Arr.simps(4) Λ.Arr-Src
      prfx-transitive)

```

```

show map (λX. X ∘ N) (filter notIde (map Λ.un-App1 (u # U))) *~*
  map (λX. X ∘ N) (map Λ.un-App1 (u # U))
proof –
  have Arr (map Λ.un-App1 (u # U))
  using *** u Std Arr-map-un-App1
  by (metis Std-imp-Arr insert-subset list.discI list.simps(15)
    mem-Collect-eq)
moreover have ¬ Ide (map Λ.un-App1 (u # U))
  using *
  by (metis Collect-cong Λ.ide-char list.set-map
    set-Ide-subset-ide)
ultimately show ?thesis
  using *** u MN N cong-filter-notIde cong-map-App2
  by (meson Λ.Ide-Src)
qed
qed
also have [M ∘ N] @ map (λX. X ∘ N) (map Λ.un-App1 (u # U)) *~*
  [M ∘ N] @ u # U
proof –
  have map (λX. X ∘ N) (map Λ.un-App1 (u # U)) *~* u # U
proof –
  have map (λX. X ∘ N) (map Λ.un-App1 (u # U)) = u # U
proof (intro map-App-map-un-App1)
  show Arr (u # U)
  using Std Std-imp-Arr by simp
  show set (u # U) ⊆ Collect Λ.is-App
  using *** u by auto
  show Λ.Ide N
  using N by simp
  show Λ.un-App2 ‘ set (u # U) ⊆ {N}
proof –
  have Λ.Src (Λ.un-App2 u) = Λ.Trq N
  using ** seq u seq-char N
  apply (cases u)
  apply simp-all
  by (metis Trq-last-Src-hd-eqI Λ.Src.simps(4) Λ.Trq.simps(3)
    Λ.lambda.inject(3) last-ConsL list.sel(1) seq)
moreover have Λ.Ide (Λ.un-App2 u) ∧ Λ.Ide N
  using ** N by simp
moreover have Ide (map Λ.un-App2 (u # U))
  using ** Std Std-imp-Arr Arr-map-un-App2
  by (metis Collect-cong Ide-char
    ⟨Arr (u # U)⟩ ⟨set (u # U) ⊆ Collect Λ.is-App⟩
    Λ.ide-char list.set-map)
ultimately show ?thesis
  by (metis hd-map Λ.Ide-iff-Src-self Λ.Ide-iff-Trq-self
    Λ.Ide-implies-Arr list.discI list.sel(1)
    list.set-map set-Ide-subset-single-hd)
qed

```

```

qed
thus ?thesis
  by (simp add: Resid-Arr-self Std ide-char)
qed
thus ?thesis
  using MN cong-append
  by (metis (no-types, lifting) 1 cong-standard-development
    cong-transitive  $\Lambda$ .Arr.simps(4) seq)
qed
also have  $[M \circ N] @ (u \# U) = (M \circ N) \# u \# U$ 
  by simp
finally show ?thesis by blast
next
assume  $N: \neg \Lambda$ .Ide  $N$ 
have stdz-insert  $(M \circ N) (u \# U) =$ 
  map  $(\lambda X. X \circ \Lambda$ .Src  $N)$ 
  (stdz-insert  $M$  (filter notIde (map  $\Lambda$ .un-App1  $(u \# U)$ ))) @
  map  $(\Lambda$ .App  $(\Lambda$ .Trg  $(\Lambda$ .un-App1 (last  $(u \# U)$ ))))
  (standard-development  $N$ )
  using 4 by simp
also have ... *~* map  $(\lambda X. X \circ \Lambda$ .Src  $N)$ 
  ( $M \#$  filter notIde (map  $\Lambda$ .un-App1  $(u \# U)$ )) @
  map  $(\Lambda$ .App  $(\Lambda$ .Trg  $(\Lambda$ .un-App1 (last  $(u \# U)$ )))) [ $N$ ]
proof (intro cong-append)
  show 23: map  $(\lambda X. X \circ \Lambda$ .Src  $N)$ 
  (stdz-insert  $M$  (filter notIde (map  $\Lambda$ .un-App1  $(u \# U)$ ))) *~*
  map  $(\lambda X. X \circ \Lambda$ .Src  $N)$ 
  ( $M \#$  filter notIde (map  $\Lambda$ .un-App1  $(u \# U)$ ))
  using * A MN  $\Lambda$ .Ide-Src cong-map-App2 by blast
  show 22: map  $(\Lambda$ .App  $(\Lambda$ .Trg  $(\Lambda$ .un-App1 (last  $(u \# U)$ ))))
  (standard-development  $N$ ) *~*
  map  $(\Lambda$ .App  $(\Lambda$ .Trg  $(\Lambda$ .un-App1 (last  $(u \# U)$ )))) [ $N$ ]
  using 6 ***  $u$  Std Std-imp-Arr MN  $N$  cong-standard-development
  cong-map-App1
  by presburger
  show seq (map  $(\lambda X. X \circ \Lambda$ .Src  $N)$ 
  (stdz-insert  $M$  (filter notIde
  (map  $\Lambda$ .un-App1  $(u \# U)$ ))))
  (map  $(\Lambda$ .App  $(\Lambda$ .Trg  $(\Lambda$ .un-App1 (last  $(u \# U)$ ))))
  (standard-development  $N$ ))
proof –
  have seq (map  $(\lambda X. X \circ \Lambda$ .Src  $N)$ 
  ( $M \#$  filter notIde
  (map  $\Lambda$ .un-App1  $(u \# U)$ )))
  (map  $(\Lambda$ .App  $(\Lambda$ .Trg  $(\Lambda$ .un-App1 (last  $(u \# U)$ )))) [ $N$ ])
proof
  show 26: Arr (map  $(\lambda X. X \circ \Lambda$ .Src  $N)$ 
  ( $M \#$  filter notIde
  (map  $\Lambda$ .un-App1  $(u \# U)$ )))

```

```

    by (metis 23 Con-implies-Arr(2) Ide.simps(1) ide-char)
  show Arr (map (Λ.App (Λ.Trq (Λ.un-App1 (last (u # U)))))) [N])
    by (meson 22 arr-char con-implies-arr(2) prfx-implies-con)
  show Λ.Trq (last (map (λX. X ◦ Λ.Src N)
    (M # filter notIde
      (map Λ.un-App1 (u # U)))))) =
    Λ.Src (hd (map (Λ.App (Λ.Trq (Λ.un-App1 (last (u # U))))))
      [N]))
proof –
  have Λ.Trq (last (map (λX. X ◦ Λ.Src N)
    (M # map Λ.un-App1 (u # U))))
    ~
    Λ.Trq (last (map (λX. X ◦ Λ.Src N)
      (M # filter notIde
        (map Λ.un-App1 (u # U))))))
proof –
  have targets (map (λX. X ◦ Λ.Src N)
    (M # filter notIde
      (map Λ.un-App1 (u # U)))) =
    targets (map (λX. X ◦ Λ.Src N)
      (M # map Λ.un-App1 (u # U)))
proof –
  have map (λX. X ◦ Λ.Src N)
    (M # filter notIde (map Λ.un-App1 (u # U))) *~*
    map (λX. X ◦ Λ.Src N)
      (M # map Λ.un-App1 (u # U))
proof –
  have map (λX. X ◦ Λ.Src N)
    (M # map Λ.un-App1 (u # U)) =
    map (λX. X ◦ Λ.Src N)
      ([M] @ map Λ.un-App1 (u # U))
    by simp
  also have cong ... (map (λX. X ◦ Λ.Src N)
    ([M] @ filter notIde
      (map Λ.un-App1 (u # U))))
proof –
  have [M] @ map Λ.un-App1 (u # U) *~*
    [M] @ filter notIde
      (map Λ.un-App1 (u # U))
proof (intro cong-append)
  show cong [M] [M]
    using MN
    by (meson head-redex-decomp prfx-transitive)
  show seq [M] (map Λ.un-App1 (u # U))
proof
  show Arr [M]
    using MN by simp
  show Arr (map Λ.un-App1 (u # U))
    using *** u Std Arr-map-un-App1

```

```

    by (metis Std-imp-Arr insert-subset list.discI
        list.simps(15) mem-Collect-eq)
  show  $\Lambda.Trq$  (last [M]) =
     $\Lambda.Src$  (hd (map  $\Lambda.un-App1$  (u # U)))
    using MN u seq seq-char Srcs-simp $_{\Lambda P}$  by auto
  qed
  show cong (map  $\Lambda.un-App1$  (u # U))
    (filter notIde
      (map  $\Lambda.un-App1$  (u # U)))
  proof -
    have Arr (map  $\Lambda.un-App1$  (u # U))
    by (metis *** Arr-map-un-App1 Std Std-imp-Arr
        insert-subset list.discI list.simps(15)
        mem-Collect-eq u)
    moreover have  $\neg Ide$  (map  $\Lambda.un-App1$  (u # U))
    using * set-Ide-subset-ide by fastforce
    ultimately show ?thesis
    using cong-filter-notIde by blast
  qed
  qed
  thus map ( $\lambda X. X \circ \Lambda.Src N$ )
    ([M] @ map  $\Lambda.un-App1$  (u # U)) *~*
    map ( $\lambda X. X \circ \Lambda.Src N$ )
    ([M] @ filter notIde (map  $\Lambda.un-App1$  (u # U)))
    using MN cong-map-App2  $\Lambda.Ide-Src$  by presburger
  qed
  finally show ?thesis by simp
  qed
  thus ?thesis
    using cong-implies-coterminal by blast
  qed
  moreover have [ $\Lambda.Trq$  (last (map ( $\lambda X. X \circ \Lambda.Src N$ )
    (M # map  $\Lambda.un-App1$  (u # U))))]  $\in$ 
    targets (map ( $\lambda X. X \circ \Lambda.Src N$ )
      (M # map  $\Lambda.un-App1$  (u # U)))
  by (metis (no-types, lifting) 26 calculation mem-Collect-eq
    single-Trg-last-in-targets targets-char $_{\Lambda P}$ )
  moreover have [ $\Lambda.Trq$  (last (map ( $\lambda X. X \circ \Lambda.Src N$ )
    (M # filter notIde
      (map  $\Lambda.un-App1$  (u # U)))))]  $\in$ 
    targets (map ( $\lambda X. X \circ \Lambda.Src N$ )
      (M # filter notIde
        (map  $\Lambda.un-App1$  (u # U))))
  using 26 single-Trg-last-in-targets by blast
  ultimately show ?thesis
    by (metis (no-types, lifting) 26 Ide.simps(1-2) Resid-rec(1)
        in-targets-iff ide-char)
  qed
  moreover have  $\Lambda.Ide$  ( $\Lambda.Trq$  (last (map ( $\lambda X. X \circ \Lambda.Src N$ )

```

```

(M # map  $\Lambda.un-App1$  (u # U))))
by (metis 6 MN  $\Lambda.Ide.simps(4)$   $\Lambda.Ide-Src$   $\Lambda.Trq.simps(3)$ 
 $\Lambda.Trq-Src$  last-ConsR last-map list.distinct(1)
list.simps(9))
moreover have  $\Lambda.Ide$  ( $\Lambda.Trq$  (last (map ( $\lambda X. X \circ \Lambda.Src$  N)
(M # filter notIde
(map  $\Lambda.un-App1$  (u # U))))))
using  $\Lambda.ide-backward-stable$  calculation(1-2) by fast
ultimately show ?thesis
by (metis (no-types, lifting) 6 MN hd-map
 $\Lambda.Ide-iff-Src-self$   $\Lambda.Ide-implies-Arr$   $\Lambda.Src.simps(4)$ 
 $\Lambda.Trq.simps(3)$   $\Lambda.Trq-Src$   $\Lambda.cong-Ide-are-eq$ 
last.simps last-map list.distinct(1) list.map-disc-iff
list.sel(1))
qed
qed
thus ?thesis
using 22 23 cong-respects-seqP by presburger
qed
qed
also have map ( $\lambda X. X \circ \Lambda.Src$  N)
(M # filter notIde (map  $\Lambda.un-App1$  (u # U))) @
map ( $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))) [N] =
[M  $\circ$   $\Lambda.Src$  N] @
map ( $\lambda X. X \circ \Lambda.Src$  N)
(filter notIde (map  $\Lambda.un-App1$  (u # U))) @
[ $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))) N]
by simp
also have 1: [M  $\circ$   $\Lambda.Src$  N] @
map ( $\lambda X. X \circ \Lambda.Src$  N)
(filter notIde (map  $\Lambda.un-App1$  (u # U))) @
[ $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))) N] *~*
[M  $\circ$   $\Lambda.Src$  N] @
map ( $\lambda X. X \circ \Lambda.Src$  N) (map  $\Lambda.un-App1$  (u # U)) @
[ $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))) N]
proof (intro cong-append)
show [M  $\circ$   $\Lambda.Src$  N] *~* [M  $\circ$   $\Lambda.Src$  N]
using MN
by (meson head-redex-decomp lambda-calculus.Arr.simps(4)
lambda-calculus.Arr-Src prfx-transitive)
show 21: map ( $\lambda X. X \circ \Lambda.Src$  N)
(filter notIde (map  $\Lambda.un-App1$  (u # U))) *~*
map ( $\lambda X. X \circ \Lambda.Src$  N) (map  $\Lambda.un-App1$  (u # U))
proof -
have filter notIde (map  $\Lambda.un-App1$  (u # U)) *~*
map  $\Lambda.un-App1$  (u # U)
proof -
have  $\neg Ide$  (map  $\Lambda.un-App1$  (u # U))
using *

```

```

    by (metis Collect-cong  $\Lambda$ .ide-char list.set-map
        set-Ide-subset-ide)
  thus ?thesis
    using *** u Std Std-imp-Arr Arr-map-un-App1
        cong-filter-notIde
    by (metis  $\langle \neg$  Ide (map  $\Lambda$ .un-App1 (u # U)) $\rangle$ 
        list.distinct(1) mem-Collect-eq set-ConsD
        subset-code(1))
qed
thus ?thesis
    using MN cong-map-App2 [of  $\Lambda$ .Src N]  $\Lambda$ .Ide-Src by presburger
qed
show [ $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))  $\circ$  N] *~*
    [ $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))  $\circ$  N]
  by (metis 6 Con-implies-Arr(1) MN  $\Lambda$ .Ide-implies-Arr arr-char
      cong-reflexive  $\Lambda$ .Ide-iff-Src-self neq-Nil-conv
      orthogonal-App-single-single(1))
show seq (map ( $\lambda X$ . X  $\circ$   $\Lambda$ .Src N)
    (filter notIde (map  $\Lambda$ .un-App1 (u # U))))
    [ $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))  $\circ$  N]
proof
  show Arr (map ( $\lambda X$ . X  $\circ$   $\Lambda$ .Src N)
    (filter notIde (map  $\Lambda$ .un-App1 (u # U))))
    by (metis 21 Con-implies-Arr(2) Ide.simps(1) ide-char)
  show Arr [ $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))  $\circ$  N]
    by (metis Con-implies-Arr(2) Ide.simps(1)
         $\langle$ [ $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))  $\circ$  N] *~*
            [ $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))  $\circ$  N] $\rangle$ 
        ide-char)
  show  $\Lambda$ .Trg (last (map ( $\lambda X$ . X  $\circ$   $\Lambda$ .Src N)
    (filter notIde
        (map  $\Lambda$ .un-App1 (u # U))))) =
     $\Lambda$ .Src (hd [ $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))  $\circ$  N])
    by (metis (no-types, lifting) 6 21 MN Trg-last-eqI
         $\Lambda$ .Ide-iff-Src-self  $\Lambda$ .Ide-implies-Arr  $\Lambda$ .Src.simps(4)
         $\Lambda$ .Trg.simps(3)  $\Lambda$ .Trg-Src last-map list.distinct(1)
        list.map-disc-iff list.sel(1))
qed
show seq [M  $\circ$   $\Lambda$ .Src N]
    (map ( $\lambda X$ . X  $\circ$   $\Lambda$ .Src N)
        (filter notIde (map  $\Lambda$ .un-App1 (u # U))) @
        [ $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))  $\circ$  N])
proof
  show Arr [M  $\circ$   $\Lambda$ .Src N]
    using MN by simp
  show Arr (map ( $\lambda X$ . X  $\circ$   $\Lambda$ .Src N)
    (filter notIde (map  $\Lambda$ .un-App1 (u # U))) @
    [ $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))  $\circ$  N])
    apply (intro Arr-appendIP)

```

```

apply (metis 21 Con-implies-Arr(2) Ide.simps(1) ide-char)
apply (metis Con-implies-Arr(1) Ide.simps(1)
  ⟨[Λ.Trig (Λ.un-App1 (last (u # U))) ∘ N] *~*
  [Λ.Trig (Λ.un-App1 (last (u # U))) ∘ N]⟩ ide-char)
by (metis (no-types, lifting) 21 Arr.simps(1)
  Arr-append-iffP Con-implies-Arr(2) Ide.simps(1)
  append-is-Nil-conv calculation ide-char not-Cons-self2)
show Λ.Trig (last [M ∘ Λ.Src N]) =
  Λ.Src (hd (map (λX. X ∘ Λ.Src N)
    (filter notIde
      (map Λ.un-App1 (u # U))) @
      [Λ.Trig (Λ.un-App1 (last (u # U))) ∘ N]))
by (metis (no-types, lifting) Con-implies-Arr(2) Ide.simps(1)
  Trig-last-Src-hd-eqI append-is-Nil-conv arr-append-imp-seq
  arr-char calculation ide-char not-Cons-self2)
qed
qed
also have [M ∘ Λ.Src N] @
  map (λX. X ∘ Λ.Src N)(map Λ.un-App1 (u # U)) @
  [Λ.Trig (Λ.un-App1 (last (u # U))) ∘ N] *~*
  [M ∘ Λ.Src N] @
  [Λ.Trig M ∘ N] @
  map (λX. X ∘ Λ.Trig N) (map Λ.un-App1 (u # U))
proof (intro cong-append [of [Λ.App M (Λ.Src N)]])
show seq [M ∘ Λ.Src N]
  (map (λX. X ∘ Λ.Src N)
    (map Λ.un-App1 (u # U)) @
    [Λ.Trig (Λ.un-App1 (last (u # U))) ∘ N])
proof
show Arr [M ∘ Λ.Src N]
  using MN by simp
show Arr (map (λX. X ∘ Λ.Src N)
  (map Λ.un-App1 (u # U)) @
  [Λ.Trig (Λ.un-App1 (last (u # U))) ∘ N])
  by (metis (no-types, lifting) 1 Con-append(2) Con-implies-Arr(2)
  Ide.simps(1) append-is-Nil-conv ide-char not-Cons-self2)
show Λ.Trig (last [M ∘ Λ.Src N]) =
  Λ.Src (hd (map (λX. X ∘ Λ.Src N)
    (map Λ.un-App1 (u # U)) @
    [Λ.Trig (Λ.un-App1 (last (u # U))) ∘ N]))
proof –
have Λ.Trig M = Λ.Src (Λ.un-App1 u)
  using u seq
  by (metis Trig-last-Src-hd-eqI Λ.Src.simps(4) Λ.Trig.simps(3)
  Λ.lambda.collapse(3) Λ.lambda.inject(3) last-ConsL
  list.sel(1))
thus ?thesis
  using MN by auto
qed

```

```

qed
show  $[M \circ \Lambda.\text{Src } N] \text{ *}\sim\text{* } [M \circ \Lambda.\text{Src } N]$ 
  using MN
  by (metis head-redex-decomp  $\Lambda.\text{Arr}.\text{sims}(4)$   $\Lambda.\text{Arr}.\text{Src}$ 
    prfx-transitive)
show  $\text{map } (\lambda X. X \circ \Lambda.\text{Src } N) (\text{map } \Lambda.\text{un-App1 } (u \# U)) \text{ @}$ 
   $[\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } (u \# U))) \circ N] \text{ *}\sim\text{*}$ 
   $[\Lambda.\text{Trg } M \circ N] \text{ @}$ 
   $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } N) (\text{map } \Lambda.\text{un-App1 } (u \# U))$ 
proof –
  have  $\text{map } (\lambda X. X \circ \Lambda.\text{Src } (\text{hd } [N])) (\text{map } \Lambda.\text{un-App1 } (u \# U)) \text{ @}$ 
   $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\text{last } (\text{map } \Lambda.\text{un-App1 } (u \# U)))) [N] \text{ *}\sim\text{*}$ 
   $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } (\text{map } \Lambda.\text{un-App1 } (u \# U)))) [N] \text{ @}$ 
   $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } (\text{last } [N])) (\text{map } \Lambda.\text{un-App1 } (u \# U))$ 
proof –
  have Arr  $(\text{map } \Lambda.\text{un-App1 } (u \# U))$ 
  using Std *** u Arr-map-un-App1
  by (metis Std-imp-Arr insert-subset list.discI list.sims(15)
    mem-Collect-eq)
  moreover have Arr  $[N]$ 
  using MN by simp
  ultimately show ?thesis
  using orthogonal-App-cong by blast
qed
moreover
have  $\text{map } (\Lambda.\text{App } (\Lambda.\text{Src } (\text{hd } (\text{map } \Lambda.\text{un-App1 } (u \# U)))) [N] =$ 
   $[\Lambda.\text{Trg } M \circ N]$ 
  by (metis Trg-last-Src-hd-eqI lambda-calculus.Src.sims(4)
     $\Lambda.\text{Trg}.\text{sims}(3)$   $\Lambda.\text{lambda}.\text{collapse}(3)$   $\Lambda.\text{lambda}.\text{sel}(3)$ 
    last-ConsL list.sel(1) list.sims(8) list.sims(9) seq u)
  moreover have  $[\Lambda.\text{Trg } (\Lambda.\text{un-App1 } (\text{last } (u \# U))) \circ N] =$ 
   $\text{map } (\Lambda.\text{App } (\Lambda.\text{Trg } (\text{last } (\text{map } \Lambda.\text{un-App1 } (u \# U)))) [N]$ 
  by (simp add: last-map)
  ultimately show ?thesis
  using last-map by auto
qed
qed
also have  $[M \circ \Lambda.\text{Src } N] \text{ @}$ 
   $[\Lambda.\text{Trg } M \circ N] \text{ @}$ 
   $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } N) (\text{map } \Lambda.\text{un-App1 } (u \# U)) =$ 
   $([M \circ \Lambda.\text{Src } N] \text{ @ } [\Lambda.\text{Trg } M \circ N]) \text{ @}$ 
   $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } N) (\text{map } \Lambda.\text{un-App1 } (u \# U))$ 
  by simp
also have ...  $\text{ *}\sim\text{* } [M \circ N] \text{ @ } (u \# U)$ 
proof (intro cong-append)
  show  $[M \circ \Lambda.\text{Src } N] \text{ @ } [\Lambda.\text{Trg } M \circ N] \text{ *}\sim\text{* } [M \circ N]$ 
  using MN  $\Lambda.\text{resid-Arr-self}$   $\Lambda.\text{Arr-not-Nil}$   $\Lambda.\text{Ide-Trg ide-char}$ 
  by auto
show 1:  $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } N) (\text{map } \Lambda.\text{un-App1 } (u \# U)) \text{ *}\sim\text{* } u \# U$ 

```

```

proof –
  have  $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } N) (\text{map } \Lambda.\text{un-App1 } (u \# U)) = u \# U$ 
  proof (intro map-App-map-un-App1)
  show  $\text{Arr } (u \# U)$ 
    using Std Std-imp-Arr by simp
  show  $\text{set } (u \# U) \subseteq \text{Collect } \Lambda.\text{is-App}$ 
    using *** u by auto
  show  $\Lambda.\text{Ide } (\Lambda.\text{Trg } N)$ 
    using MN  $\Lambda.\text{Ide-Trg}$  by simp
  show  $\Lambda.\text{un-App2 } \text{' set } (u \# U) \subseteq \{\Lambda.\text{Trg } N\}$ 
  proof –
    have  $\Lambda.\text{Src } (\Lambda.\text{un-App2 } u) = \Lambda.\text{Trg } N$ 
      using u seq seq-char
      apply (cases u)
      apply simp-all
      by (metis Trg-last-Src-hd-eqI  $\Lambda.\text{Src.simps}(4)$   $\Lambda.\text{Trg.simps}(3)$   $\Lambda.\text{lambda.inject}(3)$  last-ConsL list.sel(1) seq)
    moreover have  $\Lambda.\text{Ide } (\Lambda.\text{un-App2 } u)$ 
      using ** by simp
    moreover have  $\text{Ide } (\text{map } \Lambda.\text{un-App2 } (u \# U))$ 
      using ** Std Std-imp-Arr Arr-map-un-App2
      by (metis Collect-cong Ide-char
         $\langle \text{Arr } (u \# U) \rangle \langle \text{set } (u \# U) \subseteq \text{Collect } \Lambda.\text{is-App} \rangle$ 
         $\Lambda.\text{ide-char list.set-map}$ )
    ultimately show ?thesis
      by (metis  $\Lambda.\text{Ide-iff-Src-self}$   $\Lambda.\text{Ide-implies-Arr}$  list.sel(1) list.set-map list.simps(9) set-Ide-subset-single-hd singleton-insert-inj-eq)

  qed
  qed
  thus ?thesis
    by (simp add: Resid-Arr-self Std ide-char)
  qed
  show  $\text{seq } ([M \circ \Lambda.\text{Src } N] @ [\Lambda.\text{Trg } M \circ N])$ 
    ( $\text{map } (\lambda X. X \circ \Lambda.\text{Trg } N) (\text{map } \Lambda.\text{un-App1 } (u \# U))$ )
  proof
    show  $\text{Arr } ([M \circ \Lambda.\text{Src } N] @ [\Lambda.\text{Trg } M \circ N])$ 
      using MN by simp
    show  $\text{Arr } (\text{map } (\lambda X. X \circ \Lambda.\text{Trg } N) (\text{map } \Lambda.\text{un-App1 } (u \# U)))$ 
      using MN Std Std-imp-Arr Arr-map-un-App1 Arr-map-App1
      by (metis 1 Con-implies-Arr(1) Ide.simps(1) ide-char)
    show  $\Lambda.\text{Trg } (\text{last } ([M \circ \Lambda.\text{Src } N] @ [\Lambda.\text{Trg } M \circ N])) =$ 
       $\Lambda.\text{Src } (\text{hd } (\text{map } (\lambda X. X \circ \Lambda.\text{Trg } N) (\text{map } \Lambda.\text{un-App1 } (u \# U))))$ 
      using MN Std Std-imp-Arr Arr-map-un-App1 Arr-map-App1
       $\text{seq seq-char } u \text{ Srcs-simp}_{\Lambda P}$  by auto

  qed
  qed
  also have  $[M \circ N] @ (u \# U) = (M \circ N) \# u \# U$ 
    by simp

```

```

    finally show ?thesis by blast
  qed
  qed
  qed
  qed
  show  $\llbracket \neg \Lambda.un-App1 \text{ ' set } (u \# U) \subseteq Collect \Lambda.Ide;$ 
     $\neg \Lambda.un-App2 \text{ ' set } (u \# U) \subseteq Collect \Lambda.Ide \rrbracket$ 
     $\implies ?thesis$ 
  proof -
    assume *:  $\neg \Lambda.un-App1 \text{ ' set } (u \# U) \subseteq Collect \Lambda.Ide$ 
    assume **:  $\neg \Lambda.un-App2 \text{ ' set } (u \# U) \subseteq Collect \Lambda.Ide$ 
    show ?thesis
    proof (intro conjI)
      show Std (stdz-insert (M o N) (u # U))
    proof -
      have Std (map ( $\lambda X. X \circ \Lambda.Src N$ )
        (stdz-insert M (filter notIde (map  $\Lambda.un-App1$  (u # U)))) @
        map ( $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))))
        (stdz-insert N (filter notIde (map  $\Lambda.un-App2$  (u # U))))))
    proof (intro Std-append)
      show Std (map ( $\lambda X. X \circ \Lambda.Src N$ )
        (stdz-insert M (filter notIde (map  $\Lambda.un-App1$  (u # U))))))
      using * A  $\Lambda.Ide-Src$  MN Std-map-App1 by presburger
      show Std (map ( $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))))
        (stdz-insert N (filter notIde (map  $\Lambda.un-App2$  (u # U))))))
    proof -
      have  $\Lambda.Arr$  ( $\Lambda.un-App1$  (last (u # U)))
      by (metis ***  $\Lambda.Arr.simps(4)$  Std Std-imp-Arr Arr.simps(2)
        Arr-append-iffP append-butlast-last-id append-self-conv2
         $\Lambda.arr-char$   $\Lambda.lambda.collapse(3)$  last.simps last-in-set
        list.discI mem-Collect-eq subset-code(1) u)
      thus ?thesis
      using ** B  $\Lambda.Ide-Trq$  MN Std-map-App2 by presburger
    qed
  show map ( $\lambda X. X \circ \Lambda.Src N$ )
    (stdz-insert M (filter notIde (map  $\Lambda.un-App1$  (u # U)))) = []  $\vee$ 
    map ( $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))))
    (stdz-insert N (filter notIde (map  $\Lambda.un-App2$  (u # U)))) = []  $\vee$ 
     $\Lambda.sseq$  (last (map ( $\lambda X. X \circ \Lambda.Src N$ )
      (stdz-insert M (filter notIde (map  $\Lambda.un-App1$  (u # U))))))
      (hd (map ( $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))))
        (stdz-insert N (filter notIde (map  $\Lambda.un-App2$  (u # U))))))
    proof -
      have  $\Lambda.sseq$  (last (map ( $\lambda X. X \circ \Lambda.Src N$ )
        (stdz-insert M (filter notIde (map  $\Lambda.un-App1$  (u # U))))))
        (hd (map ( $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))))
          (stdz-insert N (filter notIde (map  $\Lambda.un-App2$  (u # U))))))
    proof -
      let ?M =  $\Lambda.un-App1$  (last (map ( $\lambda X. X \circ \Lambda.Src N$ )

```

```

      (stdz-insert M
        (filter notIde
          (map  $\Lambda.un-App1$  (u # U))))))
let ?M' =  $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U)))
let ?N =  $\Lambda.Src$  N
let ?N' =  $\Lambda.un-App2$ 
      (hd (map ( $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))))
        (stdz-insert N
          (filter notIde
            (map  $\Lambda.un-App2$  (u # U))))))
have M: ?M = last (stdz-insert M
  (filter notIde (map  $\Lambda.un-App1$  (u # U))))
  by (metis * A Ide.simps(1) Resid.simps(1) ide-char
     $\Lambda.lambda.sel(3)$  last-map)
have N': ?N' = hd (stdz-insert N
  (filter notIde (map  $\Lambda.un-App2$  (u # U))))
  by (metis ** B Ide.simps(1) Resid.simps(2) ide-char
     $\Lambda.lambda.sel(4)$  hd-map)
have AppMN: last (map ( $\lambda X. X \circ \Lambda.Src$  N)
  (stdz-insert M
    (filter notIde (map  $\Lambda.un-App1$  (u # U))))) =
  ?M  $\circ$  ?N
  by (metis * A Ide.simps(1) M Resid.simps(2) ide-char last-map)
moreover
have 4: hd (map ( $\Lambda.App$  ( $\Lambda.Trq$  ( $\Lambda.un-App1$  (last (u # U))))))
  (stdz-insert N
    (filter notIde (map  $\Lambda.un-App2$  (u # U))))) =
  ?M'  $\circ$  ?N'
  by (metis (no-types, lifting) ** B Resid.simps(2) con-char
    prfx-implies-con  $\Lambda.lambda.collapse(3)$   $\Lambda.lambda.discI(3)$ 
     $\Lambda.lambda.inject(3)$  list.map-sel(1))
moreover have MM:  $\Lambda.elementary-reduction$  ?M
  by (metis * A Arr.simps(1) Con-implies-Arr(2) Ide.simps(1)
    M ide-char in-mono last-in-set mem-Collect-eq)
moreover have NN':  $\Lambda.elementary-reduction$  ?N'
  using ** B N'
  by (metis Arr.simps(1) Con-implies-Arr(2) Ide.simps(1)
    ide-char in-mono list.set-sel(1) mem-Collect-eq)
moreover have  $\Lambda.Trq$  ?M = ?M'
proof –
  have 1: [ $\Lambda.Trq$  ?M] * $\sim$ * [?M']
proof –
  have [ $\Lambda.Trq$  ?M] * $\sim$ *
    [ $\Lambda.Trq$  (last (M # filter notIde (map  $\Lambda.un-App1$  (u # U)))))]
proof –
  have targets (stdz-insert M
    (filter notIde (map  $\Lambda.un-App1$  (u # U)))) =
    targets (M # filter notIde (map  $\Lambda.un-App1$  (u # U)))
  using * A cong-implies-coterminal by blast

```

**moreover**  
**have**  $[\Lambda. \text{Trg} (\text{last} (M \# \text{filter notIde} (\text{map } \Lambda. \text{un-App1} (u \# U))))]$   
 $\in \text{targets} (M \# \text{filter notIde} (\text{map } \Lambda. \text{un-App1} (u \# U)))$   
**by** (*metis* (*no-types*, *lifting*) \* *A*  $\Lambda. \text{Arr-Trg } \Lambda. \text{Ide-Trg}$   
 $\text{Arr.simps}(2)$   $\text{Arr-append-iff}_P$   $\text{Arr-iff-Con-self}$   
 $\text{Con-implies-Arr}(2)$   $\text{Ide.simps}(1)$   $\text{Ide.simps}(2)$   
 $\text{Resid-Arr-Ide-ind}$  *ide-char* *append-butlast-last-id*  
 $\text{append-self-conv2}$   $\Lambda. \text{arr-char}$  *in-targets-iff*  $\Lambda. \text{ide-char}$   
*list.discI*)  
**ultimately show** *?thesis*  
**using** \* *A* *M* *in-targets-iff*  
**by** (*metis* (*no-types*, *lifting*)  $\text{Con-implies-Arr}(1)$   
*con-char* *prfx-implies-con* *in-targets-iff*)  
**qed**  
**also have**  $2: [\Lambda. \text{Trg} (\text{last} (M \# \text{filter notIde}$   
 $(\text{map } \Lambda. \text{un-App1} (u \# U))))] \sim^*$   
 $[\Lambda. \text{Trg} (\text{last} (\text{filter notIde}$   
 $(\text{map } \Lambda. \text{un-App1} (u \# U))))]$   
**by** (*metis* (*no-types*, *lifting*) \* *prfx-transitive*  
*calculation* *empty-filter-conv* *last-ConsR* *list.set-map*  
*mem-Collect-eq* *subsetI*)  
**also have**  $[\Lambda. \text{Trg} (\text{last} (\text{filter notIde}$   
 $(\text{map } \Lambda. \text{un-App1} (u \# U))))] \sim^*$   
 $[\Lambda. \text{Trg} (\text{last} (\text{map } \Lambda. \text{un-App1} (u \# U)))]$   
**proof** –  
**have**  $\text{map } \Lambda. \text{un-App1} (u \# U) \sim^*$   
 $\text{filter notIde} (\text{map } \Lambda. \text{un-App1} (u \# U))$   
**by** (*metis* (*mono-tags*, *lifting*) \* \*\*\*  $\text{Arr-map-un-App1}$   
 $\text{Std Std-imp-Arr}$  *cong-filter-notIde* *empty-filter-conv*  
 $\text{filter-notIde-Ide}$  *insert-subset* *list.discI* *list.set-map*  
 $\text{list.simps}(15)$  *mem-Collect-eq* *subsetI* *u*)  
**thus** *?thesis*  
**by** (*metis*  $2$   $\text{Trg-last-eqI}$  *prfx-transitive*)  
**qed**  
**also have**  $[\Lambda. \text{Trg} (\text{last} (\text{map } \Lambda. \text{un-App1} (u \# U)))] = [?M]$   
**by** (*simp* *add: last-map*)  
**finally show** *?thesis* **by** *blast*  
**qed**  
**have**  $3: \Lambda. \text{Trg } ?M = \Lambda. \text{Trg } ?M \setminus ?M'$   
**by** (*metis* (*no-types*, *lifting*)  $1$  \* *A* *M*  $\text{Con-implies-Arr}(2)$   
 $\text{Ide.simps}(1)$   $\text{Resid-Arr-Ide-ind}$   $\text{Resid-rec}(1)$   
*ide-char* *target-is-ide* *in-targets-iff* *list.inject*)  
**also have**  $\dots = ?M'$   
**by** (*metis* (*no-types*, *lifting*)  $1$   $4$   $\text{Arr.simps}(2)$   $\text{Con-implies-Arr}(2)$   
 $\text{Ide.simps}(1)$   $\text{Ide.simps}(2)$   $\text{MM NN}'$   $\text{Resid-Arr-Ide-ind}$   
 $\text{Resid-rec}(1)$   $\text{Src-hd-eqI}$  *calculation* *ide-char*  
 $\Lambda. \text{Ide-iff-Src-self}$   $\Lambda. \text{Src-Trg}$   $\Lambda. \text{arr-char}$   
 $\Lambda. \text{elementary-reduction.simps}(4)$   
 $\Lambda. \text{elementary-reduction-App-iff}$   $\Lambda. \text{elementary-reduction-is-arr}$ )

```

       $\Lambda$ .elementary-reduction-not-ide  $\Lambda$ .lambda.discI(3)
       $\Lambda$ .lambda.sel(3) list.sel(1))
    finally show ?thesis by blast
  qed
  moreover have ?N =  $\Lambda$ .Src ?N'
  proof -
    have 1: [ $\Lambda$ .Src ?N'] *~* [?N]
    proof -
      have sources (stdz-insert N
        (filter notIde (map  $\Lambda$ .un-App2 (u # U)))) =
        sources [N]
      using ** B
      by (metis Con-implies-Arr(2) Ide.simps(1) coinitalE
        cong-implies-coinital ide-char sources-cons)
    thus ?thesis
    by (metis (no-types, lifting) AppMN ** B  $\Lambda$ .Ide-Src
      MM MN N' NN'  $\Lambda$ .Trg-Src Arr.simps(1) Arr.simps(2)
      Con-implies-Arr(1) Ide.simps(2) con-char ideE ide-char
      sources-cons  $\Lambda$ .arr-char in-targets-iff
       $\Lambda$ .elementary-reduction.simps(4)  $\Lambda$ .elementary-reduction-App-iff
       $\Lambda$ .elementary-reduction-is-arr  $\Lambda$ .elementary-reduction-not-ide
       $\Lambda$ .lambda.disc(14)  $\Lambda$ .lambda.sel(4) last-ConsL list.exhaust-sel
      targets-single-Src)
  qed
  have  $\Lambda$ .Src ?N' =  $\Lambda$ .Src ?N' \ ?N
    by (metis (no-types, lifting) 1 MN  $\Lambda$ .Coinital-iff-Con
       $\Lambda$ .Ide-Src Arr.simps(2) Ide.simps(1) Ide-implies-Arr
      Resid-rec(1) ide-char  $\Lambda$ .not-arr-null  $\Lambda$ .null-char
       $\Lambda$ .resid-Arr-Ide)
  also have ... = ?N
    by (metis 1 MN NN' Src-hd-eqI calculation  $\Lambda$ .Src-Src  $\Lambda$ .arr-char
       $\Lambda$ .elementary-reduction-is-arr list.sel(1))
  finally show ?thesis by simp
  qed
  ultimately show ?thesis
    using u  $\Lambda$ .sseq.simps(4)
    by (metis (mono-tags, lifting))
  qed
  thus ?thesis by blast
  qed
  qed
  thus ?thesis
    using 4 by presburger
  qed
  show  $\neg$  Ide ((M  $\circ$  N) # u # U)  $\longrightarrow$ 
    stdz-insert (M  $\circ$  N) (u # U) *~* (M  $\circ$  N) # u # U
  proof
    have stdz-insert (M  $\circ$  N) (u # U) =
      map ( $\lambda$ X. X  $\circ$   $\Lambda$ .Src N)

```

```

      (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) @
      map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U)))))
      (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))
using 4 by simp
also have ... *~* map (λX. X o Λ.Src N)
      (M # map Λ.un-App1 (u # U)) @
      map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U)))))
      (N # map Λ.un-App2 (u # U))
proof (intro cong-append)
have X: stdz-insert M (filter notIde (map Λ.un-App1 (u # U))) *~*
      M # map Λ.un-App1 (u # U)
proof –
have stdz-insert M (filter notIde (map Λ.un-App1 (u # U))) *~*
      [M] @ filter notIde (map Λ.un-App1 (u # U))
using * A by simp
also have [M] @ filter notIde (map Λ.un-App1 (u # U)) *~*
      [M] @ map Λ.un-App1 (u # U)
proof –
have filter notIde (map Λ.un-App1 (u # U)) *~*
      map Λ.un-App1 (u # U)
using * cong-filter-notIde
by (metis (mono-tags, lifting) *** Arr-map-un-App1 Std
      Std-imp-Arr empty-filter-conv filter-notIde-Ide insert-subset
      list.discI list.set-map list.simps(15) mem-Collect-eq subsetI u)
moreover have seq [M] (filter notIde (map Λ.un-App1 (u # U)))
by (metis * A Arr.simps(1) Con-implies-Arr(1) append-Cons
      append-Nil arr-append-imp-seq arr-char calculation
      ide-implies-arr list.discI)
ultimately show ?thesis
using cong-append cong-reflexive by blast
qed
also have [M] @ map Λ.un-App1 (u # U) =
      M # map Λ.un-App1 (u # U)
by simp
finally show ?thesis by blast
qed
have Y: stdz-insert N (filter notIde (map Λ.un-App2 (u # U))) *~*
      N # map Λ.un-App2 (u # U)
proof –
have 5: stdz-insert N (filter notIde (map Λ.un-App2 (u # U))) *~*
      [N] @ filter notIde (map Λ.un-App2 (u # U))
using ** B by simp
also have [N] @ filter notIde (map Λ.un-App2 (u # U)) *~*
      [N] @ map Λ.un-App2 (u # U)
proof –
have filter notIde (map Λ.un-App2 (u # U)) *~*
      map Λ.un-App2 (u # U)
using ** cong-filter-notIde
by (metis (mono-tags, lifting) *** Arr-map-un-App2 Std

```

```

      Std-imp-Arr empty-filter-conv filter-notIde-Ide insert-subset
      list.discI list.set-map list.simps(15) mem-Collect-eq subsetI u)
moreover have seq [N] (filter notIde (map  $\Lambda$ .un-App2 (u # U)))
  by (metis 5 Arr.simps(1) Con-implies-Arr(2) Ide.simps(1)
      arr-append-imp-seq arr-char calculation ide-char not-Cons-self2)
ultimately show ?thesis
  using cong-append cong-reflexive by blast
qed
also have [N] @ map  $\Lambda$ .un-App2 (u # U) =
      N # map  $\Lambda$ .un-App2 (u # U)
  by simp
finally show ?thesis by blast
qed
show seq (map ( $\lambda X$ . X  $\circ$   $\Lambda$ .Src N)
      (stdz-insert M (filter notIde (map  $\Lambda$ .un-App1 (u # U))))
      (map ( $\Lambda$ .App ( $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))))
      (stdz-insert N (filter notIde (map  $\Lambda$ .un-App2 (u # U))))
  by (metis 4 * ** A B Ide.simps(1) Nil-is-append-conv Nil-is-map-conv
      Resid.simps(1) Std-imp-Arr  $\langle$ Std (stdz-insert (M  $\circ$  N) (u # U)) $\rangle$ 
      arr-append-imp-seq arr-char ide-char)
show map ( $\lambda X$ . X  $\circ$   $\Lambda$ .Src N)
      (stdz-insert M (filter notIde (map  $\Lambda$ .un-App1 (u # U)))) *~*
      map ( $\lambda X$ . X  $\circ$   $\Lambda$ .Src N) (M # map  $\Lambda$ .un-App1 (u # U))
  using X cong-map-App2 MN lambda-calculus.Ide-Src by presburger
show map ( $\Lambda$ .App ( $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))))
      (stdz-insert N (filter notIde (map  $\Lambda$ .un-App2 (u # U)))) *~*
      map ( $\Lambda$ .App ( $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))))
      (N # map  $\Lambda$ .un-App2 (u # U))
proof –
  have set U  $\subseteq$  Collect  $\Lambda$ .Arr  $\cap$  Collect  $\Lambda$ .is-App
  using *** Std Std-implies-set-subset-elementary-reduction
       $\Lambda$ .elementary-reduction-is-arr
  by blast
  hence  $\Lambda$ .Ide ( $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U))))
  by (metis inf.boundedE  $\Lambda$ .Arr.simps(4)  $\Lambda$ .Ide-Trg
       $\Lambda$ .lambda.collapse(3) last.simps last-in-set mem-Collect-eq
      subset-eq u)
  thus ?thesis
  using Y cong-map-App1 by blast
qed
qed
also have map ( $\lambda X$ . X  $\circ$   $\Lambda$ .Src N) (M # map  $\Lambda$ .un-App1 (u # U)) @
      map ( $\Lambda$ .App ( $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))))
      (N # map  $\Lambda$ .un-App2 (u # U)) *~*
      [M  $\circ$  N] @ [u] @ U
proof –
  have (map ( $\lambda X$ . X  $\circ$   $\Lambda$ .Src N) (M # map  $\Lambda$ .un-App1 (u # U)) @
      map ( $\Lambda$ .App ( $\Lambda$ .Trg ( $\Lambda$ .un-App1 (last (u # U)))))
      (N # map  $\Lambda$ .un-App2 (u # U))) =

```

```

      ([M ◦ Λ.Src N] @
       map (λX. X ◦ Λ.Src N) (map Λ.un-App1 (u # U))) @
      ([Λ.Trig (Λ.un-App1 (last (u # U))) ◦ N] @
       map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))
              (map Λ.un-App2 (u # U))))
    by simp
  also have ... = [M ◦ Λ.Src N] @
    (map (λX. X ◦ Λ.Src N) (map Λ.un-App1 (u # U))) @
    map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U)))) [N]) @
    map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))
          (map Λ.un-App2 (u # U)))

  by auto
  also have ... *~* [M ◦ Λ.Src N] @
    (map (Λ.App (Λ.Src (Λ.un-App1 u))) [N] @
     map (λX. X ◦ Λ.Trig N) (map Λ.un-App1 (u # U))) @
    map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))
          (map Λ.un-App2 (u # U)))

  proof –

  have (map (λX. X ◦ Λ.Src N) (map Λ.un-App1 (u # U))) @
    map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U)))) [N]) @
    map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))
          (map Λ.un-App2 (u # U))) *~*
    (map (Λ.App (Λ.Src (Λ.un-App1 u))) [N] @
     map (λX. X ◦ Λ.Trig N) (map Λ.un-App1 (u # U))) @
    map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))
          (map Λ.un-App2 (u # U)))

  proof –
  have 1: Arr (map Λ.un-App1 (u # U))
    using u ***
  by (metis Arr-map-un-App1 Std Std-imp-Arr list.discI
      mem-Collect-eq set-ConsD subset-code(1))
  have map (λX. Λ.App X (Λ.Src N)) (map Λ.un-App1 (u # U)) @
    map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U)))) [N]) *~*
    map (Λ.App (Λ.Src (Λ.un-App1 u))) [N] @
    map (λX. Λ.App X (Λ.Trig N)) (map Λ.un-App1 (u # U))

  proof –
  have Arr [N]
    using MN by simp
  moreover have Λ.Trig (last (map Λ.un-App1 (u # U))) =
    Λ.Trig (Λ.un-App1 (last (u # U)))
  by (simp add: last-map)
  ultimately show ?thesis
    using 1 orthogonal-App-cong [of map Λ.un-App1 (u # U) [N]]
    by simp
  qed
  moreover have seq (map (λX. X ◦ Λ.Src N) (map Λ.un-App1 (u #
    U))) @
    map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U)))) [N])

```

```

      (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))))
      (map Λ.un-App2 (u # U)))
proof
show Arr (map (λX. X ∘ Λ.Src N)
  (map Λ.un-App1 (u # U)) @
  map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))) [N])
  by (metis Con-implies-Arr(1) Ide.simps(1) calculation ide-char)
show Arr (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))))
  (map Λ.un-App2 (u # U)))
  using u ***
  by (metis 1 Arr-imp-arr-last Arr-map-App2 Arr-map-un-App2
    Std Std-imp-Arr Λ.Ide-Trg Λ.arr-char last-map list.discI
    mem-Collect-eq set-ConsD subset-code(1))
show Λ.Trg (last (map (λX. X ∘ Λ.Src N)
  (map Λ.un-App1 (u # U)) @
  map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))) [N]))) =
  Λ.Src (hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))))
  (map Λ.un-App2 (u # U))))
proof –
  have 1: Λ.Arr (Λ.un-App1 u)
  using u Λ.is-App-def by force
  have 2: U ≠ [] ⇒ Λ.Arr (Λ.un-App1 (last U))
  by (metis *** Arr-imp-arr-last Arr-map-un-App1
    ⟨U ≠ [] ⇒ Arr U⟩ Λ.arr-char last-map)
  have 3: Λ.Trg N = Λ.Src (Λ.un-App2 u)
  by (metis Trg-last-Src-hd-eqI Λ.Src.simps(4) Λ.Trg.simps(3)
    Λ.lambda.collapse(3) Λ.lambda.inject(3) last-ConsL
    list.sel(1) seq u)
  show ?thesis
  using u *** seq 1 2 3
  by (cases U = []) auto
qed
qed
moreover have map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))))
  (map Λ.un-App2 (u # U)) *~*
  map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))))
  (map Λ.un-App2 (u # U))
  using calculation(2) cong-reflexive by blast
  ultimately show ?thesis
  using cong-append by blast
qed
moreover have seq [M ∘ Λ.Src N]
  ((map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U)) @
  map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))) [N]) @
  map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))))
  (map Λ.un-App2 (u # U)))
proof
show Arr [M ∘ Λ.Src N]

```

```

    using MN by simp
  show Arr ((map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U))) @
            map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U)))))) [N]) @
            map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))))
            (map Λ.un-App2 (u # U)))
    using MN u seq
  by (metis Con-implies-Arr(1) Ide.simps(1) calculation ide-char)
  show Λ.Trig (last [M ∘ Λ.Src N]) =
    Λ.Src (hd ((map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U))) @
              map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U)))))) [N]) @
          map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))))
          (map Λ.un-App2 (u # U)))
    using MN u seq seq-char Srcs-simpΛP
  by (cases u) auto
qed
ultimately show ?thesis
  using cong-append
  by (meson Resid-Arr-self ide-char seq-char)
qed
also have [M ∘ Λ.Src N] @
  (map (Λ.App (Λ.Src (Λ.un-App1 u))) [N]) @
  map (λX. Λ.App X (Λ.Trig N)) (map Λ.un-App1 (u # U))) @
  map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))))
  (map Λ.un-App2 (u # U)) =
  ([M ∘ Λ.Src N] @ [Λ.Src (Λ.un-App1 u) ∘ N]) @
  (map (λX. X ∘ Λ.Trig N) (map Λ.un-App1 (u # U))) @
  map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))))
  (map Λ.un-App2 (u # U))

  by simp
also have ... *~* ([M ∘ N] @ [u] @ U)
proof -
  have [M ∘ Λ.Src N] @ [Λ.Src (Λ.un-App1 u) ∘ N] *~* [M ∘ N]
  proof -
    have Λ.Src (Λ.un-App1 u) = Λ.Trig M
    by (metis Trig-last-Src-hd-eqI Λ.Src.simps(4) Λ.Trig.simps(3)
        Λ.lambda.collapse(3) Λ.lambda.inject(3) last.simps
        list.sel(1) seq u)
    thus ?thesis
    using MN u seq seq-char Λ.Arr-not-Nil Λ.resid-Arr-self ide-char
        Λ.Ide-Trig
    by simp
  qed
moreover have map (λX. X ∘ Λ.Trig N) (map Λ.un-App1 (u # U)) @
  map (Λ.App (Λ.Trig (Λ.un-App1 (last (u # U))))))
  (map Λ.un-App2 (u # U)) *~*
  [u] @ U
proof -
  have Arr ([u] @ U)
  by (simp add: Std)

```

```

moreover have  $set ([u] @ U) \subseteq Collect \Lambda.is-App$ 
using *** u by auto
moreover have  $\Lambda.Src (\Lambda.un-App2 (hd ([u] @ U))) = \Lambda.Trig N$ 
proof –
  have  $\Lambda.Ide (\Lambda.Trig N)$ 
    using MN lambda-calculus.Ide-Trig by presburger
  moreover have  $\Lambda.Ide (\Lambda.Src (\Lambda.un-App2 (hd ([u] @ U))))$ 
    by (metis Std Std-implies-set-subset-elementary-reduction
       $\Lambda.Ide-Src \Lambda.arr-iff-has-source \Lambda.ide-implies-arr$ 
       $\langle set ([u] @ U) \subseteq Collect \Lambda.is-App \rangle$  append-Cons
       $\Lambda.elementary-reduction-App-iff \Lambda.elementary-reduction-is-arr$ 
       $\Lambda.sources-char_{\Lambda} list.sel(1) list.set-intros(1)$ 
      mem-Collect-eq subset-code(1))
  moreover have  $\Lambda.Src (\Lambda.Trig N) =$ 
     $\Lambda.Src (\Lambda.Src (\Lambda.un-App2 (hd ([u] @ U))))$ 
proof –
  have  $\Lambda.Src (\Lambda.Trig N) = \Lambda.Trig N$ 
    using MN by simp
  also have  $... = \Lambda.Src (\Lambda.un-App2 u)$ 
    using u seq seq-char Srcs-simp $_{\Lambda P}$ 
    by (cases u) auto
  also have  $... = \Lambda.Src (\Lambda.Src (\Lambda.un-App2 (hd ([u] @ U))))$ 
    by (metis  $\Lambda.Ide-iff-Src-self \Lambda.Ide-implies-Arr$ 
       $\langle \Lambda.Ide (\Lambda.Src (\Lambda.un-App2 (hd ([u] @ U)))) \rangle$ 
      append-Cons list.sel(1))
  finally show ?thesis by blast
qed
ultimately show ?thesis
  by (metis  $\Lambda.Ide-iff-Src-self \Lambda.Ide-implies-Arr$ )
qed
ultimately show ?thesis
  using map-App-decomp
  by (metis append-Cons append-Nil)
qed
moreover have  $seq ([M \circ \Lambda.Src N] @ [\Lambda.Src (\Lambda.un-App1 u) \circ N])$ 
   $(map (\lambda X. X \circ \Lambda.Trig N) (map \Lambda.un-App1 (u \# U))) @$ 
   $map (\Lambda.App (\Lambda.Trig (\Lambda.un-App1 (last (u \# U))))$ 
   $(map \Lambda.un-App2 (u \# U)))$ 
  using calculation(1–2) cong-respects-seq $_P$  seq by auto
ultimately show ?thesis
  using cong-append by presburger
qed
finally show ?thesis by blast
qed
also have  $[M \circ N] @ [u] @ U = (M \circ N) \# u \# U$ 
  by simp
finally show stdz-insert (M \circ N) (u \# U) *~* (M \circ N) \# u \# U
  by blast
qed

```

qed  
 qed  
 qed  
 qed  
 qed  
 qed  
 qed  
 qed  
 qed

The eight remaining subgoals are now trivial consequences of fact \*. Unfortunately, I haven't found a way to discharge them without having to state each one of them explicitly.

```

show  $\bigwedge N u U. [\Lambda.Ide (\# \circ N) \implies ?P (hd (u \# U)) (tl (u \# U))];$ 
   $[\neg \Lambda.Ide (\# \circ N); \Lambda.seq (\# \circ N) (hd (u \# U));$ 
     $\Lambda.contains-head-reduction (\# \circ N);$ 
     $\Lambda.Ide ((\# \circ N) \setminus \Lambda.head-redex (\# \circ N))]$ 
     $\implies ?P (hd (u \# U)) (tl (u \# U));$ 
   $[\neg \Lambda.Ide (\# \circ N); \Lambda.seq (\# \circ N) (hd (u \# U));$ 
     $\Lambda.contains-head-reduction (\# \circ N);$ 
     $\neg \Lambda.Ide ((\# \circ N) \setminus \Lambda.head-redex (\# \circ N))]$ 
     $\implies ?P ((\# \circ N) \setminus \Lambda.head-redex (\# \circ N)) (u \# U);$ 
   $[\neg \Lambda.Ide (\# \circ N); \Lambda.seq (\# \circ N) (hd (u \# U));$ 
     $\neg \Lambda.contains-head-reduction (\# \circ N);$ 
     $\Lambda.contains-head-reduction (hd (u \# U));$ 
     $\Lambda.Ide ((\# \circ N) \setminus \Lambda.head-strategy (\# \circ N))]$ 
     $\implies ?P (\Lambda.head-strategy (\# \circ N)) (tl (u \# U));$ 
   $[\neg \Lambda.Ide (\# \circ N); \Lambda.seq (\# \circ N) (hd (u \# U));$ 
     $\neg \Lambda.contains-head-reduction (\# \circ N);$ 
     $\Lambda.contains-head-reduction (hd (u \# U));$ 
     $\neg \Lambda.Ide ((\# \circ N) \setminus \Lambda.head-strategy (\# \circ N))]$ 
     $\implies ?P (\Lambda.resid (\# \circ N) (\Lambda.head-strategy (\# \circ N))) (tl (u \# U));$ 
   $[\neg \Lambda.Ide (\# \circ N); \Lambda.seq (\# \circ N) (hd (u \# U));$ 
     $\neg \Lambda.contains-head-reduction (\# \circ N);$ 
     $\neg \Lambda.contains-head-reduction (hd (u \# U))]$ 
     $\implies ?P \# (filter notIde (map \Lambda.un-App1 (u \# U)));$ 
   $[\neg \Lambda.Ide (\# \circ N); \Lambda.seq (\# \circ N) (hd (u \# U));$ 
     $\neg \Lambda.contains-head-reduction (\# \circ N);$ 
     $\neg \Lambda.contains-head-reduction (hd (u \# U))]$ 
     $\implies ?P N (filter notIde (map \Lambda.un-App2 (u \# U)))$ 
     $\implies ?P (\# \circ N) (u \# U)$ 

using *  $\Lambda.lambda.disc(6)$  by presburger
show  $\bigwedge x N u U. [\Lambda.Ide (\langle x \rangle \circ N) \implies ?P (hd (u \# U)) (tl (u \# U))];$ 
   $[\neg \Lambda.Ide (\langle x \rangle \circ N); \Lambda.seq (\langle x \rangle \circ N) (hd (u \# U));$ 
     $\Lambda.contains-head-reduction (\langle x \rangle \circ N);$ 
     $\Lambda.Ide ((\langle x \rangle \circ N) \setminus \Lambda.head-redex (\langle x \rangle \circ N))]$ 
     $\implies ?P (hd (u \# U)) (tl (u \# U));$ 
   $[\neg \Lambda.Ide (\langle x \rangle \circ N); \Lambda.seq (\langle x \rangle \circ N) (hd (u \# U));$ 
     $\Lambda.contains-head-reduction (\langle x \rangle \circ N);$ 

```

$$\begin{aligned}
& \neg \Lambda.\text{Ide } ((\langle x \rangle \circ N) \setminus \Lambda.\text{head-redex } (\langle x \rangle \circ N)) \\
& \implies ?P ((\langle x \rangle \circ N) \setminus \Lambda.\text{head-redex } (\langle x \rangle \circ N)) (u \# U); \\
& \llbracket \neg \Lambda.\text{Ide } (\langle x \rangle \circ N); \Lambda.\text{seq } (\langle x \rangle \circ N) (\text{hd } (u \# U)); \\
& \neg \Lambda.\text{contains-head-reduction } (\langle x \rangle \circ N); \\
& \Lambda.\text{contains-head-reduction } (\text{hd } (u \# U)); \\
& \Lambda.\text{Ide } ((\langle x \rangle \circ N) \setminus \Lambda.\text{head-strategy } (\langle x \rangle \circ N)) \rrbracket \\
& \implies ?P (\Lambda.\text{head-strategy } (\langle x \rangle \circ N)) (\text{tl } (u \# U)); \\
& \llbracket \neg \Lambda.\text{Ide } (\langle x \rangle \circ N); \Lambda.\text{seq } (\langle x \rangle \circ N) (\text{hd } (u \# U)); \\
& \neg \Lambda.\text{contains-head-reduction } (\langle x \rangle \circ N); \\
& \Lambda.\text{contains-head-reduction } (\text{hd } (u \# U)); \\
& \neg \Lambda.\text{Ide } ((\langle x \rangle \circ N) \setminus \Lambda.\text{head-strategy } (\langle x \rangle \circ N)) \rrbracket \\
& \implies ?P ((\langle x \rangle \circ N) \setminus \Lambda.\text{head-strategy } (\langle x \rangle \circ N)) (\text{tl } (u \# U)); \\
& \llbracket \neg \Lambda.\text{Ide } (\langle x \rangle \circ N); \Lambda.\text{seq } (\langle x \rangle \circ N) (\text{hd } (u \# U)); \\
& \neg \Lambda.\text{contains-head-reduction } (\langle x \rangle \circ N); \\
& \neg \Lambda.\text{contains-head-reduction } (\text{hd } (u \# U)) \rrbracket \\
& \implies ?P \langle x \rangle (\text{filter notIde } (\text{map } \Lambda.\text{un-App1 } (u \# U))); \\
& \llbracket \neg \Lambda.\text{Ide } (\langle x \rangle \circ N); \Lambda.\text{seq } (\langle x \rangle \circ N) (\text{hd } (u \# U)); \\
& \neg \Lambda.\text{contains-head-reduction } (\langle x \rangle \circ N); \\
& \neg \Lambda.\text{contains-head-reduction } (\text{hd } (u \# U)) \rrbracket \\
& \implies ?P N (\text{filter notIde } (\text{map } \Lambda.\text{un-App2 } (u \# U))) \\
& \implies ?P (\langle x \rangle \circ N) (u \# U) \\
& \text{using } * \Lambda.\text{lambda.disc}(7) \text{ by } \text{presburger} \\
& \text{show } \bigwedge M1 M2 N u U. \llbracket \Lambda.\text{Ide } (M1 \circ M2 \circ N) \implies ?P (\text{hd } (u \# U)) (\text{tl } (u \# U)); \\
& \llbracket \neg \Lambda.\text{Ide } (M1 \circ M2 \circ N); \Lambda.\text{seq } (M1 \circ M2 \circ N) (\text{hd } (u \# U)); \\
& \Lambda.\text{contains-head-reduction } (M1 \circ M2 \circ N); \\
& \Lambda.\text{Ide } ((M1 \circ M2 \circ N) \setminus \Lambda.\text{head-redex } (M1 \circ M2 \circ N)) \rrbracket \\
& \implies ?P (\text{hd } (u \# U)) (\text{tl } (u \# U)); \\
& \llbracket \neg \Lambda.\text{Ide } (M1 \circ M2 \circ N); \Lambda.\text{seq } (M1 \circ M2 \circ N) (\text{hd } (u \# U)); \\
& \Lambda.\text{contains-head-reduction } (M1 \circ M2 \circ N); \\
& \neg \Lambda.\text{Ide } ((M1 \circ M2 \circ N) \setminus \Lambda.\text{head-redex } (M1 \circ M2 \circ N)) \rrbracket \\
& \implies ?P ((M1 \circ M2 \circ N) \setminus \Lambda.\text{head-redex } (M1 \circ M2 \circ N)) (u \# U); \\
& \llbracket \neg \Lambda.\text{Ide } (M1 \circ M2 \circ N); \Lambda.\text{seq } (M1 \circ M2 \circ N) (\text{hd } (u \# U)); \\
& \neg \Lambda.\text{contains-head-reduction } (M1 \circ M2 \circ N); \\
& \Lambda.\text{contains-head-reduction } (\text{hd } (u \# U)); \\
& \Lambda.\text{Ide } ((M1 \circ M2 \circ N) \setminus \Lambda.\text{head-strategy } (M1 \circ M2 \circ N)) \rrbracket \\
& \implies ?P (\Lambda.\text{head-strategy } (M1 \circ M2 \circ N)) (\text{tl } (u \# U)); \\
& \llbracket \neg \Lambda.\text{Ide } (M1 \circ M2 \circ N); \Lambda.\text{seq } (M1 \circ M2 \circ N) (\text{hd } (u \# U)); \\
& \neg \Lambda.\text{contains-head-reduction } (M1 \circ M2 \circ N); \\
& \Lambda.\text{contains-head-reduction } (\text{hd } (u \# U)); \\
& \neg \Lambda.\text{Ide } ((M1 \circ M2 \circ N) \setminus \Lambda.\text{head-strategy } (M1 \circ M2 \circ N)) \rrbracket \\
& \implies ?P ((M1 \circ M2 \circ N) \setminus \Lambda.\text{head-strategy } (M1 \circ M2 \circ N)) (\text{tl } (u \# U)); \\
& \llbracket \neg \Lambda.\text{Ide } (M1 \circ M2 \circ N); \Lambda.\text{seq } (M1 \circ M2 \circ N) (\text{hd } (u \# U)); \\
& \neg \Lambda.\text{contains-head-reduction } (M1 \circ M2 \circ N); \\
& \neg \Lambda.\text{contains-head-reduction } (\text{hd } (u \# U)) \rrbracket \\
& \implies ?P (M1 \circ M2) (\text{filter notIde } (\text{map } \Lambda.\text{un-App1 } (u \# U))); \\
& \llbracket \neg \Lambda.\text{Ide } (M1 \circ M2 \circ N); \Lambda.\text{seq } (M1 \circ M2 \circ N) (\text{hd } (u \# U)); \\
& \neg \Lambda.\text{contains-head-reduction } (M1 \circ M2 \circ N); \\
& \neg \Lambda.\text{contains-head-reduction } (\text{hd } (u \# U)) \rrbracket \\
& \implies ?P N (\text{filter notIde } (\text{map } \Lambda.\text{un-App2 } (u \# U)))
\end{aligned}$$

$\implies ?P (M1 \circ M2 \circ N) (u \# U)$   
**using** \*  $\Lambda.lambda.disc(9)$  **by** *presburger*  
**show**  $\bigwedge M1 M2 N u U. [\Lambda.Ide (\lambda[M1] \bullet M2 \circ N) \implies ?P (hd (u \# U)) (tl (u \# U));$   
 $[\neg \Lambda.Ide (\lambda[M1] \bullet M2 \circ N); \Lambda.seq (\lambda[M1] \bullet M2 \circ N) (hd (u \# U));$   
 $\Lambda.contains-head-reduction (\lambda[M1] \bullet M2 \circ N);$   
 $\Lambda.Ide ((\lambda[M1] \bullet M2 \circ N) \setminus (\Lambda.head-redex (\lambda[M1] \bullet M2 \circ N)))]$   
 $\implies ?P (hd (u \# U)) (tl (u \# U));$   
 $[\neg \Lambda.Ide (\lambda[M1] \bullet M2 \circ N); \Lambda.seq (\lambda[M1] \bullet M2 \circ N) (hd (u \# U));$   
 $\Lambda.contains-head-reduction (\lambda[M1] \bullet M2 \circ N);$   
 $\neg \Lambda.Ide ((\lambda[M1] \bullet M2 \circ N) \setminus (\Lambda.head-redex (\lambda[M1] \bullet M2 \circ N)))]$   
 $\implies ?P (\Lambda.resid (\lambda[M1] \bullet M2 \circ N) (\Lambda.head-redex (\lambda[M1] \bullet M2 \circ N)))$   
 $(u \# U);$   
 $[\neg \Lambda.Ide (\lambda[M1] \bullet M2 \circ N); \Lambda.seq (\lambda[M1] \bullet M2 \circ N) (hd (u \# U));$   
 $\neg \Lambda.contains-head-reduction (\lambda[M1] \bullet M2 \circ N);$   
 $\Lambda.contains-head-reduction (hd (u \# U));$   
 $\Lambda.Ide ((\lambda[M1] \bullet M2 \circ N) \setminus \Lambda.head-strategy (\lambda[M1] \bullet M2 \circ N))]$   
 $\implies ?P (\Lambda.head-strategy (\lambda[M1] \bullet M2 \circ N)) (tl (u \# U));$   
 $[\neg \Lambda.Ide (\lambda[M1] \bullet M2 \circ N); \Lambda.seq (\lambda[M1] \bullet M2 \circ N) (hd (u \# U));$   
 $\neg \Lambda.contains-head-reduction (\lambda[M1] \bullet M2 \circ N);$   
 $\Lambda.contains-head-reduction (hd (u \# U));$   
 $\neg \Lambda.Ide ((\lambda[M1] \bullet M2 \circ N) \setminus \Lambda.head-strategy (\lambda[M1] \bullet M2 \circ N))]$   
 $\implies ?P ((\lambda[M1] \bullet M2 \circ N) \setminus \Lambda.head-strategy (\lambda[M1] \bullet M2 \circ N))$   
 $(tl (u \# U));$   
 $[\neg \Lambda.Ide (\lambda[M1] \bullet M2 \circ N); \Lambda.seq (\lambda[M1] \bullet M2 \circ N) (hd (u \# U));$   
 $\neg \Lambda.contains-head-reduction (\lambda[M1] \bullet M2 \circ N);$   
 $\neg \Lambda.contains-head-reduction (hd (u \# U))]$   
 $\implies ?P (\lambda[M1] \bullet M2) (filter notIde (map \Lambda.un-App1 (u \# U)));$   
 $[\neg \Lambda.Ide (\lambda[M1] \bullet M2 \circ N); \Lambda.seq (\lambda[M1] \bullet M2 \circ N) (hd (u \# U));$   
 $\neg \Lambda.contains-head-reduction (\lambda[M1] \bullet M2 \circ N);$   
 $\neg \Lambda.contains-head-reduction (hd (u \# U))]$   
 $\implies ?P N (filter notIde (map \Lambda.un-App2 (u \# U)))]$   
 $\implies ?P (\lambda[M1] \bullet M2 \circ N) (u \# U)$   
**using** \*  $\Lambda.lambda.disc(10)$  **by** *presburger*  
**show**  $\bigwedge M N U. [\Lambda.Ide (M \circ N) \implies ?P (hd (\# \# U)) (tl (\# \# U));$   
 $[\neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\# \# U));$   
 $\Lambda.contains-head-reduction (M \circ N);$   
 $\Lambda.Ide ((M \circ N) \setminus \Lambda.head-redex (M \circ N))]$   
 $\implies ?P (hd (\# \# U)) (tl (\# \# U));$   
 $[\neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\# \# U));$   
 $\Lambda.contains-head-reduction (M \circ N);$   
 $\neg \Lambda.Ide ((M \circ N) \setminus \Lambda.head-redex (M \circ N))]$   
 $\implies ?P ((M \circ N) \setminus \Lambda.head-redex (M \circ N)) (\# \# U);$   
 $[\neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\# \# U));$   
 $\neg \Lambda.contains-head-reduction (M \circ N);$   
 $\Lambda.contains-head-reduction (hd (\# \# U));$   
 $\Lambda.Ide (\Lambda.resid (M \circ N) (\Lambda.head-strategy (M \circ N)))]$   
 $\implies ?P (\Lambda.head-strategy (M \circ N)) (tl (\# \# U));$   
 $[\neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\# \# U));$   
 $\neg \Lambda.contains-head-reduction (M \circ N);$

$\Lambda.contains\text{-head}\text{-reduction} (hd (\# \# U));$   
 $\neg \Lambda.Ide ((M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N))$   
 $\implies ?P ((M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N)) (tl (\# \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\# \# U));$   
 $\neg \Lambda.contains\text{-head}\text{-reduction} (M \circ N);$   
 $\neg \Lambda.contains\text{-head}\text{-reduction} (hd (\# \# U)) \rrbracket$   
 $\implies ?P M (filter\ notIde (map \Lambda.un\text{-App1} (\# \# U)));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\# \# U));$   
 $\neg \Lambda.contains\text{-head}\text{-reduction} (M \circ N);$   
 $\neg \Lambda.contains\text{-head}\text{-reduction} (hd (\# \# U)) \rrbracket$   
 $\implies ?P N (filter\ notIde (map \Lambda.un\text{-App2} (\# \# U)))$   
 $\implies ?P (M \circ N) (\# \# U)$   
**using** \*  $\Lambda.lambda.disc(16)$  **by** *presburger*  
**show**  $\bigwedge M N x U. \llbracket \Lambda.Ide (M \circ N) \implies ?P (hd (\langle x \rangle \# U)) (tl (\langle x \rangle \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\langle x \rangle \# U));$   
 $\Lambda.contains\text{-head}\text{-reduction} (M \circ N);$   
 $\Lambda.Ide ((M \circ N) \setminus \Lambda.head\text{-redex} (M \circ N)) \rrbracket$   
 $\implies ?P (hd (\langle x \rangle \# U)) (tl (\langle x \rangle \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\langle x \rangle \# U));$   
 $\Lambda.contains\text{-head}\text{-reduction} (M \circ N);$   
 $\neg \Lambda.Ide ((M \circ N) \setminus \Lambda.head\text{-redex} (M \circ N)) \rrbracket$   
 $\implies ?P ((M \circ N) \setminus \Lambda.head\text{-redex} (M \circ N)) (\langle x \rangle \# U);$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\langle x \rangle \# U));$   
 $\neg \Lambda.contains\text{-head}\text{-reduction} (M \circ N);$   
 $\Lambda.contains\text{-head}\text{-reduction} (hd (\langle x \rangle \# U));$   
 $\Lambda.Ide ((M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N)) \rrbracket$   
 $\implies ?P (\Lambda.head\text{-strategy} (M \circ N)) (tl (\langle x \rangle \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\langle x \rangle \# U));$   
 $\neg \Lambda.contains\text{-head}\text{-reduction} (M \circ N);$   
 $\Lambda.contains\text{-head}\text{-reduction} (hd (\langle x \rangle \# U));$   
 $\neg \Lambda.Ide ((M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N)) \rrbracket$   
 $\implies ?P ((M \circ N) \setminus \Lambda.head\text{-strategy} (M \circ N)) (tl (\langle x \rangle \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\langle x \rangle \# U));$   
 $\neg \Lambda.contains\text{-head}\text{-reduction} (M \circ N);$   
 $\neg \Lambda.contains\text{-head}\text{-reduction} (hd (\langle x \rangle \# U)) \rrbracket$   
 $\implies ?P M (filter\ notIde (map \Lambda.un\text{-App1} (\langle x \rangle \# U)));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\langle x \rangle \# U));$   
 $\neg \Lambda.contains\text{-head}\text{-reduction} (M \circ N);$   
 $\neg \Lambda.contains\text{-head}\text{-reduction} (hd (\langle x \rangle \# U)) \rrbracket$   
 $\implies ?P N (filter\ notIde (map \Lambda.un\text{-App2} (\langle x \rangle \# U)))$   
 $\implies ?P (M \circ N) (\langle x \rangle \# U)$   
**using** \*  $\Lambda.lambda.disc(17)$  **by** *presburger*  
**show**  $\bigwedge M N P U. \llbracket \Lambda.Ide (M \circ N) \implies ?P (hd (\lambda[P] \# U)) (tl (\lambda[P] \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\lambda[P] \# U));$   
 $\Lambda.contains\text{-head}\text{-reduction} (M \circ N);$   
 $\Lambda.Ide ((M \circ N) \setminus \Lambda.head\text{-redex} (M \circ N)) \rrbracket$   
 $\implies ?P (hd (\lambda[P] \# U)) (tl (\lambda[P] \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\lambda[P] \# U));$   
 $\Lambda.contains\text{-head}\text{-reduction} (M \circ N);$

$\neg \Lambda.Ide ((M \circ N) \setminus \Lambda.head-redex (M \circ N))$   
 $\implies ?P ((M \circ N) \setminus \Lambda.head-redex (M \circ N)) (\lambda[P] \# U);$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\lambda[P] \# U));$   
 $\neg \Lambda.contains-head-reduction (M \circ N);$   
 $\Lambda.contains-head-reduction (hd (\lambda[P] \# U));$   
 $\Lambda.Ide ((M \circ N) \setminus \Lambda.head-strategy (M \circ N)) \rrbracket$   
 $\implies ?P (\Lambda.head-strategy (M \circ N)) (tl (\lambda[P] \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\lambda[P] \# U));$   
 $\neg \Lambda.contains-head-reduction (M \circ N);$   
 $\Lambda.contains-head-reduction (hd (\lambda[P] \# U));$   
 $\neg \Lambda.Ide ((M \circ N) \setminus \Lambda.head-strategy (M \circ N)) \rrbracket$   
 $\implies ?P (\Lambda.resid (M \circ N) (\Lambda.head-strategy (M \circ N))) (tl (\lambda[P] \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\lambda[P] \# U));$   
 $\neg \Lambda.contains-head-reduction (M \circ N);$   
 $\neg \Lambda.contains-head-reduction (hd (\lambda[P] \# U)) \rrbracket$   
 $\implies ?P M (filter notIde (map \Lambda.un-App1 (\lambda[P] \# U)));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd (\lambda[P] \# U));$   
 $\neg \Lambda.contains-head-reduction (M \circ N);$   
 $\neg \Lambda.contains-head-reduction (hd (\lambda[P] \# U)) \rrbracket$   
 $\implies ?P N (filter notIde (map \Lambda.un-App2 (\lambda[P] \# U)))$   
 $\implies ?P (M \circ N) (\lambda[P] \# U)$   
**using** \*  $\Lambda.lambda.disc(18)$  **by** *presburger*  
**show**  $\bigwedge M N P1 P2 U. \llbracket \Lambda.Ide (M \circ N)$   
 $\implies ?P (hd ((P1 \circ P2) \# U)) (tl ((P1 \circ P2) \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd ((P1 \circ P2) \# U));$   
 $\Lambda.contains-head-reduction (M \circ N);$   
 $\Lambda.Ide ((M \circ N) \setminus \Lambda.head-redex (M \circ N)) \rrbracket$   
 $\implies ?P (hd ((P1 \circ P2) \# U)) (tl((P1 \circ P2) \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd ((P1 \circ P2) \# U));$   
 $\Lambda.contains-head-reduction (M \circ N);$   
 $\neg \Lambda.Ide ((M \circ N) \setminus \Lambda.head-redex (M \circ N)) \rrbracket$   
 $\implies ?P ((M \circ N) \setminus \Lambda.head-redex (M \circ N)) ((P1 \circ P2) \# U);$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd ((P1 \circ P2) \# U));$   
 $\neg \Lambda.contains-head-reduction (M \circ N);$   
 $\Lambda.contains-head-reduction (hd ((P1 \circ P2) \# U));$   
 $\Lambda.Ide ((M \circ N) \setminus \Lambda.head-strategy (M \circ N)) \rrbracket$   
 $\implies ?P (\Lambda.head-strategy (M \circ N)) (tl ((P1 \circ P2) \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd ((P1 \circ P2) \# U));$   
 $\neg \Lambda.contains-head-reduction (M \circ N);$   
 $\Lambda.contains-head-reduction (hd ((P1 \circ P2) \# U));$   
 $\neg \Lambda.Ide ((M \circ N) \setminus \Lambda.head-strategy (M \circ N)) \rrbracket$   
 $\implies ?P ((M \circ N) \setminus \Lambda.head-strategy (M \circ N)) (tl ((P1 \circ P2) \# U));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd ((P1 \circ P2) \# U));$   
 $\neg \Lambda.contains-head-reduction (M \circ N);$   
 $\neg \Lambda.contains-head-reduction (hd ((P1 \circ P2) \# U)) \rrbracket$   
 $\implies ?P M (filter notIde (map \Lambda.un-App1 ((P1 \circ P2) \# U)));$   
 $\llbracket \neg \Lambda.Ide (M \circ N); \Lambda.seq (M \circ N) (hd ((P1 \circ P2) \# U));$   
 $\neg \Lambda.contains-head-reduction (M \circ N);$   
 $\neg \Lambda.contains-head-reduction (hd ((P1 \circ P2) \# U)) \rrbracket$

```

    ⇒ ?P N (filter notIde (map Λ.un-App2 ((P1 ∘ P2) # U)))
  ⇒ ?P (M ∘ N) ((P1 ∘ P2) # U)
  using * Λ.lambda.disc(19) by presburger
qed

```

## The Standardization Theorem

Using the function *standardize*, we can now prove the Standardization Theorem. There is still a little bit more work to do, because we have to deal with various cases in which the reduction path to be standardized is empty or consists entirely of identities.

**theorem** *standardization-theorem*:

**shows**  $Arr\ T \implies Std\ (standardize\ T) \wedge (Ide\ T \longrightarrow standardize\ T = []) \wedge$   
 $(\neg\ Ide\ T \longrightarrow cong\ (standardize\ T)\ T)$

**proof** (*induct T*)

**show**  $Arr\ [] \implies Std\ (standardize\ []) \wedge (Ide\ [] \longrightarrow standardize\ [] = []) \wedge$   
 $(\neg\ Ide\ [] \longrightarrow cong\ (standardize\ [])\ [])$

by *simp*

**fix**  $t\ T$

**assume**  $ind: Arr\ T \implies Std\ (standardize\ T) \wedge (Ide\ T \longrightarrow standardize\ T = []) \wedge$   
 $(\neg\ Ide\ T \longrightarrow cong\ (standardize\ T)\ T)$

**assume**  $tT: Arr\ (t\ \# T)$

**have**  $t: \Lambda.Arr\ t$

using  $tT\ Arr\text{-imp-arr-hd}$  by *force*

**show**  $Std\ (standardize\ (t\ \# T)) \wedge (Ide\ (t\ \# T) \longrightarrow standardize\ (t\ \# T) = []) \wedge$   
 $(\neg\ Ide\ (t\ \# T) \longrightarrow cong\ (standardize\ (t\ \# T))\ (t\ \# T))$

**proof** (*cases T = []*)

**show**  $T = [] \implies ?thesis$

using  $t\ T\ Ide\text{-iff-standard-development-empty}\ Std\text{-standard-development}$   
 $cong\text{-standard-development}$

by *simp*

**assume**  $0: T \neq []$

**hence**  $T: Arr\ T$

using  $tT$

by (*metis Arr-imp-Arr-tl list.sel(3)*)

**show**  $?thesis$

**proof** (*intro conjI*)

**show**  $Std\ (standardize\ (t\ \# T))$

**proof** –

**have**  $1: \neg\ Ide\ T \implies seq\ [t]\ (standardize\ T)$

using  $t\ T\ ind\ 0\ ide\text{-char}\ Con\text{-implies-Arr}(1)$

**apply** (*intro seqI<sub>ΛP</sub>*)

**apply** *simp*

**apply** (*metis Con-implies-Arr(1) Ide.simps(1) ide-char*)

by (*metis Src-hd-eqI Trg-last-Src-hd-eqI ‹T ≠ []› append-Cons arrI<sub>P</sub>*  
 $arr\text{-append-imp-seq}\ list.\text{distinct}(1)\ self\text{-append-conv2}\ tT)$

**show**  $?thesis$

using  $T\ 1\ ind\ Std\text{-standard-development}\ stdz\text{-insert-correctness}$  by *auto*

**qed**

**show**  $Ide\ (t\ \# T) \longrightarrow standardize\ (t\ \# T) = []$

```

using Ide-consE Ide-iff-standard-development-empty Ide-implies-Arr ind
  Λ.Ide-implies-Arr Λ.ide-char
by (metis list.sel(1,3) standardize.simps(1-2) stdz-insert.simps(1))
show  $\neg \text{Ide } (t \# T) \longrightarrow \text{standardize } (t \# T) \text{ *~* } t \# T$ 
proof
  assume  $1: \neg \text{Ide } (t \# T)$ 
  show  $\text{standardize } (t \# T) \text{ *~* } t \# T$ 
  proof (cases Λ.Ide t)
    assume  $t: \Lambda.\text{Ide } t$ 
    have  $2: \neg \text{Ide } T$ 
    using  $1 \ t \ tT$  by fastforce
    have  $\text{standardize } (t \# T) = \text{stdz-insert } t \ (\text{standardize } T)$ 
    by simp
    also have ...  $\text{ *~* } t \# T$ 
    proof –
      have  $3: \text{Std } (\text{standardize } T) \wedge \text{standardize } T \text{ *~* } T$ 
      using  $T \ 2$  ind by blast
      have  $\text{stdz-insert } t \ (\text{standardize } T) =$ 
         $\text{stdz-insert } (\text{hd } (\text{standardize } T)) \ (\text{tl } (\text{standardize } T))$ 
      proof –
        have  $\text{seq } [t] \ (\text{standardize } T)$ 
        using  $0 \ 2 \ tT$  ind
        by (metis Arr.elims(2) Con-imp-eq-Srcs Con-implies-Arr(1) Ide.simps(1-2))
          Ide-implies-Arr Trgs.simps(2) ide-char Λ.ide-char list.inject
          seq-char seq-implies-Trgs-eq-Srcs t
        thus ?thesis
        using  $t \ 3$  stdz-insert-Ide-Std by blast
      qed
    also have ...  $\text{ *~* } \text{hd } (\text{standardize } T) \# \text{tl } (\text{standardize } T)$ 
    proof –
      have  $\neg \text{Ide } (\text{standardize } T)$ 
      using  $2 \ 3$  ide-backward-stable ide-char by blast
      moreover have  $\text{tl } (\text{standardize } T) \neq [] \implies$ 
         $\text{seq } [\text{hd } (\text{standardize } T)] \ (\text{tl } (\text{standardize } T)) \wedge$ 
         $\text{Std } (\text{tl } (\text{standardize } T))$ 
      by (metis 3 Std-consE Std-imp-Arr append.left-neutral append-Cons)
        arr-append-imp-seq arr-char hd-Cons-tl list.discI tl-Nil
      ultimately show ?thesis
      by (metis 2 Ide.simps(2) Resid.simps(1) Std-consE T cong-standard-development)
        ide-char ind Λ.ide-char list.exhaust-sel stdz-insert.simps(1)
        stdz-insert-correctness
    qed
  also have  $\text{hd } (\text{standardize } T) \# \text{tl } (\text{standardize } T) = \text{standardize } T$ 
  by (metis 3 Arr.simps(1) Con-implies-Arr(2) Ide.simps(1) ide-char)
    list.exhaust-sel
  also have  $\text{standardize } T \text{ *~* } T$ 
  using  $3$  by simp
  also have  $T \text{ *~* } t \# T$ 
  using  $0 \ t \ tT$  arr-append-imp-seq arr-char cong-cons-ideI(2) by simp

```

```

    finally show ?thesis by blast
  qed
  thus ?thesis by auto
  next
  assume t:  $\neg \Lambda.Ide\ t$ 
  show ?thesis
  proof (cases Ide T)
    assume T: Ide T
    have standardize (t # T) = standard-development t
      using t T Ide-implies-Arr ind by simp
    also have ...  $\sim^*$  [t]
      using t T tT cong-standard-development [of t] by blast
    also have [t]  $\sim^*$  [t] @ T
      using t T tT cong-append-ideI(4) [of [t] T]
      by (simp add: 0 arrIP arr-append-imp-seq ide-char)
    finally show ?thesis by auto
  next
  assume T:  $\neg Ide\ T$ 
  have 1: Std (standardize T)  $\wedge$  standardize T  $\sim^*$  T
    using T  $\langle Arr\ T \rangle$  ind by blast
  have 2: seq [t] (standardize T)
    by (metis 0 Arr.simps(2) Arr.simps(3) Con-imp-eq-Srcs Con-implies-Arr(2)
      Ide.elims(3) Ide.simps(1) T Trgs.simps(2) ide-char ind
      seq-char seq-implies-Trgs-eq-Srcs tT)
  have stdz-insert t (standardize T)  $\sim^*$  t # standardize T
    using t 1 2 stdz-insert-correctness [of t standardize T] by blast
  also have t # standardize T  $\sim^*$  t # T
    using 1 2
    by (meson Arr.simps(2)  $\Lambda$ .prfx-reflexive cong-cons seq-char)
  finally show ?thesis by auto
  qed
  qed
  qed
  qed
  qed
  qed

```

## The Leftmost Reduction Theorem

In this section we prove the Leftmost Reduction Theorem, which states that leftmost reduction is a normalizing strategy.

We first show that if a standard reduction path reaches a normal form, then the path must be the one produced by following the leftmost reduction strategy. This is because, in a standard reduction path, once a leftmost redex is skipped, all subsequent reductions occur “to the right of it”, hence they are all non-leftmost reductions that do not contract the skipped redex, which remains in the leftmost position.

The Leftmost Reduction Theorem then follows from the Standardization Theorem. If a term is normalizable, there is a reduction path from that term to a normal form.

By the Standardization Theorem we may as well assume that path is standard. But a standard reduction path to a normal form is the path generated by following the leftmost reduction strategy, hence leftmost reduction reaches a normal form after a finite number of steps.

**lemma** *sseq-reflects-leftmost-reduction*:

**assumes**  $\Lambda.sseq\ t\ u$  **and**  $\Lambda.is\text{-leftmost-reduction}\ u$

**shows**  $\Lambda.is\text{-leftmost-reduction}\ t$

**proof** –

**have** \*:  $\bigwedge u. u = \Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ t) \setminus t \implies \neg \Lambda.sseq\ t\ u$  **for**  $t$

**proof** (*induct*  $t$ )

**show**  $\bigwedge u. \neg \Lambda.sseq\ \# u$

**using**  $\Lambda.sseq\text{-imp-sseq}$  **by** *blast*

**show**  $\bigwedge x\ u. \neg \Lambda.sseq\ \langle x \rangle u$

**using**  $\Lambda.elementary\text{-reduction.simps}(2)$   $\Lambda.sseq\text{-imp-elementary-reduction1}$  **by** *blast*

**show**  $\bigwedge t\ u. \llbracket \bigwedge u. u = \Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ t) \setminus t \implies \neg \Lambda.sseq\ t\ u;$

$u = \Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ \lambda[t]) \setminus \lambda[t] \rrbracket$

$\implies \neg \Lambda.sseq\ \lambda[t]\ u$

**by** *auto*

**show**  $\bigwedge t1\ t2\ u. \llbracket \bigwedge u. u = \Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ t1) \setminus t1 \implies \neg \Lambda.sseq\ t1\ u;$

$\bigwedge u. u = \Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ t2) \setminus t2 \implies \neg \Lambda.sseq\ t2\ u;$

$u = \Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ (\lambda[t1] \bullet t2)) \setminus (\lambda[t1] \bullet t2) \rrbracket$

$\implies \neg \Lambda.sseq\ (\lambda[t1] \bullet t2)\ u$

**apply** *simp*

**by** (*metis*  $\Lambda.sseq\text{-imp-elementary-reduction2}$   $\Lambda.Coinitial\text{-iff-Con}$   $\Lambda.Ide\text{-Src}$

$\Lambda.Ide\text{-Subst}$   $\Lambda.elementary\text{-reduction-not-ide}$   $\Lambda.ide\text{-char}$   $\Lambda.resid\text{-Ide-Arr}$ )

**show**  $\bigwedge t1\ t2. \llbracket \bigwedge u. u = \Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ t1) \setminus t1 \implies \neg \Lambda.sseq\ t1\ u;$

$\bigwedge u. u = \Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ t2) \setminus t2 \implies \neg \Lambda.sseq\ t2\ u;$

$u = \Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ (\Lambda.App\ t1\ t2)) \setminus \Lambda.App\ t1\ t2 \rrbracket$

$\implies \neg \Lambda.sseq\ (\Lambda.App\ t1\ t2)\ u$  **for**  $u$

**apply** (*cases*  $u$ )

**apply** *simp-all*

**apply** (*metis*  $\Lambda.elementary\text{-reduction.simps}(2)$   $\Lambda.sseq\text{-imp-elementary-reduction2}$ )

**apply** (*metis*  $\Lambda.Src.simps(3)$   $\Lambda.Src\text{-resid}$   $\Lambda.Trq.simps(3)$   $\Lambda.lambda.distinct(15)$

$\Lambda.lambda.distinct(3)$ )

**proof** –

**show**  $\bigwedge t1\ t2\ u1\ u2.$

$\llbracket \neg \Lambda.sseq\ t1\ (\Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ t1) \setminus t1);$

$\neg \Lambda.sseq\ t2\ (\Lambda.leftmost\text{-strategy}\ (\Lambda.Src\ t2) \setminus t2);$

$\lambda[u1] \bullet u2 = \Lambda.leftmost\text{-strategy}\ (\Lambda.App\ (\Lambda.Src\ t1)\ (\Lambda.Src\ t2)) \setminus \Lambda.App\ t1\ t2;$

$u = \Lambda.leftmost\text{-strategy}\ (\Lambda.App\ (\Lambda.Src\ t1)\ (\Lambda.Src\ t2)) \setminus \Lambda.App\ t1\ t2 \rrbracket$

$\implies \neg \Lambda.sseq\ (\Lambda.App\ t1\ t2)$

$(\Lambda.leftmost\text{-strategy}\ (\Lambda.App\ (\Lambda.Src\ t1)\ (\Lambda.Src\ t2)) \setminus \Lambda.App\ t1\ t2)$

**by** (*metis*  $\Lambda.sseq\text{-imp-elementary-reduction1}$   $\Lambda.Arr.simps(5)$   $\Lambda.Arr\text{-resid}$

$\Lambda.Coinitial\text{-iff-Con}$   $\Lambda.Ide.simps(5)$   $\Lambda.Ide\text{-iff-Src-self}$   $\Lambda.Src.simps(4)$

$\Lambda.Src\text{-resid}$   $\Lambda.contains\text{-head-reduction.simps}(8)$   $\Lambda.is\text{-head-reduction-if}$

$\Lambda.lambda.discI(3)$   $\Lambda.lambda.distinct(7)$

$\Lambda.leftmost\text{-strategy-selects-head-reduction}$   $\Lambda.resid\text{-Arr-self}$

$\Lambda.sseq\text{-preserves-App-and-no-head-reduction}$ )

**show**  $\bigwedge u1\ u2.$

```

[[¬ Λ.sseq t1 (Λ.leftmost-strategy (Λ.Src t1) \ t1);
¬ Λ.sseq t2 (Λ.leftmost-strategy (Λ.Src t2) \ t2);
Λ.App u1 u2 = Λ.leftmost-strategy (Λ.App (Λ.Src t1) (Λ.Src t2)) \ Λ.App t1 t2;
u = Λ.leftmost-strategy (Λ.App (Λ.Src t1) (Λ.Src t2)) \ Λ.App t1 t2]]
⇒ ¬ Λ.sseq (Λ.App t1 t2)
(Λ.leftmost-strategy (Λ.App (Λ.Src t1) (Λ.Src t2)) \ Λ.App t1 t2)
for t1 t2
apply (cases ¬ Λ.Arr t1)
apply simp-all
apply (meson Λ.Arr.simps(4) Λ.seq-char Λ.sseq-imp-seq)
apply (cases ¬ Λ.Arr t2)
apply simp-all
apply (meson Λ.Arr.simps(4) Λ.seq-char Λ.sseq-imp-seq)
using Λ.Arr-not-Nil
apply (cases t1)
apply simp-all
using Λ.NF-iff-has-no-redex Λ.has-redex-iff-not-Ide-leftmost-strategy
Λ.Ide-iff-Src-self Λ.Ide-iff-Trg-self
Λ.NF-def Λ.elementary-reduction-not-ide Λ.eq-Ide-are-cong
Λ.leftmost-strategy-is-reduction-strategy Λ.reduction-strategy-def
Λ.resid-Arr-Src
apply simp
apply (metis Λ.Arr.simps(4) Λ.Ide.simps(4) Λ.Ide-Trg Λ.Src.simps(4)
Λ.sseq-imp-elementary-reduction2)
by (metis Λ.Ide-Trg Λ.elementary-reduction-not-ide Λ.ide-char)
qed
qed
have t ≠ Λ.leftmost-strategy (Λ.Src t) ⇒ False
proof –
assume 1: t ≠ Λ.leftmost-strategy (Λ.Src t)
have 2: ¬ Λ.Ide (Λ.leftmost-strategy (Λ.Src t))
by (meson assms(1) Λ.NF-def Λ.NF-iff-has-no-redex Λ.arr-char
Λ.elementary-reduction-is-arr Λ.elementary-reduction-not-ide
Λ.has-redex-iff-not-Ide-leftmost-strategy Λ.ide-char
Λ.sseq-imp-elementary-reduction1)
have Λ.is-leftmost-reduction (Λ.leftmost-strategy (Λ.Src t) \ t)
proof –
have Λ.is-leftmost-reduction (Λ.leftmost-strategy (Λ.Src t))
by (metis assms(1) 2 Λ.Ide-Src Λ.Ide-iff-Src-self Λ.arr-char
Λ.elementary-reduction-is-arr Λ.elementary-reduction-leftmost-strategy
Λ.is-leftmost-reduction-def Λ.leftmost-strategy-is-reduction-strategy
Λ.reduction-strategy-def Λ.sseq-imp-elementary-reduction1)
moreover have 3: Λ.elementary-reduction t
using assms Λ.sseq-imp-elementary-reduction1 by simp
moreover have ¬ Λ.is-leftmost-reduction t
using 1 Λ.is-leftmost-reduction-def by auto
moreover have Λ.coinitial (Λ.leftmost-strategy (Λ.Src t)) t
using 3 Λ.leftmost-strategy-is-reduction-strategy Λ.reduction-strategy-def
Λ.Ide-Src Λ.elementary-reduction-is-arr

```

by force  
 ultimately show ?thesis  
 using 1  $\Lambda$ .leftmost-reduction-preservation by blast  
 qed  
 moreover have  $\Lambda$ .coinitial ( $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src t) \ t) u  
 using assms(1) calculation  $\Lambda$ .Arr-not-Nil  $\Lambda$ .Src-resid  $\Lambda$ .elementary-reduction-is-arr  
 $\Lambda$ .is-leftmost-reduction-def  $\Lambda$ .seq-char  $\Lambda$ .sseq-imp-seq  
 by force  
 moreover have  $\bigwedge v$ . [ $\Lambda$ .is-leftmost-reduction v;  $\Lambda$ .coinitial v u]  $\implies$  v = u  
 by (metis  $\Lambda$ .arr-iff-has-source  $\Lambda$ .arr-resid-iff-con  $\Lambda$ .confluence assms(2)  
 $\Lambda$ .Arr-not-Nil  $\Lambda$ .Cointial-iff-Con  $\Lambda$ .is-leftmost-reduction-def  $\Lambda$ .sources-char $_{\Lambda}$ )  
 ultimately have  $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src t) \ t = u  
 by blast  
 thus ?thesis  
 using assms(1) \* by blast  
 qed  
 thus ?thesis  
 using assms(1)  $\Lambda$ .is-leftmost-reduction-def  $\Lambda$ .sseq-imp-elementary-reduction1 by force  
 qed

lemma elementary-reduction-to-NF-is-leftmost:

shows [ $\Lambda$ .elementary-reduction t;  $\Lambda$ .NF (Trg [t])]  $\implies$   $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src t) = t

proof (induct t)

show  $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src #) = #

by simp

show  $\bigwedge x$ . [ $\Lambda$ .elementary-reduction «x»;  $\Lambda$ .NF (Trg [«x»])]  $\implies$   $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src «x») = «x»

by auto

show  $\bigwedge t$ . [[ $\Lambda$ .elementary-reduction t;  $\Lambda$ .NF (Trg [t])]  $\implies$   $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src t) = t;  
 $\Lambda$ .elementary-reduction  $\lambda[t]$ ;  $\Lambda$ .NF (Trg [ $\lambda[t]$ ])]  $\implies$   $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src  $\lambda[t]$ ) =  $\lambda[t]$

using lambda-calculus.NF-Lam-iff lambda-calculus.elementary-reduction-is-arr by force

show  $\bigwedge t1$  t2. [[ $\Lambda$ .elementary-reduction t1;  $\Lambda$ .NF (Trg [t1])]  $\implies$   $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src t1) = t1;  
[[ $\Lambda$ .elementary-reduction t2;  $\Lambda$ .NF (Trg [t2])]  $\implies$   $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src t2) = t2;  
 $\Lambda$ .elementary-reduction ( $\lambda[t1]$  • t2);  $\Lambda$ .NF (Trg [ $\lambda[t1]$  • t2])]  $\implies$   $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src ( $\lambda[t1]$  • t2)) =  $\lambda[t1]$  • t2

apply simp

by (metis  $\Lambda$ .Ide-iff-Src-self  $\Lambda$ .Ide-implies-Arr)

fix t1 t2

assume ind1: [ $\Lambda$ .elementary-reduction t1;  $\Lambda$ .NF (Trg [t1])]  $\implies$   $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src t1) = t1

assume ind2: [ $\Lambda$ .elementary-reduction t2;  $\Lambda$ .NF (Trg [t2])]  $\implies$   $\Lambda$ .leftmost-strategy ( $\Lambda$ .Src t2) = t2

assume t:  $\Lambda$ .elementary-reduction ( $\Lambda$ .App t1 t2)

have t1:  $\Lambda$ .Arr t1

using t  $\Lambda$ .Arr.simps(4)  $\Lambda$ .elementary-reduction-is-arr by blast

**have**  $t2: \Lambda.Arr\ t2$   
**using**  $t\ \Lambda.Arr.simps(4)\ \Lambda.elementary-reduction-is-arr$  **by** *blast*  
**assume**  $NF: \Lambda.NF\ (Trg\ [\Lambda.App\ t1\ t2])$   
**have**  $1: \neg\ \Lambda.is-Lam\ t1$   
**using**  $NF\ \Lambda.NF-def$   
**apply**  $(cases\ t1)$   
**apply** *simp-all*  
**by**  $(metis\ (mono-tags)\ \Lambda.Ide.simps(1)\ \Lambda.NF-App-iff\ \Lambda.Trig.simps(2-3)\ \Lambda.lambda.discI(2))$   
**have**  $2: \Lambda.NF\ (\Lambda.Trig\ t1) \wedge \Lambda.NF\ (\Lambda.Trig\ t2)$   
**using**  $NF\ t1\ t2\ 1\ \Lambda.NF-App-iff$  **by** *simp*  
**show**  $\Lambda.leftmost-strategy\ (\Lambda.Src\ (\Lambda.App\ t1\ t2)) = \Lambda.App\ t1\ t2$   
**using**  $t\ t1\ t2\ 1\ 2\ ind1\ ind2$   
**apply**  $(cases\ t1)$   
**apply** *simp-all*  
**apply**  $(metis\ \Lambda.Ide.simps(4)\ \Lambda.Ide-iff-Src-self\ \Lambda.Ide-iff-Trig-self$   
 $\Lambda.NF-iff-has-no-redex\ \Lambda.elementary-reduction-not-ide\ \Lambda.eq-Ide-are-cong$   
 $\Lambda.has-redex-iff-not-Ide-leftmost-strategy\ \Lambda.resid-Arr-Src\ t1)$   
**using**  $\Lambda.Ide-iff-Src-self$  **by** *blast*  
**qed**

**lemma** *Std-path-to-NF-is-leftmost:*

**shows**  $\llbracket Std\ T; \Lambda.NF\ (Trg\ T) \rrbracket \implies set\ T \subseteq Collect\ \Lambda.is-leftmost-reduction$

**proof** –

**have**  $1: \bigwedge t. \llbracket Std\ (t\ \# T); \Lambda.NF\ (Trg\ (t\ \# T)) \rrbracket \implies \Lambda.is-leftmost-reduction\ t$  **for**  $T$

**proof**  $(induct\ T)$

**show**  $\bigwedge t. \llbracket Std\ [t]; \Lambda.NF\ (Trg\ [t]) \rrbracket \implies \Lambda.is-leftmost-reduction\ t$

**using** *elementary-reduction-to-NF-is-leftmost*  $\Lambda.is-leftmost-reduction-def$  **by** *simp*

**fix**  $t\ u\ T$

**assume**  $ind: \bigwedge t. \llbracket Std\ (t\ \# T); \Lambda.NF\ (Trg\ (t\ \# T)) \rrbracket \implies \Lambda.is-leftmost-reduction\ t$

**assume**  $Std: Std\ (t\ \# u\ \# T)$

**assume**  $\Lambda.NF\ (Trg\ (t\ \# u\ \# T))$

**show**  $\Lambda.is-leftmost-reduction\ t$

**using**  $Std\ \langle \Lambda.NF\ (Trg\ (t\ \# u\ \# T)) \rangle$  *ind* *sseq-reflects-leftmost-reduction* **by** *auto*

**qed**

**show**  $\llbracket Std\ T; \Lambda.NF\ (Trg\ T) \rrbracket \implies set\ T \subseteq Collect\ \Lambda.is-leftmost-reduction$

**proof**  $(induct\ T)$

**show**  $2: set\ [] \subseteq Collect\ \Lambda.is-leftmost-reduction$

**by** *simp*

**fix**  $t\ T$

**assume**  $ind: \llbracket Std\ T; \Lambda.NF\ (Trg\ T) \rrbracket \implies set\ T \subseteq Collect\ \Lambda.is-leftmost-reduction$

**assume**  $Std: Std\ (t\ \# T)$  **and**  $NF: \Lambda.NF\ (Trg\ (t\ \# T))$

**show**  $set\ (t\ \# T) \subseteq Collect\ \Lambda.is-leftmost-reduction$

**by**  $(metis\ 1\ 2\ NF\ Std\ Std-consE\ Trig.elims\ ind\ insert-subset\ list.inject\ list.simps(15)$   
 $mem-Collect-eq)$

**qed**

**qed**

**theorem** *leftmost-reduction-theorem:*

**shows**  $\Lambda.normalizing-strategy\ \Lambda.leftmost-strategy$

```

proof (unfold  $\Lambda$ .normalizing-strategy-def, intro allI impI)
  fix a
  assume a:  $\Lambda$ .normalizable a
  show  $\exists n. \Lambda$ .NF ( $\Lambda$ .reduce  $\Lambda$ .leftmost-strategy a n)
  proof (cases  $\Lambda$ .NF a)
    show  $\Lambda$ .NF a  $\implies$  ?thesis
      by (metis lambda-calculus.reduce.simps(1))
    assume 1:  $\neg \Lambda$ .NF a
    obtain T where T: Arr T  $\wedge$  Src T = a  $\wedge$   $\Lambda$ .NF (Trg T)
      using a  $\Lambda$ .normalizable-def red-iff by auto
    have 2:  $\neg$  Ide T
      using T 1 Ide-imp-Src-eq-Trg by fastforce
    obtain U where U: Std U  $\wedge$  cong T U
      using T 2 standardization-theorem by blast
    have 3: set U  $\subseteq$  Collect  $\Lambda$ .is-leftmost-reduction
      using 1 U Std-path-to-NF-is-leftmost
      by (metis Con-Arr-self Resid-parallel Src-resid T cong-implies-coinitial)
    have  $\bigwedge U. \llbracket$ Arr U; length U = n; set U  $\subseteq$  Collect  $\Lambda$ .is-leftmost-reduction $\rrbracket \implies$ 
      U = apply-strategy  $\Lambda$ .leftmost-strategy (Src U) (length U) for n
  proof (induct n)
    show  $\bigwedge U. \llbracket$ Arr U; length U = 0; set U  $\subseteq$  Collect  $\Lambda$ .is-leftmost-reduction $\rrbracket$ 
       $\implies$  U = apply-strategy  $\Lambda$ .leftmost-strategy (Src U) (length U)
      by simp
    fix n U
    assume ind:  $\bigwedge U. \llbracket$ Arr U; length U = n; set U  $\subseteq$  Collect  $\Lambda$ .is-leftmost-reduction $\rrbracket$ 
       $\implies$  U = apply-strategy  $\Lambda$ .leftmost-strategy (Src U) (length U)
    assume U: Arr U
    assume n: length U = Suc n
    assume set: set U  $\subseteq$  Collect  $\Lambda$ .is-leftmost-reduction
    show U = apply-strategy  $\Lambda$ .leftmost-strategy (Src U) (length U)
    proof (cases n = 0)
      show n = 0  $\implies$  ?thesis
        using U n 1 set  $\Lambda$ .is-leftmost-reduction-def
        by (cases U) auto
      assume 5: n  $\neq$  0
      have 4: hd U =  $\Lambda$ .leftmost-strategy (Src U)
        using n U set  $\Lambda$ .is-leftmost-reduction-def
        by (cases U) auto
      have 6: tl U  $\neq$  []
        using 4 5 n U
        by (metis Suc-length-conv list.sel(3) list.size(3))
      show ?thesis
        using 4 5 6 n U set ind [of tl U]
        apply (cases n)
        apply simp-all
        by (metis (no-types, lifting) Arr-consE Nil-tl Nitpick.size-list-simp(2)
          ind [of tl U]  $\Lambda$ .arr-char  $\Lambda$ .trg-char list.collapse list.set-sel(2)
          old.nat.inject reduction-paths.apply-strategy.simps(2) subset-code(1))
  qed

```

**qed**  
**hence**  $U = \text{apply-strategy } \Lambda.\text{leftmost-strategy } (\text{Src } U) (\text{length } U)$   
**by** (*metis 3 Con-implies-Arr(1) Ide.simps(1) U ide-char*)  
**moreover have**  $\text{Src } U = a$   
**using**  $T \ U \ \text{cong-implies-coinitial}$   
**by** (*metis Con-imp-eq-Srcs Con-implies-Arr(2) Ide.simps(1) Srcs-simp<sub>PWE</sub> empty-set*  
*ex-un-Src ide-char list.set-intros(1) list.simps(15)*)  
**ultimately have**  $\text{Trg } U = \Lambda.\text{reduce } \Lambda.\text{leftmost-strategy } a (\text{length } U)$   
**using**  $\text{reduce-eq-Trg-apply-strategy}$   
**by** (*metis Arr.simps(1) Con-implies-Arr(1) Ide.simps(1) U a ide-char*  
 $\Lambda.\text{leftmost-strategy-is-reduction-strategy } \Lambda.\text{normalizable-def length-greater-0-conv}$ )  
**thus** *?thesis*  
**by** (*metis Ide.simps(1) Ide-imp-Src-eq-Trg Src-resid T Trg-resid-sym U ide-char*)  
**qed**  
**qed**  
  
**end**  
  
**end**

# Bibliography

- [1] H. Barendregt. *The Lambda-calculus: Its Syntax and Semantics*. North-Holland, 1984.
- [2] M. Copes. A machine-checked proof of the standardization theorem in lambda calculus using multiple substitution. Master's thesis, Universidad ORT Uruguay, 2018. <https://dspace.ort.edu.uy/bitstream/handle/20.500.11968/3725/Material%20completo.pdf>.
- [3] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.
- [4] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 34(5):381–392, 1972.
- [5] R. de Vrijer. A direct proof of the finite developments theorem. *The Journal of Symbolic Logic*, 50(2):339–343, June 1985.
- [6] R. Hindley. Reductions of residuals are finite. *Transactions of the American Mathematical Society*, 240:345–361, June 1978.
- [7] G. Huet. Residual theory in  $\lambda$ -calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.
- [8] J.-J. Lévy. *Réductions correctes et optimales dans le  $\lambda$ -calcul*. PhD thesis, U. Paris VII, 1978. Thèse d'Etat.
- [9] D. E. Schroer. *The Church-Rosser Theorem*. PhD thesis, Cornell University, 1965.
- [10] E. W. Stark. Concurrent transition systems. *Theoretical Computer Science*, 64:221–269, July 1989.
- [11] E. W. Stark. Category theory with adjunctions and limits. *Archive of Formal Proofs*, June 2016. <http://isa-afp.org/entries/Category3.shtml>, Formal proof development.