Residuated Transition Systems II: Categorical Properties

Eugene W. Stark

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

June 17, 2024

Abstract

This article extends the formal development of the theory of residuated transition systems (RTS's), begun in the author's previous article, to include category-theoretic properties. There are two main themes: (1) RTS's *as* categories; and (2) RTS's *in* categories. Concerning the first theme, we show that every RTS determines a category via the "composite completion" construction given in the previous article, and we obtain a characterization of the "categories of transitions" that arise in this way. Concerning the second theme, we show that the "small" extensional RTS's having arrows that inhabit a type with suitable closure properties, form the objects of a cartesian closed category **RTS** when equipped with simulations as morphisms. This category can in turn be regarded as contained in a larger, 2-category-like structure which is a category under "horizontal" composition, and is an RTS under "vertical" residuation. We call such structures *RTS-categories*. We explore in particular detail the RTS-category $\mathbf{RTS}^\dagger$ having RTS's as objects, simulations as 1-cells, and transformations between simulations as 2-cells. As a category, $\mathbf{RTS}^\dagger$ is also cartesian closed, and the category **RTS** occurs as the subcategory comprised of the arrows that are identities with respect to the residuation. To obtain these results various technical issues, related to the formalization within the relatively weak HOL, have to be addressed. We also consider RTS-categories from the point of view of enriched category theory and we show that RTS-categories are essentially the same thing as categories enriched in **RTS**, and that the RTS-category $\mathbf{RTS}^\dagger$ is determined up to isomorphism by its cartesian closed subcategory **RTS**.

Contents

Contents **2**

Introduction **3**

1 Preliminaries **8**
1.1 Simulations 8
1.2 Transformations 18
1.3 Binary Simulations 25
1.4 Horizontal Composite of Transformations 29

2 RTS's as Categories **33**
2.1 Categories with Bounded Pushouts 33
2.1.1 Bounded Spans 33
2.1.2 Pushouts 34
2.2 Categories of Transitions 36
2.3 Extensional RTS's with Composites as Categories 39
2.4 Characterization 42

3 RTS Constructions **48**
3.1 Notation 48
3.2 Some Constraints on a Type 49
3.2.1 Nondegenerate 49
3.2.2 Lifting 49
3.2.3 Pairing 50
3.2.4 Exponentiation 51
3.2.5 Universe 54
3.3 Small RTS's 55
3.4 Injective Images of RTS's 61
3.5 Empty RTS 67
3.6 One-Transition RTS 69
3.7 Sub-RTS 74
3.8 Fibered Product RTS 77
3.9 Product RTS 85
3.9.1 Associators 100

3.10 Exponential RTS . . . 101
3.10.1 Exponential of Small RTS's . . . 134
3.10.2 Exponential into RTS with Composites . . . 136
3.10.3 Exponential by One . . . 140
3.10.4 Evaluation Map . . . 143
3.10.5 Currying . . . 145
3.10.6 Currying and Uncurrying as Inverse Simulations . . . 172
3.10.7 Coextension of a Simulation . . . 204
3.10.8 Compositors . . . 215
3.10.9 Functoriality of Exponential . . . 220

4 RTS's in Categories 230
4.1 RTS-Categories . . . 230
4.1.1 Definition and Basic Properties . . . 230
4.1.2 Hom-RTS's . . . 237
4.1.3 Additional Notions . . . 247
4.2 Concrete RTS-Categories . . . 250
4.3 The RTS-Category of RTS's and Transformations . . . 267
4.3.1 Terminal Object . . . 282
4.3.2 Products . . . 299
4.3.3 Exponentials . . . 322
4.3.4 Cartesian Closure . . . 358
4.3.5 Repleteness . . . 359
4.4 The Category of RTS's and Simulations . . . 377
4.4.1 Terminal Object . . . 383
4.4.2 Products . . . 385
4.4.3 Exponentials . . . 392
4.4.4 Cartesian Closure . . . 394
4.4.5 Associators . . . 395
4.4.6 Compositors . . . 404
4.5 Top-Level Interpretation . . . 420

5 RTS-Enriched Categories 422
5.1 RTS-Enriched Categories . . . 422
5.2 RTS-Enriched Categories induce RTS-Categories . . . 425
5.2.1 The Small Case . . . 458
5.2.2 Functoriality . . . 470
5.3 RTS-Categories induce RTS-Enriched Categories . . . 478
5.3.1 Functoriality . . . 500
5.4 Equivalence of RTS-Enriched Categories and RTS-Categories 510
5.4.1 RTS-Category to Enriched Category to RTS-Category 510
5.4.2 Enriched Category to RTS-Category to Enriched Category . . . 516
5.5 **RTS**† Determined by its Underlying Category . . . 533

Bibliography **547**

Introduction

This article continues the formal development of the theory of residuated transition systems (RTS's) which was begun in the previous article [4]. A particular theme of the present article is the development of category-theoretic properties. These were intentionally omitted from the previous article in order to avoid dependence of the basic RTS theories on a development of category theory. The present article has two main themes: (1) considering RTS's as themselves being (or perhaps generating) categories; and (2) studying RTS's as objects within categories. With respect to the first theme, we recall from the previous article that every RTS determines an extensional RTS with composites – its "composite completion." As a structure consisting of a set of arrows equipped with a partial composition that is associative and satisfies left and right identity laws, an RTS with composites can obviously be regarded as a category. In view of the fact that the composition is derived from an underlying residuation operation, it is straightforward to show that such a category has unique "bounded pushouts"; that is, a unique pushout exists for every span that is "bounded" in the sense that it can be completed to a commutative square. In addition, in such a category, every arrow is an epimorphism and there are no non-trivial isomorphisms. We define a "category of transitions" to be a category with these properties, and we show that every such category is in fact an extensional RTS with composites, where consistency of a span of transitions coincides with boundedness and the residuation is obtained from pushouts.

Concerning the second theme, we are interested a category **RTS** whose objects are extensional RTS's and whose morphisms are simulations between such RTS's. Since the set of morphisms between two extensional RTS's A and B is itself an extensional RTS having as its morphisms the transformations between simulations, it is clear that **RTS** has additional structure to be explored here beyond that of a simple category. As we expect the category **RTS** (or closely related structures) to be the main focus of attention in applications of residuated transition systems (to programming language semantics, for example), our objective is to clarify this structure as much as possible and to set up technology for working formally with this category. There are some technical problems that arise with the formalization of this material in the context of Isabelle/HOL, though. First of all, RTS's exist

with arrows at arbitrary types, so it is impossible in HOL to directly formalize a category whose objects encompass "all" RTS's. Instead, we have to consider categories of RTS's whose arrows inhabit some particular type, though we may exploit polymorphism to prove general theorems that hold for any such category. Secondly, the fact that the hom-sets of a category of extensional RTS's and simulations themselves admit the structure of an extensional RTS suggests that we ought to be looking at the category **RTS** as a *closed* category; in fact cartesian closed, as it happens. Here again, we face some limitations posed by our choice to carry out the formalization within Isabelle/HOL. In particular, a cartesian closed category, having RTS's as objects and as homs the sets of all simulations between such, cannot be constructed in pure HOL, unless we restrict our attention to finite RTS's only. However, we can work around this limitation if we are willing to extend pure HOL with the assumption that there is a "universe" type that is "large" enough to be closed under the function space constructions that we need to perform in order to achieve cartesian closure. The Isabelle HOL library already has a well-developed theory of this kind; namely, the *ZFC_in_HOL* theory, which axiomatically extends HOL with a type V and a notion of "smallness", such that the small sets of type V satisfy the axioms of ZFC. In our development, we presuppose the existence of a notion of smallness and suppose that the underlying arrow type of the category **RTS** is closed under the construction of small function spaces, but we have avoided "hard coding" the particular type V defined in *ZFC_in_HOL* into our development. This independence from a particular choice of "universe" comes at a cost, however: in order to show that the category **RTS** admits type-increasing constructions such as products and exponentials, we have to concern ourselves in each case with finding a suitable "type-reducing map" to show that the RTS's constructed with arrows at the higher types are in fact isomorphic to RTS's that already exist at the original arrow type. We note that these complications only arise in showing that **RTS** admits various categorical constructions — once this has been done we can work with these constructions in a simple way using the universal properties that characterize them.

To summarize the above, one of our main results is the construction of a cartesian closed category **RTS**, whose objects are in bijection with "small", extensional RTS's whose arrows inhabit a suitable "universe" type α and each of whose hom-sets $\mathbf{RTS}(A, B)$ is in bijective correspondence with the set of all simulations between A and B. We show that the particular type V axiomatized in *ZFC_in_HOL* satisfies the requirements for the universe type α, but our development does not otherwise depend on details of *ZFC_in_HOL* except for the notion of smallness defined therein. We prove theorems that allow us to pass back and forth between notions internal to **RTS** and corresponding external notions expressed in terms of the concrete structure of RTS's and simulations.

The fact that **RTS** is cartesian closed is a consequence of the fact that the transformations between simulations from an extensional RTS A to an extensional RTS B may themselves be regarded as the arrows of an exponential RTS $[A, B]$. The category **RTS** may therefore be regarded as a category "enriched in itself" [1]. However, it seems more natural to think of **RTS** as something more like a 2-category, where the 0-cells correspond to RTS's, the 1-cells correspond to simulations, and the 2-cells correspond to transformations between simulations. Unless we restrict ourselves *a priori* to RTS's with composites (something that we do not wish to do), the resulting structure will not actually be a 2-category, because the homs will in general be RTS's that do not necessarily admit composition of transitions (*i.e.* of transformations). So instead the kind of structure we obtain consists of a category under "horizontal" composition, and an RTS under "vertical" residuation. We formalize such a structure, calling it an "RTS-category". We show that there is an RTS-category $\mathbf{RTS}^\dagger$, whose 0-cells (objects) are in bijection with the small, extensional RTS's with arrows at a universe type α, whose 1-cells (arrows) are in bijection with simulations between such RTS's, and whose 2-cells are in bijection with transformations between such simulations. As a category, $\mathbf{RTS}^\dagger$ is itself cartesian closed, and the subcategory defined by the 1-cells (which coincide with the identities of the residuation) is the ordinary cartesian closed category **RTS**. We prove results that allow us to pass back and forth between notions internal to $\mathbf{RTS}^\dagger$ and the corresponding external notions. The construction of $\mathbf{RTS}^\dagger$ and the proof of associated facts constitutes a second group of main results of this article.

Finally, our third group of main results concerns the clarification of the relationship between the notion of RTS-category and that of a category enriched in **RTS**. We show that from a category E enriched in **RTS** we can construct an RTS-category C having as its set of 2-cells the disjoint union of the sets of arrows of the RTS's underlying the hom-objects of E. Conversely, given an RTS-category C we can construct a corresponding category E enriched in **RTS** by taking as the "hom-objects" of E the objects of **RTS** corresponding to the "hom-RTS's" of C. These correspondences are functorial and extend to an equivalence between a category of RTS-categories and a category of **RTS**-enriched categories (for a suitable definition of morphism in each case). So, RTS-categories and categories enriched in **RTS** amount to the same thing, though the definition of RTS-categories is more elementary and will likely be easier to work with in applications.

The remainder of this article is organized as follows: In Chapter 1, we have proved various facts we need about RTS's, simulations, and transformations, which are not part of the previous article [4]. In Chapter 2, we present the results discussed above which pertain to the theme "RTS's as Categories". Chapter 3 defines various concrete constructions on RTS's, including product and exponential, and proves associated universal properties. In addition, this section defines the constraints on a type α required for it to

serve as a "universe" and establishes related facts. Finally, in Chapter 4, we define the notion of RTS-category, construct the RTS-category $\mathbf{RTS}^{\dagger}$ and prove facts about it, including cartesian closure, construct the subcategory **RTS** and prove facts about it as well, and finally establish the equivalence of RTS-categories and categories enriched in **RTS**.

Chapter 1

Preliminaries

This section develops some extensions to theories contained in the previous AFP articles [4] and [3].

theory *Preliminaries*
imports *Main HOL−Library.FuncSet*
ResiduatedTransitionSystem.ResiduatedTransitionSystem
begin

lemma (**in** *extensional-rts*) *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

1.1 Simulations

abbreviation *I*
where *I* ≡ *identity-simulation.map*

lemma *comp-identity-simulation*:
assumes *simulation A B F*
shows *I B* ∘ *F* = *F*
using *assms simulation.extensional simulation.preserves-reflects-arr*
by *fastforce*

lemma *comp-simulation-identity*:
assumes *simulation A B F*
shows *F* ∘ *I A* = *F*

using *assms residuation.not-arr-null rts.axioms(1) simulation.extensional simulation-def*
by *fastforce*

lemma *product-identity-simulation*:
assumes *rts A* **and** *rts B*
shows *product-simulation.map A B (I A) (I B) = I (product-rts.resid A B)*
proof −
interpret *A*: *rts A*
using *assms(1)* **by** *blast*
interpret *B*: *rts B*
using *assms(2)* **by** *blast*
interpret *A*: *identity-simulation A* **..**
interpret *B*: *identity-simulation B* **..**
interpret *AxB*: *product-rts A B* **..**
interpret *IAxIB*: *product-simulation A B A B A.map B.map* **..**
interpret *AxB*: *identity-simulation AxB.resid* **..**
show *IAxIB.map = AxB.map*
using *IAxIB.map-def* **by** *auto*
qed

locale *constant-simulation* =
A: *rts A* +
B: *rts B*
for *A* :: *'a resid* (**infix** $\backslash_A$ *70*)
and *B* :: *'b resid* (**infix** $\backslash_B$ *70*)
and *b* :: *'b* +
assumes *ide-b*: *B.ide b*
begin

abbreviation *map*
where *map t* ≡ *if A.arr t then b else B.null*

sublocale *simulation A B map*
using *ide-b A.con-implies-arr*
by *unfold-locales auto*

end

locale *inverse-simulations* =
A: *rts A* +
B: *rts B* +
F: *simulation B A F* +
G: *simulation A B G*
for *A* :: *'a resid* (**infix** $\backslash_A$ *70*)
and *B* :: *'b resid* (**infix** $\backslash_B$ *70*)
and *F* :: *'b* ⇒ *'a*
and *G* :: *'a* ⇒ *'b* +

assumes *inv*: $G\ o\ F = I\ B$
and *inv'*: $F\ o\ G = I\ A$
begin

lemma *inv-simp* [*simp*]:
assumes $B.arr\ y$
shows $G\ (F\ y) = y$
using *assms inv* **by** (*metis comp-apply*)

lemma *inv'-simp* [*simp*]:
assumes $A.arr\ x$
shows $F\ (G\ x) = x$
using *assms inv'* **by** (*metis comp-apply*)

lemma *induce-bij-betw-arr-sets*:
shows *bij-betw F* (*Collect B.arr*) (*Collect A.arr*)
using *inv inv'* **by** (*intro bij-betwI*) *auto*

end

lemma *inverse-simulations-sym*:
assumes *inverse-simulations A B F G*
shows *inverse-simulations B A G F*
using *assms*
by (*simp add*: *inverse-simulations-axioms-def inverse-simulations-def*)

locale *invertible-simulation* =
simulation +
assumes *invertible*: $\exists\, G.$ *inverse-simulations A B G F*

lemma *invertible-simulation-def'*:
shows *invertible-simulation A B F* $\longleftrightarrow$ $(\exists\, G.$ *inverse-simulations A B G F*$)$
using *inverse-simulations-def invertible-simulation-axioms-def*
invertible-simulation-def
by *blast*

lemma *invertible-simulation-iff*:
shows *invertible-simulation A B F* $\longleftrightarrow$
simulation A B F $\wedge$
bij-betw F (*Collect* (*residuation.arr A*)) (*Collect* (*residuation.arr B*)) $\wedge$
$(\forall\, t\ u.$ *residuation.con B* (*F t*) (*F u*) $\longrightarrow$ *residuation.con A t u*)
proof (*intro iffI conjI*)
assume *F*: *invertible-simulation A B F*
interpret *F*: *invertible-simulation A B F*
using *F* **by** *blast*
obtain *G* **where** *G*: *inverse-simulations A B G F*
using *F.invertible* **by** *blast*
interpret *FG*: *inverse-simulations A B G F*

```
  using G by blast
show simulation A B F
  using F.simulation-axioms by blast
have ∗: ⋀y. y ∈ Collect F.B.arr ⟹ F (G y) = y
  by (metis CollectD FG.inv comp-apply)
show bij-betw F (Collect F.A.arr) (Collect F.B.arr)
  using G inverse-simulations.induce-bij-betw-arr-sets inverse-simulations-sym
  by blast
show ∀ t u. F.B.con (F t) (F u) ⟶ F.A.con t u
by (metis F.B.not-con-null(1) F.B.not-con-null(2) F.extensional FG.F.preserves-con
    FG.inv′ comp-apply)
next
assume F: simulation A B F ∧
        bij-betw F (Collect (residuation.arr A)) (Collect (residuation.arr B)) ∧
        (∀ t u. residuation.con B (F t) (F u) ⟶ residuation.con A t u)
interpret F: simulation A B F
  using F by blast
let ?G = λy. if y ∈ F ‘ Collect F.A.arr
             then inv-into (Collect F.A.arr) F y
             else F.A.null
interpret G: simulation B A ?G
proof
  show ⋀t. ¬ F.B.arr t ⟹ ?G t = F.A.null
    by fastforce
  fix t u
  assume 1: F.B.con t u
  have 2: ?G t = inv-into (Collect F.A.arr) F t ∧
          ?G u = inv-into (Collect F.A.arr) F u
    using 1 F F.B.con-implies-arr
    by (simp add: bij-betw-def)
  have 3: F.A.con (inv-into (Collect F.A.arr) F t)
                  (inv-into (Collect F.A.arr) F u)
    using 1 F.B.con-implies-arr F
    by (metis bij-betw-imp-surj-on f-inv-into-f mem-Collect-eq)
  thus 4: F.A.con (?G t) (?G u)
    using 2 by simp
  have ?G (B t u) = inv-into (Collect F.A.arr) F (B t u)
    by (metis 1 F CollectI F.B.arr-resid-iff-con bij-betw-def)
  also have ... = A (?G t) (?G u)
  proof −
    have F (A (?G t) (?G u)) = B (F (?G t)) (F (?G u))
      using 4 F.preserves-resid by presburger
    also have ... = B t u
      using 4 F.A.not-con-null(1−2) f-inv-into-f
      by (auto simp add: f-inv-into-f)
    finally have F (A (?G t) (?G u)) = B t u by blast
    thus ?thesis
      by (metis 4 F F.A.arr-resid bij-betw-inv-into-left mem-Collect-eq)
  qed
```

```
    finally show ?G (B t u) = A (?G t) (?G u) by blast
  qed
  interpret FG: inverse-simulations A B ?G F
  proof
    show F ∘ ?G = I B
    proof
      fix t
      show (F ∘ ?G) t = I B t
        apply simp
        by (metis F F.A.not-arr-null bij-betw-def f-inv-into-f mem-Collect-eq
            simulation.extensional)
    qed
    show ?G ∘ F = I A
    proof
      fix t
      show (?G ∘ F) t = I A t
        apply simp
      by (metis F F.preserves-reflects-arr bij-betw-def inv-into-f-f mem-Collect-eq)
    qed
  qed
  show invertible-simulation A B F
    using FG.inverse-simulations-axioms invertible-simulation-def
    by unfold-locales blast
qed

context invertible-simulation
begin

  lemma is-bijection-betw-arr-sets:
  shows bij-betw F (Collect A.arr) (Collect B.arr)
    using invertible-simulation-axioms invertible-simulation-iff by metis

  lemma reflects-con:
  assumes residuation.con B (F t) (F u)
  shows residuation.con A t u
    using assms invertible-simulation-axioms invertible-simulation-iff by metis

end

context inverse-simulations
begin

  sublocale F: invertible-simulation B A F
  by (meson inverse-simulations-axioms inverse-simulations-sym invertible-simulation-def')

  sublocale G: invertible-simulation A B G
    by (meson inverse-simulations-axioms invertible-simulation-def')

end
```

lemma *inverse-simulation-unique*:
assumes *inverse-simulations A B G F*
and *inverse-simulations A B G′ F*
shows *G = G′*
proof −
  interpret *FG*: *inverse-simulations A B G F*
    using *assms*(*1*) **by** *simp*
  interpret *FG′*: *inverse-simulations A B G′ F*
    using *assms*(*2*) **by** *simp*
  show *?thesis*
  proof
    fix *x*
    show *G x = G′ x*
    by (*metis FG′.F.extensional FG′.F.preserves-reflects-arr FG′.inv FG.F.extensional*
        FG.inv′ comp-apply)
  qed
qed

locale *inverse-simulation* =
  A: *rts A* +
  B: *rts B* +
  F: *invertible-simulation*
begin

  definition *map*
  where *map* ≡ *SOME G. inverse-simulations A B G F*

  interpretation *inverse-simulations A B map F*
    using *F.invertible invertible-simulation-def map-def*
        someI-ex [*of* λ*G. inverse-simulations A B G F*]
    by *auto*

  sublocale *simulation B A map* **..**

  lemma *is-simulation*:
  shows *simulation B A map*
    ..

  sublocale *inverse-simulations A B map F* **..**

  lemma *map-simp*:
  assumes *B.arr x*
  shows *map x = inv-into* (*Collect A.arr*) *F x*
    by (*metis CollectI F.preserves-reflects-arr assms bij-betw-inv-into-left*
      inv-simp inverse-simulations.induce-bij-betw-arr-sets
      inverse-simulations-axioms inverse-simulations-sym)

  lemma *map-eq*:

```
  shows map = (λx. if B.arr x then inv-into (Collect A.arr) F x else A.null)
    using map-simp by (meson F.extensional)

end

lemma invertible-simulation-identity:
assumes rts A
shows [intro]: invertible-simulation A A (I A)
and inverse-simulations A A (I A) (I A)
and inverse-simulation.map A A (I A) = I A
proof −
  interpret A: rts A
    using assms by blast
  interpret id: identity-simulation A ..
  show 1: inverse-simulations A A id.map id.map
    by unfold-locales auto
  thus invertible-simulation A A id.map
    using id.simulation-axioms invertible-simulation.intro
          invertible-simulation-axioms.intro
    by blast
  show inverse-simulation.map A A id.map = id.map
    using 1 inverse-simulation-unique [of A A id.map id.map]
    by (metis (no-types, lifting) ‹invertible-simulation A A id.map› assms
        inverse-simulation.intro inverse-simulation.map-def someI-ex)
qed

lemma inverse-simulations-compose:
assumes inverse-simulations A B F' F and inverse-simulations B C G' G
shows inverse-simulations A C (F' ∘ G') (G ∘ F)
proof −
  interpret FF': inverse-simulations A B F' F
    using assms by blast
  interpret GG': inverse-simulations B C G' G
    using assms by blast
  interpret GoF: composite-simulation A B C F G ..
  interpret F'oG': composite-simulation C B A G' F' ..
  show ?thesis
  proof
    show GoF.map ∘ F'oG'.map = I C
      by (metis FF'.inv GG'.F.extensional GG'.F.preserves-reflects-arr
          GG'.inv comp-def)
    show F'oG'.map ∘ GoF.map = I A
      by (metis FF'.A.not-arr-null FF'.G.preserves-reflects-arr FF'.inv
          FF'.inv' GG'.inv' comp-apply)
  qed
qed

lemma invertible-simulation-comp [intro]:
assumes invertible-simulation A B F and invertible-simulation B C G
```

shows *invertible-simulation A C* (*G* ∘ *F*)
and *inverse-simulations A C*
(*inverse-simulation.map A B F* ∘ *inverse-simulation.map B C G*) (*G* ∘ *F*)
and *inverse-simulation.map A C* (*G* ∘ *F*) =
inverse-simulation.map A B F ∘ *inverse-simulation.map B C G*
proof −
obtain *F′* **where** *F′*: *inverse-simulations A B F′ F*
using *assms*(*1*) *invertible-simulation.invertible* **by** *blast*
interpret *FF′*: *inverse-simulations A B F′ F*
using *F′* **by** *blast*
obtain *G′* **where** *G′*: *inverse-simulations B C G′ G*
using *assms*(*2*) *invertible-simulation.invertible* **by** *blast*
interpret *GG′*: *inverse-simulations B C G′ G*
using *G′* **by** *blast*
interpret *GoF*: *composite-simulation A B C F G* **..**
interpret *F′oG′*: *composite-simulation C B A G′ F′* **..**
interpret *F′*: *inverse-simulation A B F*
using *F′ assms*(*1*) *inverse-simulation-def inverse-simulations-def* **by** *blast*
interpret *G′*: *inverse-simulation B C G*
using *G′ assms*(*2*)
by (*simp add*: *inverse-simulation.intro inverse-simulations-def*)
have *F′-eq*: *F′* = *F′.map*
using *F′ F′.inverse-simulations-axioms inverse-simulation-unique* **by** *blast*
have *G′-eq*: *G′* = *G′.map*
using *G′ G′.inverse-simulations-axioms inverse-simulation-unique* **by** *blast*
have *1*: *inverse-simulations A C* (*F′* ∘ *G′*) (*G* ∘ *F*)
using *F′ G′ inverse-simulations-compose* **by** *blast*
thus *2*: *invertible-simulation A C* (*G* ∘ *F*)
using *invertible-simulation-def* **by** *unfold-locales blast*
show *inverse-simulations A C* (*F′.map* ∘ *G′.map*) *GoF.map*
using *F′-eq G′-eq 1* **by** *blast*
show *inverse-simulation.map A C GoF.map* = *F′.map* ∘ *G′.map*
using *1 2 F′-eq G′-eq inverse-simulation-unique*
by (*metis FF′.A.rts-axioms GG′.B.rts-axioms inverse-simulation.intro*
inverse-simulation.map-def someI-ex)
qed

lemma *inverse-simulation-product* [*intro*]:
assumes *invertible-simulation A B F* **and** *invertible-simulation C D G*
shows *invertible-simulation* (*product-rts.resid A C*) (*product-rts.resid B D*)
(*product-simulation.map A C F G*)
and *inverse-simulations* (*product-rts.resid A C*) (*product-rts.resid B D*)
(*product-simulation.map B D*
(*inverse-simulation.map A B F*) (*inverse-simulation.map C D G*))
(*product-simulation.map A C F G*)
and *inverse-simulation.map* (*product-rts.resid A C*) (*product-rts.resid B D*)
(*product-simulation.map A C F G*) =
product-simulation.map B D
(*inverse-simulation.map A B F*) (*inverse-simulation.map C D G*)

```
proof −
  interpret A: rts A
    using assms(1) by (simp add: invertible-simulation-iff simulation-def)
  interpret B: rts B
    using assms(1) by (simp add: invertible-simulation-iff simulation-def)
  interpret C: rts C
    using assms(2) by (simp add: invertible-simulation-iff simulation-def)
  interpret D: rts D
    using assms(2) by (simp add: invertible-simulation-iff simulation-def)
  interpret AxC: product-rts A C ..
  interpret BxD: product-rts B D ..
  interpret F: simulation A B F
    using assms(1) invertible-simulation-def invertible-simulation-iff by blast
  interpret F': inverse-simulation A B F
    using assms(1) invertible-simulation.invertible by unfold-locales auto
  interpret FF': inverse-simulations A B F'.map F ..
  interpret G: simulation C D G
    using assms(2) invertible-simulation-def invertible-simulation-iff by blast
  interpret G': inverse-simulation C D G
    using assms(2) invertible-simulation.invertible by unfold-locales auto
  interpret GG': inverse-simulations C D G'.map G ..
  interpret FxG: product-simulation A C B D F G ..
  interpret F'xG': product-simulation B D A C F'.map G'.map ..
  interpret inverse-simulations AxC.resid BxD.resid F'xG'.map FxG.map
  proof
    show FxG.map ∘ F'xG'.map = I BxD.resid
    proof
      fix t
      show (FxG.map ∘ F'xG'.map) t = I BxD.resid t
        using F'.inv G'.inv FxG.map-def F'xG'.map-def
              F.extensional G.extensional
        by auto
    qed
    show F'xG'.map ∘ FxG.map = I AxC.resid
    proof
      fix t
      show (F'xG'.map ∘ FxG.map) t = I AxC.resid t
        using F'.inv' G'.inv' FxG.map-def F'xG'.map-def
              F'.extensional G'.extensional
        by auto
    qed
  qed
  show inverse-simulations AxC.resid BxD.resid F'xG'.map FxG.map ..
  show invertible-simulation AxC.resid BxD.resid FxG.map
    using inverse-simulations-axioms
    by unfold-locales blast
  thus inverse-simulation.map AxC.resid BxD.resid FxG.map = F'xG'.map
    by (metis inverse-simulation.intro inverse-simulation.map-def
      inverse-simulation-unique inverse-simulations-axioms invertible-simulation-def
```

simulation-def someI-ex)
qed

lemma *invertible-simulation-cancel-left*:
assumes *invertible-simulation A B H*
shows ⟦*simulation C A F*; *simulation C A G*; *H* ∘ *F* = *H* ∘ *G*⟧ ⟹ *F* = *G*
proof −
  obtain *K* **where** *K*: *inverse-simulations A B K H*
    using *assms*(*1*) *invertible-simulation.invertible* **by** *blast*
  interpret *HK*: *inverse-simulations A B K H*
    using *K* **by** *blast*
  show ⟦*simulation C A F*; *simulation C A G*; *H* ∘ *F* = *H* ∘ *G*⟧ ⟹ *F* = *G*
    by (*metis HK.inv' HOL.ext comp-def simulation.extensional*
        simulation.preserves-reflects-arr)
qed

lemma *invertible-simulation-cancel-right*:
assumes *invertible-simulation A B H*
shows ⟦*simulation B C F*; *simulation B C G*; *F* ∘ *H* = *G* ∘ *H*⟧ ⟹ *F* = *G*
proof −
  obtain *K* **where** *K*: *inverse-simulations A B K H*
    using *assms*(*1*) *invertible-simulation.invertible* **by** *blast*
  interpret *HK*: *inverse-simulations A B K H*
    using *K* **by** *blast*
  show ⟦*simulation B C F*; *simulation B C G*; *F* ∘ *H* = *G* ∘ *H*⟧ ⟹ *F* = *G*
    by (*metis HK.inv HOL.ext comp-def simulation.extensional*)
qed

definition *isomorphic-rts*
where *isomorphic-rts A B* ≡ ∃ *F G*. *inverse-simulations A B G F*

lemma *isomorphic-rts-reflexive*:
assumes *rts A*
shows *isomorphic-rts A A*
  using *assms invertible-simulation-identity isomorphic-rts-def* **by** *blast*

lemma *isomorphic-rts-symmetric*:
assumes *isomorphic-rts A B*
shows *isomorphic-rts B A*
  using *assms inverse-simulations-sym isomorphic-rts-def* **by** *meson*

lemma *isomorphic-rts-transitive* [*trans*]:
assumes *isomorphic-rts A B* **and** *isomorphic-rts B C*
shows *isomorphic-rts A C*
  using *assms invertible-simulation-comp*(*1*) *isomorphic-rts-def*
      invertible-simulation-def
  by (*metis inverse-simulations.axioms*(*4*) *invertible-simulation.invertible*
      invertible-simulation-axioms.intro)

lemma (**in** *simulation*) *simulation-inv-intoI*:
assumes *inj-on F* (*Collect A.arr*) **and** *F* ‘ (*Collect A.arr*) = *Collect B.arr*
and $\bigwedge t.\ \neg$ *B.arr t* $\Longrightarrow$ *inv-into* (*Collect A.arr*) *F t* = *A.null*
and $\bigwedge t\ u.$ *t* $\frown_B$ *u* $\Longrightarrow$
inv-into (*Collect A.arr*) *F t* $\frown_A$ *inv-into* (*Collect A.arr*) *F u*
shows *simulation B A* (*inv-into* (*Collect A.arr*) *F*)
proof
show $\bigwedge t.\ \neg$ *B.arr t* $\Longrightarrow$ *inv-into* (*Collect A.arr*) *F t* = *A.null*
using *assms*(*3*) **by** *blast*
show $\bigwedge t\ u.$ *t* $\frown_B$ *u* $\Longrightarrow$
inv-into (*Collect A.arr*) *F t* $\frown_A$ *inv-into* (*Collect A.arr*) *F u*
using *assms*(*4*) **by** *blast*
show $\bigwedge t\ u.$ *t* $\frown_B$ *u* $\Longrightarrow$
inv-into (*Collect A.arr*) *F* (*t* $\backslash_B$ *u*) =
inv-into (*Collect A.arr*) *F t* $\backslash_A$ *inv-into* (*Collect A.arr*) *F u*
by (*simp add*: *assms*(*1*−*2*,*4*) *f-inv-into-f inv-into-f-eq B.con-implies-arr*(*1*−*2*))
qed

1.2 Transformations

lemma (**in** *transformation*) *preserves-arr*:
assumes *A.arr t*
shows *B.arr* (τ *t*)
using *assms B.join-of-un-upto-cong B.prfx-implies-con naturality3* **by** *blast*

lemma (**in** *transformation*) *preserves-con*:
assumes *t* $\frown_A$ *u*
shows τ *t* $\frown_B$ τ *u* **and** τ *t* $\frown_B$ *F u*
proof −
show τ *t* $\frown_B$ τ *u*
proof (*intro B.con-with-join-of-iff*(*1*))
show *B.join-of* (τ (*A.src t*)) (*F t*) (τ *t*)
using *assms naturality3* [*of t*] *A.con-implies-arr*(*1*) **by** *blast*
show *F t* $\frown_B$ τ *u* $\wedge$ τ *u* $\backslash_B$ *F t* $\frown_B$ τ (*A.src t*) $\backslash_B$ *F t*
proof (*intro conjI*)
show τ *u* $\backslash_B$ *F t* $\frown_B$ τ (*A.src t*) $\backslash_B$ *F t*
proof −
have (τ *u* $\backslash_B$ *F t* $\frown_B$ τ (*A.src t*) $\backslash_B$ *F t*) $\longleftrightarrow$
(τ *u* $\backslash_B$ τ (*A.src t*) $\frown_B$ *F t* $\backslash_B$ τ (*A.src t*))
using *B.con-def B.cube* **by** *force*
also have ... $\longleftrightarrow$ τ *u* $\backslash_B$ τ (*A.src t*) $\frown_B$ *G t*
using *A.con-implies-arr*(*1*) *assms naturality2* **by** *force*
also have ... $\longleftrightarrow$ *G u* $\frown_B$ *G t*
by (*metis* (*mono-tags*, *lifting*) *A.con-imp-eq-src A.con-implies-arr*(*2*)
B.composite-ofE B.cong-subst-left(*1*) *B.join-ofE assms*
naturality2 naturality3)
also have ... = *True*
using *assms A.con-sym G.preserves-con* **by** *blast*
finally show *?thesis* **by** *blast*

```
      qed
      thus F t ⌢_B τ u
        by (metis B.con-def B.con-sym B.not-con-null(1))
    qed
  qed
  thus τ t ⌢_B F u
    using assms
    by (meson A.residuation-axioms B.con-sym B.con-with-join-of-iff(2)
        B.join-of-symmetric naturality3 residuation.con-implies-arr(2))
qed

lemma (in transformation) naturality1':
assumes A.arr t
shows B.composite-of (F t) (τ (A.trg t)) (τ t)
  using assms
  by (metis B.rts-axioms naturality1 naturality3 rts.join-ofE)

lemma (in transformation) naturality2':
assumes A.arr t
shows B.composite-of (τ (A.src t)) (G t) (τ t)
    by (metis B.join-ofE assms naturality2 naturality3)

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

  notation B.comp  (infixr ·_B 55)
  notation B.join  (infix ⊔_B 52)

  lemma naturality1'_E:
  shows F t ·_B τ (A.trg t) = τ t
  and A.arr t ⟹ B.composable (F t) (τ (A.trg t))
  proof −
    show F t ·_B τ (A.trg t) = τ t
      using naturality1' B.comp-is-composite-of(2)
      by (metis A.arr-trg-iff-arr B.comp-null(2) extensional)
    thus A.arr t ⟹ B.composable (F t) (τ (A.trg t))
      by (simp add: B.composable-iff-arr-comp preserves-arr)
  qed

  lemma naturality2'_E:
  shows τ (A.src t) ·_B G t = τ t
  and A.arr t ⟹ B.composable (τ (A.src t)) (G t)
  proof −
    show τ (A.src t) ·_B G t = τ t
      using naturality2' B.comp-is-composite-of(2)
      by (metis B.comp-null(2) G.extensional extensional)
    thus A.arr t ⟹ B.composable (τ (A.src t)) (G t)
```

by (*simp add*: *B.composable-iff-arr-comp preserves-arr*)
qed

lemma *naturality3'*$_E$:
shows $\tau\ (A.src\ t) \sqcup_B F\ t = \tau\ t$
and $A.arr\ t \Longrightarrow B.joinable\ (\tau\ (A.src\ t))\ (F\ t)$
proof −
  show $\tau\ (A.src\ t) \sqcup_B F\ t = \tau\ t$
    using *naturality3*
    by (*metis A.not-arr-null A.src-def B.join-def B.join-is-join-of*
        B.join-of-unique B.joinable-def B.joinable-implies-con
        B.not-arr-null F.extensional extensional B.arrI)
  thus $A.arr\ t \Longrightarrow B.joinable\ (\tau\ (A.src\ t))\ (F\ t)$
    by (*metis B.joinable-iff-arr-join preserves-arr*)
qed

lemma *naturality*$_E$:
shows $\tau\ (A.src\ t) \cdot_B G\ t = F\ t \cdot_B \tau\ (A.trg\ t)$
  using *naturality1'*$_E(1)$ *naturality2'*$_E(1)$ **by** *presburger*

lemma *general-naturality*:
assumes *A.con x y*
shows $\tau\ x \setminus_B F\ y = \tau\ (x \setminus_A y)$
and $F\ x \setminus_B \tau\ y = G\ (x \setminus_A y)$
proof −
  show $\tau\ x \setminus_B F\ y = \tau\ (x \setminus_A y)$
    using *assms*
    by (*metis* (*full-types*) *A.residuation-axioms A.weakly-extensional-rts-axioms*
     B.extensional-rts-axioms G.simulation-axioms extensional-rts.resid-comp(*2*)
      naturality2'$_E(1)$ *residuation.con-implies-arr*(*2*) *simulation.preserves-resid*
       transformation.naturality1-ax transformation.naturality2-ax
       transformation.preserves-con(*2*) *transformation-axioms*
       weakly-extensional-rts.con-imp-eq-src weakly-extensional-rts.src-resid)
  show $F\ x \setminus_B \tau\ y = G\ (x \setminus_A y)$
    using *assms*
    by (*metis A.con-sym A.src-resid B.resid-comp*(*1*) *F.preserves-resid*
        naturality1'$_E(1)$ *naturality2 preserves-con*(*2*))
qed

lemma *preserves-prfx*:
assumes $t \lesssim_A u$
shows $\tau\ t \lesssim_B \tau\ u$
proof −
  have $\tau\ t \setminus_B \tau\ u = B.trg\ (\tau\ (A.src\ t)) \setminus_B (F\ u \setminus_B \tau\ (A.src\ t)) \sqcup_B$
                    $B.trg\ (F\ u) \setminus_B (\tau\ (A.src\ t) \setminus_B F\ u)$
  proof −
    have $\tau\ t \setminus_B \tau\ u = (\tau\ (A.src\ t) \sqcup_B F\ t) \setminus_B (\tau\ (A.src\ t) \sqcup_B F\ u)$
      by (*metis A.con-imp-eq-src A.prfx-implies-con assms naturality3'*$_E(1)$)

```
    also have ... = (τ (A.src t) \_B (τ (A.src t) ⊔_B F u)) ⊔_B
                    (F t \_B (τ (A.src t) ⊔_B F u))
      by (metis assms calculation A.con-implies-arr(2) A.con-sym
          A.prfx-implies-con B.arr-resid-iff-con B.con-sym B.resid-join_E(3)
          naturality3'_E(2) preserves-con(1))
    also have ... = (τ (A.src t) \_B τ (A.src t)) \_B (F u \_B τ (A.src t)) ⊔_B
                    (F t \_B F u) \_B (τ (A.src t) \_B F u)
      by (metis (full-types) assms calculation A.prfx-implies-con
          B.arr-prfx-join-self B.conE B.conI B.join-def B.prfx-implies-con
          B.resid-join_E(1-2) B.null-is-zero(2) preserves-con(1))
    also have ... = B.trg (τ (A.src t)) \_B (F u \_B τ (A.src t)) ⊔_B
                    B.trg (F u) \_B (τ (A.src t) \_B F u)
      by (metis A.arr-resid-iff-con A.con-implies-arr(2) A.ide-implies-arr
          A.src-ide A.src-resid B.trg-def F.preserves-resid F.preserves-trg
          assms)
    finally show ?thesis by blast
  qed
  moreover have B.ide ...
  proof -
    have B.ide (B.trg (τ (A.src t)) \_B (F u \_B τ (A.src t)))
      by (metis assms A.con-imp-eq-src A.con-implies-arr(2) A.cong-reflexive
          A.ide-src A.prfx-implies-con A.resid-src-arr A.trg-def naturality2
          preserves-trg G.preserves-prfx)
    moreover have B.ide (B.trg (F u) \_B (τ (A.src t) \_B F u))
      using B.apex-sym B.cube B.trg-def calculation by force
    ultimately show ?thesis
      by (metis B.apex-sym B.ide-iff-src-self B.ide-implies-arr
          B.join-arr-self B.prfx-implies-con B.src-resid)
  qed
  ultimately show ?thesis by simp
qed

end



locale transformation-by-components =
  A: weakly-extensional-rts A +
  B: extensional-rts B +
  F: simulation A B F +
  G: simulation A B G
for A :: 'a resid      (infix \_A 55)
and B :: 'b resid      (infix \_B 55)
and F :: 'a ⇒ 'b
and G :: 'a ⇒ 'b
and τ :: 'a ⇒ 'b +
assumes preserves-src: A.ide a ⟹ B.src (τ a) = F a
and preserves-trg: A.ide a ⟹ B.trg (τ a) = G a
and naturality1: A.arr t ⟹ τ (A.src t) \_B F t = τ (A.trg t)
```

and *naturality2*: $A.arr\ t \Longrightarrow F\ t \setminus_B \tau\ (A.src\ t) = G\ t$
and *joinable*: $A.arr\ t \Longrightarrow B.joinable\ (\tau\ (A.src\ t))\ (F\ t)$
begin

notation $B.comp$ (**infixr** $\cdot_B$ *55*)
notation $B.join$ (**infix** $\sqcup_B$ *52*)

definition *map*
where $map\ t = \tau\ (A.src\ t) \sqcup_B F\ t$

lemma *map-eq*:
shows $map\ t = (if\ A.arr\ t\ then\ \tau\ (A.src\ t) \sqcup_B F\ t\ else\ B.null)$
unfolding *map-def*
by (*metis B.conE B.join-def B.joinable-implies-con B.null-is-zero*(*2*) *F.extensional*)

lemma *map-simp-ide* [*simp*]:
assumes *A.ide a*
shows $map\ a = \tau\ a$
using *assms map-def*
by (*metis A.ide-iff-src-self A.ide-implies-arr B.arr-trg-iff-arr B.join-src B.join-sym G.preserves-ide preserves-src preserves-trg B.ide-implies-arr*)

sublocale *transformation A B F G map*
using *map-eq preserves-src preserves-trg naturality1 naturality2 joinable*
apply (*unfold-locales*, *simp-all*)
apply (*metis A.ide-implies-arr A.src-ide B.src-join*)
apply (*metis A.ide-implies-arr A.src-ide A.trg-ide B.trg-join*)
apply (*metis A.arr-src-iff-arr A.arr-trg-iff-arr A.ide-src A.src-src A.src-trg map-simp-ide*)
apply (*metis A.arr-src-iff-arr A.ide-src A.src-src map-simp-ide*)
by (*metis A.arr-src-iff-arr A.ide-src A.src-src B.join-is-join-of map-simp-ide*)

lemma *is-transformation*:
shows *transformation A B F G map*
..

end

locale *identity-transformation* =
transformation +
assumes *identity*: $A.ide\ a \Longrightarrow B.ide\ (\tau\ a)$
begin

lemma *src-eq-trg*:
shows $F = G$
proof
fix *x*
show $F\ x = G\ x$

```
    by (metis A.ide-src B.ideE B.resid-arr-src B.src-eqI B.src-join-of(1−2)
      F.extensional F.preserves-reflects-arr G.extensional identity
      naturality3 naturality2)
  qed

  sublocale simulation ..

end

lemma comp-identity-transformation:
assumes transformation A B F G T
shows I B ∘ T = T
  using assms transformation.extensional transformation.preserves-arr
  by fastforce

lemma comp-transformation-identity:
assumes transformation A B F G T
shows T ∘ I A = T
  using assms residuation.not-arr-null rts.axioms(1) transformation.extensional
      transformation-def simulation-def
  by fastforce

locale constant-transformation =
  A: weakly-extensional-rts A +
  B: weakly-extensional-rts B
for A :: 'a resid      (infix \_A 70)
and B :: 'b resid      (infix \_B 70)
and t :: 'b +
assumes arr-t: B.arr t
begin

  abbreviation map
  where map x ≡ if A.arr x then t else B.null

  abbreviation F
  where F ≡ constant-simulation.map A B (B.src t)

  abbreviation G
  where G ≡ constant-simulation.map A B (B.trg t)

  interpretation F: simulation A B F
    using arr-t A.con-implies-arr B.resid-arr-ide
    by unfold-locales auto

  interpretation G: simulation A B G
    using arr-t A.con-implies-arr B.resid-arr-ide
    by unfold-locales auto

  sublocale transformation A B F G map
```

```
    using arr-t B.join-of-arr-src(2)
    by unfold-locales auto

end

locale simulation-as-transformation =
  simulation +
  A: weakly-extensional-rts A +
  B: weakly-extensional-rts B
begin

  sublocale transformation A B F F F
    using extensional A.con-arr-src(1−2) B.join-of-arr-src(1)
    apply unfold-locales
    subgoal by simp
    subgoal by simp
    subgoal by simp
    subgoal by simp (metis A.con-arr-src(2) A.resid-src-arr preserves-resid)
    subgoal by simp (metis A.con-arr-src(1) A.resid-arr-src preserves-resid)
    subgoal by simp
    done

  sublocale identity-transformation A B F F F
    by unfold-locales auto

end

lemma transformation-eqI:
assumes transformation A B F G σ and transformation A B F G τ
and extensional-rts B
and ⋀a. residuation.ide A a ⟹ σ a = τ a
shows σ = τ
proof
  interpret A: weakly-extensional-rts A
    using assms(1) simulation.axioms(1) transformation-def by blast
  interpret B: extensional-rts B
    using assms(3) by simp
  interpret F: simulation A B F
    using assms(1) transformation.axioms(3) by blast
  interpret G: simulation A B G
    using assms(1) transformation.axioms(4) by blast
  interpret σ: transformation A B F G σ
    using assms(1) by blast
  interpret τ: transformation A B F G τ
    using assms(2) by blast
  fix t
  show σ t = τ t
    by (metis A.ide-src B.join-of-unique σ.extensional σ.naturality3
        τ.extensional τ.naturality3 assms(4))
```

qed

lemma *invertible-simulation-cancel-left'*:
assumes *invertible-simulation A B H*
shows ⟦*transformation C A F G S*; *transformation C A F G T*;
H ∘ S = H ∘ T⟧
⟹ *S = T*
proof −
obtain *K* **where** *K*: *inverse-simulations A B K H*
using *assms*(*1*) *invertible-simulation.invertible* **by** *blast*
interpret *HK*: *inverse-simulations A B K H*
using *K* **by** *blast*
show ⟦*transformation C A F G S*; *transformation C A F G T*;
H ∘ S = H ∘ T⟧
⟹ *S = T*
by (*metis HK.inv' HOL.ext comp-def transformation.extensional*
transformation.preserves-arr)
qed

lemma *invertible-simulation-cancel-right'*:
assumes *invertible-simulation A B H*
shows ⟦*transformation B C F G S*; *transformation B C F G T*;
S ∘ H = T ∘ H⟧
⟹ *S = T*
proof −
obtain *K* **where** *K*: *inverse-simulations A B K H*
using *assms*(*1*) *invertible-simulation.invertible* **by** *blast*
interpret *HK*: *inverse-simulations A B K H*
using *K* **by** *blast*
show ⟦*transformation B C F G S*; *transformation B C F G T*;
S ∘ H = T ∘ H⟧
⟹ *S = T*
by (*metis HK.inv HOL.ext comp-def transformation.extensional*)
qed

1.3 Binary Simulations

locale *binary-simulation-between-weakly-extensional-rts* =
binary-simulation +
A1: *weakly-extensional-rts A1* +
A0: *weakly-extensional-rts A0* +
B: *weakly-extensional-rts B*
begin

interpretation *A*: *product-of-weakly-extensional-rts A1 A0* **..**
sublocale *simulation-to-weakly-extensional-rts A.resid B F* **..**

lemma *fixing-arr-gives-transformation-1*:
assumes *A1.arr t1*

```
shows transformation A0 B
        (λt0. F (A1.src t1, t0)) (λt0. F (A1.trg t1, t0))
        (λt0. F (t1, t0))
proof −
  interpret Fsrc: simulation A0 B ‹λt0. F (A1.src t1, t0)›
    using assms fixing-ide-gives-simulation-1 by simp
  interpret Fsrc: simulation-to-weakly-extensional-rts A0 B
                ‹λt0. F (A1.src t1, t0)›
    ..
  interpret Ftrg: simulation A0 B ‹λt0. F (A1.trg t1, t0)›
    using assms fixing-ide-gives-simulation-1 by simp
  interpret Ftrg: simulation-to-weakly-extensional-rts A0 B
                ‹λt0. F (A1.trg t1, t0)›
    ..
  show transformation A0 B
        (λt0. F (A1.src t1, t0)) (λt0. F (A1.trg t1, t0))
        (λt0. F (t1, t0))
  proof
    show ⋀f. ¬ A0.arr f ⟹ F (t1, f) = B.null
      by (simp add: extensional)
    show ⋀f. A0.ide f ⟹ B.src (F (t1, f)) = F (A1.src t1, f)
      using assms B.src-eqI by simp
    show ⋀f. A0.ide f ⟹ B.trg (F (t1, f)) = F (A1.trg t1, f)
      by (metis assms fixing-ide-gives-simulation-0 simulation.preserves-trg)
    fix f
    assume f: A0.arr f
    show F (t1, A0.src f) \B F (A1.src t1, f) = F (t1, A0.trg f)
    proof −
      have F (t1, A0.src f) \B F (A1.src t1, f) =
           F (A.resid (t1, A0.src f) (A1.src t1, f))
        by (simp add: assms f)
      also have ... = F (t1, A0.trg f)
        by (simp add: assms f A.resid-def)
      finally show ?thesis by blast
    qed
    show F (A1.src t1, f) \B F (t1, A0.src f) = F (A1.trg t1, f)
    proof −
      have F (A1.src t1, f) \B F (t1, A0.src f) =
           F (A.resid (A1.src t1, f) (t1, A0.src f))
        by (simp add: assms f)
      also have ... = F (A1.trg t1, f)
        by (simp add: assms f A.resid-def)
      finally show ?thesis by blast
    qed
    show B.join-of (F (t1, A0.src f)) (F (A1.src t1, f)) (F (t1, f))
      by (simp add: f A1.join-of-arr-src(2) A0.join-of-arr-src(1)
          assms preserves-joins A.join-of-char(1))
  qed
qed
```

```
  lemma fixing-arr-gives-transformation-0:
  assumes A0.arr t2
  shows transformation A1 B
          (λt1. F (t1, A0.src t2)) (λt1. F (t1, A0.trg t2))
          (λt1. F (t1, t2))
  proof −
    interpret Fsrc: simulation A1 B ‹λt1. F (t1, A0.src t2)›
      using assms fixing-ide-gives-simulation-0 by simp
    interpret Fsrc: simulation-to-weakly-extensional-rts A1 B
      ‹λt1. F (t1, A0.src t2)›
      ..
    interpret Ftrg: simulation A1 B ‹λt1. F (t1, A0.trg t2)›
      using assms fixing-ide-gives-simulation-0 by simp
    interpret Ftrg: simulation-to-weakly-extensional-rts A1 B
                    ‹λt1. F (t1, A0.trg t2)›
      ..
    show transformation A1 B
          (λt1. F (t1, A0.src t2)) (λt1. F (t1, A0.trg t2))
          (λt1. F (t1, t2))
    proof
      show ⋀f. ¬ A1.arr f ⟹ F (f, t2) = B.null
        by (simp add: extensional)
      show ⋀f. A1.ide f ⟹ B.src (F (f, t2)) = F (f, A0.src t2)
        using assms B.src-eqI by simp
      show ⋀f. A1.ide f ⟹ B.trg (F (f, t2)) = F (f, A0.trg t2)
        by (metis assms fixing-ide-gives-simulation-1 simulation.preserves-trg)
      fix f
      assume f: A1.arr f
      show F (A1.src f, t2) \_B F (f, A0.src t2) = F (A1.trg f, t2)
        using f fixing-arr-gives-transformation-1 transformation.naturality2
        by fastforce
      show F (f, A0.src t2) \_B F (A1.src f, t2) = F (f, A0.trg t2)
        by (metis assms f fixing-arr-gives-transformation-1
            transformation.naturality1-ax)
      show ⋀f. A1.arr f ⟹
                B.join-of (F (A1.src f, t2)) (F (f, A0.src t2)) (F (f, t2))
        by (simp add: A1.join-of-arr-src(1) A0.join-of-arr-src(2)
            assms preserves-joins A.join-of-char(1))
    qed
  qed

end

locale transformation-of-binary-simulations =
  A1: weakly-extensional-rts A1 +
  A0: weakly-extensional-rts A0 +
  B: weakly-extensional-rts B +
  A1xA0: product-rts A1 A0 +
```

```
  F: binary-simulation A1 A0 B F +
  G: binary-simulation A1 A0 B G +
  transformation A1xA0.resid B F G τ
for A1 :: 'a1 resid      (infix \<^sub>A1 55)
and A0 :: 'a0 resid      (infix \<^sub>A0 55)
and B :: 'b resid        (infix \<^sub>B 55)
and F :: 'a1 * 'a0 ⇒ 'b
and G :: 'a1 * 'a0 ⇒ 'b
and τ :: 'a1 * 'a0 ⇒ 'b
begin

  notation A0.con      (infix ⌢A0 50)
  notation A0.prfx     (infix ≲A0 50)
  notation A0.cong     (infix ∼A0 50)

  notation A1.con      (infix ⌢A1 50)
  notation A1.prfx     (infix ≲A1 50)
  notation A1.cong     (infix ∼A1 50)

  notation B.con       (infix ⌢B 50)
  notation B.prfx      (infix ≲B 50)
  notation B.cong      (infix ∼B 50)

  notation A1xA0.resid     (infix \<^sub>A1xA0 55)

  sublocale A1xA0: product-of-weakly-extensional-rts A1 A0 ..

  lemma fixing-ide-gives-transformation-1:
  assumes A1.ide a1
  shows transformation A0 B (λf0. F (a1, f0)) (λf0. G (a1, f0))
          (λf0. τ (a1, f0))
  proof −
    interpret Fa1: simulation A0 B ‹λf0. F (a1, f0)›
      using assms F.fixing-ide-gives-simulation-1 by simp
    interpret Ga1: simulation A0 B ‹λf0. G (a1, f0)›
      using assms G.fixing-ide-gives-simulation-1 by simp
    show ?thesis
    proof
      fix f
      show ¬ A0.arr f ⟹ τ (a1, f) = B.null
        by (simp add: extensional)
      show A0.ide f ⟹ B.src (τ (a1, f)) = F (a1, f)
        using assms preserves-src by force
      show A0.ide f ⟹ B.trg (τ (a1, f)) = G (a1, f)
        by (simp add: assms preserves-trg)
      show A0.arr f ⟹ τ (a1, A0.src f) \<^sub>B F (a1, f) = τ (a1, A0.trg f)
        by (metis A1.ide-iff-src-self A1.ide-iff-trg-self A1.ide-implies-arr
            A1xA0.src-char A1xA0.trg-char F.preserves-reflects-arr
            Fa1.preserves-reflects-arr assms fst-conv naturality1 snd-conv)
```

```
    show A0.arr f ⟹ F (a1, f) \_B τ (a1, A0.src f) = G (a1, f)
      by (metis assms A1.ide-iff-src-self A1.ide-implies-arr A1xA0.src-char
          G.preserves-reflects-arr Ga1.preserves-reflects-arr fst-conv
          naturality2 snd-conv)
    show A0.arr f ⟹ B.join-of (τ (a1, A0.src f)) (F (a1, f)) (τ (a1, f))
      by (metis assms A1.ide-iff-src-self A1.ide-implies-arr A1xA0.src-char
          G.preserves-reflects-arr Ga1.preserves-reflects-arr fst-conv naturality3
          snd-conv)
  qed
qed

lemma fixing-ide-gives-transformation-0:
assumes A0.ide a0
shows transformation A1 B (λf1. F (f1, a0)) (λf1. G (f1, a0))
        (λf1. τ (f1, a0))
proof −
  interpret Fa0: simulation A1 B ‹λf1. F (f1, a0)›
    using assms F.fixing-ide-gives-simulation-0 by simp
  interpret Ga0: simulation A1 B ‹λf1. G (f1, a0)›
    using assms G.fixing-ide-gives-simulation-0 by simp
  show ?thesis
  proof
    fix f
    show ¬ A1.arr f ⟹ τ (f, a0) = B.null
      by (simp add: extensional)
    show A1.ide f ⟹ B.src (τ (f, a0)) = F (f, a0)
      by (simp add: assms preserves-src)
    show A1.ide f ⟹ B.trg (τ (f, a0)) = G (f, a0)
      by (simp add: assms preserves-trg)
    show A1.arr f ⟹ τ (A1.src f, a0) \_B F (f, a0) = τ (A1.trg f, a0)
      by (metis A1xA0.src-char A1xA0.trg-char A0.ide-iff-src-self
          A0.ide-iff-trg-self A0.ide-implies-arr F.preserves-reflects-arr
          Fa0.preserves-reflects-arr assms fst-conv naturality1 snd-conv)
    show A1.arr f ⟹ F (f, a0) \_B τ (A1.src f, a0) = G (f, a0)
      by (metis A1xA0.src-char A0.ide-iff-src-self A0.ide-implies-arr
          F.preserves-reflects-arr Fa0.preserves-reflects-arr assms
          fst-conv naturality2 snd-conv)
    show A1.arr f ⟹ B.join-of (τ (A1.src f, a0)) (F (f, a0)) (τ (f, a0))
      by (metis A1xA0.src-char A0.ide-iff-src-self A0.ide-implies-arr
          F.preserves-reflects-arr Fa0.preserves-reflects-arr assms
          fst-conv naturality3 snd-conv)
  qed
qed

end
```

1.4 Horizontal Composite of Transformations

```
lemma transformation-whisker-left:
```

assumes *transformation A B F G τ* **and** *simulation B C H*
and *weakly-extensional-rts C*
shows *transformation A C* $(H \circ F)$ $(H \circ G)$ $(H \circ \tau)$
proof −
  interpret *C*: *weakly-extensional-rts C*
    using *assms*(*3*) **by** *blast*
  interpret *τ*: *transformation A B F G τ*
    using *assms* **by** *blast*
  interpret *H*: *simulation B C H*
    using *assms* **by** *blast*
  interpret *HoF*: *composite-simulation A B C F H* **..**
  interpret *HoG*: *composite-simulation A B C G H* **..**
  show *?thesis*
  proof
    fix *t*
    show $\neg$ *τ.A.arr t* $\Longrightarrow$ $(H \circ \tau)$ *t* = *C.null*
      by (*simp add*: *H.extensional τ.extensional*)
    show *τ.A.ide t* $\Longrightarrow$ *C.src* $((H \circ \tau)$ *t*$)$ = *HoF.map t*
      by (*metis C.src-eqI H.preserves-con HoF.preserves-ide τ.A.ide-implies-arr*
          τ.B.con-arr-src(*2*) *τ.preserves-arr τ.preserves-src comp-eq-dest-lhs*)
    show *τ.A.ide t* $\Longrightarrow$ *C.trg* $((H \circ \tau)$ *t*$)$ = *HoG.map t*
      by (*metis H.preserves-trg τ.A.ide-implies-arr τ.preserves-arr*
          τ.preserves-trg comp-apply)
    show *τ.A.arr t* $\Longrightarrow$
          C $((H \circ \tau)$ $(\tau.A.src\ t))$ *(HoF.map t)* = $(H \circ \tau)$ $(\tau.A.trg\ t)$
      by (*metis H.preserves-resid τ.B.conI τ.B.con-sym-ax τ.B.not-arr-null*
          τ.G.preserves-reflects-arr τ.naturality1-ax τ.naturality2-ax comp-apply)
    show *τ.A.arr t* $\Longrightarrow$ *C* *(HoF.map t)* $((H \circ \tau)$ $(\tau.A.src\ t))$ = *HoG.map t*
      by (*metis H.preserves-resid τ.B.arr-resid-iff-con τ.G.preserves-reflects-arr*
          τ.naturality2-ax comp-apply)
    show *τ.A.arr t* $\Longrightarrow$ *C.join-of* $((H \circ \tau)$ $(\tau.A.src\ t))$ *(HoF.map t)* $((H \circ \tau)$ *t*$)$
      by (*metis H.simulation-axioms τ.naturality3 comp-apply*
          simulation.preserves-joins)
  qed
qed

lemma *transformation-whisker-right*:
assumes *transformation B C F G τ* **and** *simulation A B H*
and *weakly-extensional-rts A*
shows *transformation A C* $(F \circ H)$ $(G \circ H)$ $(\tau \circ H)$
proof −
  interpret *A*: *weakly-extensional-rts A*
    using *assms*(*3*) **by** *blast*
  interpret *τ*: *transformation B C F G τ*
    using *assms* **by** *blast*
  interpret *H*: *simulation A B H*
    using *assms* **by** *blast*
  interpret *FoH*: *composite-simulation A B C H F* **..**
  interpret *GoH*: *composite-simulation A B C H G* **..**

```
  show ?thesis
  proof
    fix t
    show ¬ A.arr t ⟹ (τ ∘ H) t = τ.B.null
      by (simp add: τ.extensional)
    show A.ide t ⟹ τ.B.src ((τ ∘ H) t) = FoH.map t
      by (simp add: τ.preserves-src)
    show A.ide t ⟹ τ.B.trg ((τ ∘ H) t) = GoH.map t
      by (simp add: τ.preserves-trg)
    show A.arr t ⟹ C ((τ ∘ H) (A.src t)) (FoH.map t) = (τ ∘ H) (A.trg t)
      by (metis A.arr-has-un-source A.arr-src-iff-arr A.con-arr-src(1)
          A.src-in-sources A.trg-src H.preserves-con H.preserves-trg
          τ.A.con-imp-eq-src τ.A.src-trg τ.naturality1 comp-def)
    show A.arr t ⟹ C (FoH.map t) ((τ ∘ H) (A.src t)) = GoH.map t
      by (metis A.con-arr-src(2) A.ide-src H.preserves-con H.preserves-ide
          τ.A.con-imp-eq-src τ.naturality2 comp-apply τ.A.src-ide)
    show A.arr t ⟹ τ.B.join-of ((τ ∘ H) (A.src t)) (FoH.map t) ((τ ∘ H) t)
      by (metis (full-types) A.con-arr-src(2) A.ide-src FoH.preserves-reflects-arr
          H.preserves-con H.preserves-ide τ.B.not-arr-null
          τ.F.extensional τ.naturality3 comp-apply τ.A.con-imp-eq-src τ.A.src-ide)
  qed
qed
```

Horizontal composition of transformations requires reasoning about joins which it is not clear that it is possible to do unless extensionality is assumed.

```
lemma horizontal-composite:
assumes transformation B C F G σ and transformation A B H K τ
and extensional-rts A and extensional-rts B and extensional-rts C
shows transformation A C (F ∘ H) (G ∘ K) (σ ∘ τ)
proof −
  interpret A: extensional-rts A
    using assms(3) by blast
  interpret B: extensional-rts B
    using assms(4) by blast
  interpret C: extensional-rts C
    using assms(5) by blast
  interpret σ: transformation B C F G σ
    using assms by blast
  interpret σ: transformation-to-extensional-rts B C F G σ ..
  interpret τ: transformation A B H K τ
    using assms by blast
  interpret τ: transformation-to-extensional-rts A B H K τ ..
  interpret FoH: composite-simulation A B C H F ..
  interpret GoH: composite-simulation A B C H G ..
  interpret FoK: composite-simulation A B C K F ..
  interpret GoK: composite-simulation A B C K G ..
  show ?thesis
  proof
    write A  (infix \_A 55)
```

```
    write B  (infix \<^sub>B 55)
    write C  (infix \<^sub>C 55)
    write B.join  (infixr ⊔⇩B 52)
    write C.join  (infixr ⊔⇩C 52)
    fix t
    show ¬ A.arr t ⟹ (σ ∘ τ) t = C.null
      by (simp add: σ.extensional τ.extensional)
    show A.ide t ⟹ C.src ((σ ∘ τ) t) = FoH.map t
      by (metis A.arr-def A.ideE C.weakly-extensional-rts-axioms σ.naturality3
        σ.transformation-axioms τ.F.preserves-ide τ.preserves-arr τ.preserves-src
        comp-apply transformation.preserves-src weakly-extensional-rts.src-join-of(1))
    show A.ide t ⟹ C.trg ((σ ∘ τ) t) = GoK.map t
      by (metis A.arr-resid-iff-con A.ideE C.apex-sym C.trg-join-of(1)
          σ.G.preserves-trg σ.naturality2 σ.naturality3 τ.preserves-arr
          τ.preserves-trg comp-apply)
    assume t: A.arr t
    show (σ ∘ τ) (A.src t) \<^sub>C FoH.map t = (σ ∘ τ) (A.trg t)
      by (simp add: σ.general-naturality(1) τ.naturality1-ax
          τ.preserves-con(2) t)
    show FoH.map t \<^sub>C (σ ∘ τ) (A.src t) = GoK.map t
      by (simp add: B.con-sym σ.general-naturality(2) τ.naturality2-ax
          τ.preserves-con(2) t)
    show C.join-of ((σ ∘ τ) (A.src t)) (FoH.map t) ((σ ∘ τ) t)
    proof −
      have σ (τ (A.src t)) ⊔⇩C F (H t) = σ (τ t)
      proof (intro C.join-eqI)
        show 1: C.prfx (σ (τ (A.src t))) (σ (τ t))
          using t τ.preserves-prfx σ.preserves-prfx by simp
        show 2: C.prfx (F (H t)) (σ (τ t))
          using t
          by (meson C.composite-ofE C.prfx-transitive σ.F.preserves-composites
              σ.naturality1' τ.naturality1' τ.preserves-arr)
        show 3: σ (τ t) \<^sub>C F (H t) = σ (τ (A.src t)) \<^sub>C F (H t)
          by (simp add: A.trg-def σ.general-naturality(1) τ.general-naturality(1)
              τ.preserves-con(2) t)
        show σ (τ t) \<^sub>C σ (τ (A.src t)) = F (H t) \<^sub>C σ (τ (A.src t))
          using t 1 2 3 C.apex-arr-prfx(1) C.apex-sym C.cube C.extensional
          C.ideE C.trg-def
          by (smt (verit, del-insts))
      qed
      thus ?thesis
        using t
        by (metis C.join-is-join-of C.joinable-iff-arr-join σ.preserves-arr
            τ.preserves-arr comp-apply)
    qed
  qed
qed

end
```

Chapter 2

RTS's as Categories

As shown in the previous article [4], every RTS extends to an extensional RTS that has a composite for each pair of composable transitions. Such an RTS may be regarded as a category, and in this section we establish a characterization of the kind of categories that are obtained from RTS's in this way.

2.1 Categories with Bounded Pushouts

2.1.1 Bounded Spans

We call a span in a category "bounded" if it can be completed to a commuting square. A category with bounded pushouts is a category in which every bounded span has a pushout.

theory *CategoryWithBoundedPushouts*
imports *Category3.EpiMonoIso Category3.CategoryWithPullbacks*
begin

context *category*
begin

definition *bounded-span*
where *bounded-span h k* ≡ ∃*f g*. *commutative-square f g h k*

lemma *bounded-spanI* [*intro*]:
assumes *commutative-square f g h k*
shows *bounded-span h k*
using *assms bounded-span-def* **by** *auto*

lemma *bounded-spanE* [*elim*]:
assumes *bounded-span h k*
obtains *f g* **where** *commutative-square f g h k*
using *assms bounded-span-def* **by** *auto*

lemma *bounded-span-sym*:
shows *bounded-span h k* $\Longrightarrow$ *bounded-span k h*
unfolding *bounded-span-def commutative-square-def*
by (*metis seqE seqI*)

end

2.1.2 Pushouts

Here we give a definition of the notion "pushout square" in a category, and prove that pushout squares compose. The definition here is currently a "free-standing" one, because it has been stated on its own, without deriving it from a general notion of colimit. At some future time, once the general development of limits given in [2] has been suitably dualized to obtain a corresponding development of colimits, this formal connection should be made.

context *category*
begin

definition *pushout-square*
where *pushout-square f g h k* $\equiv$
commutative-square f g h k $\wedge$
$(\forall f'\ g'.$ *commutative-square* $f'\ g'\ h\ k \longrightarrow (\exists!l.\ l \cdot f = f' \wedge l \cdot g = g'))$

lemma *pushout-squareI* [*intro*]:
assumes *cospan f g* **and** *span h k* **and** *dom f* $=$ *cod h* **and** $f \cdot h = g \cdot k$
and $\bigwedge f'\ g'.$ *commutative-square* $f'\ g'\ h\ k \Longrightarrow \exists!l.\ l \cdot f = f' \wedge l \cdot g = g'$
shows *pushout-square f g h k*
using *assms pushout-square-def* **by** *simp*

lemma *composition-of-pushouts*:
assumes *pushout-square* $u'\ t'\ t\ u$ **and** *pushout-square* $v'\ t''\ t'\ v$
shows *pushout-square* $(v' \cdot u')\ t''\ t\ (v \cdot u)$
proof
show *1*: *cospan* $(v' \cdot u')\ t''$
using *assms*
by (*metis* (*mono-tags*, *lifting*) *commutative-square-def seqI cod-comp pushout-square-def*)
show *span* $t\ (v \cdot u)$
using *assms pushout-square-def* **by** *fastforce*
show *dom* $(v' \cdot u') =$ *cod* t
using *assms*
by (*metis 1 commutative-squareE dom-comp pushout-square-def*)
show $(v' \cdot u') \cdot t = t'' \cdot v \cdot u$
using *assms*
by (*metis commutative-squareE comp-assoc pushout-square-def*)
fix *w x*
assume *wx*: *commutative-square w x t* $(v \cdot u)$

show $\exists!l.\ l \cdot v' \cdot u' = w \wedge l \cdot t'' = x$
proof −
have *1*: *commutative-square* w $(x \cdot v)$ t u
using *wx*
by (*metis* (*mono-tags*, *lifting*) *cod-comp commutative-square-def dom-comp comp-assoc seqE seqI*)
hence $*$: $\exists!z.\ z \cdot u' = w \wedge z \cdot t' = x \cdot v$
using *assms pushout-square-def* **by** *auto*
obtain z **where** *z*: $z \cdot u' = w \wedge z \cdot t' = x \cdot v$
using $*$ **by** *auto*
have *2*: *commutative-square* z x t' v
using *z*
by (*metis* (*mono-tags*, *lifting*) *1 cod-comp commutative-square-def dom-comp seqE*)
hence $**$: $\exists\, l.\ l \cdot v' = z \wedge l \cdot t'' = x$
by (*meson assms*(*2*) *pushout-square-def*)
obtain l **where** *l*: $l \cdot v' = z \wedge l \cdot t'' = x$
using $**$ **by** *auto*
have $l \cdot v' \cdot u' = w \wedge l \cdot t'' = x$
using *l comp-assoc z* **by** *force*
moreover have $\bigwedge l\ l'.\ [\![l \cdot v' \cdot u' = w \wedge l \cdot t'' = x;\ l' \cdot v' \cdot u' = w \wedge l' \cdot t'' = x]\!] \Longrightarrow l = l'$
proof −
fix l l'
assume *l*: $l \cdot v' \cdot u' = w \wedge l \cdot t'' = x$
assume *l'*: $l' \cdot v' \cdot u' = w \wedge l' \cdot t'' = x$
have $(l \cdot v') \cdot u' = w \wedge (l \cdot v') \cdot t' = x \cdot v \wedge (l' \cdot v') \cdot u' = w \wedge (l' \cdot v') \cdot t' = x \cdot v$
using *assms*(*2*)
by (*metis commutative-squareE pushout-square-def l l' comp-assoc*)
thus $l = l'$
by (*metis* $*$ *2 assms*(*2*) *l l' pushout-square-def z*)
qed
ultimately show *?thesis* **by** *blast*
qed
qed

end

locale *elementary-category-with-bounded-pushouts* =
category C
for *C* :: *'a comp* (**infixr** $\cdot$ *55*)
and *inj0* :: *'a* ⇒ *'a* ⇒ *'a* (i_0[-, -])
and *inj1* :: *'a* ⇒ *'a* ⇒ *'a* (i_1[-, -]) +
assumes *inj0-ext*: ¬ *bounded-span h k* $\Longrightarrow \mathrm{i}_0[h, k] = null$
and *inj1-ext*: ¬ *bounded-span h k* $\Longrightarrow \mathrm{i}_1[h, k] = null$
and *pushout-commutes* [*intro*]:
bounded-span h k $\Longrightarrow$ *commutative-square* $\mathrm{i}_1[h, k]$ $\mathrm{i}_0[h, k]$ *h k*

and *pushout-universal*:
commutative-square f g h k $\Longrightarrow \exists!l.\ l \cdot \mathrm{i}_1[h,\ k] = f \wedge l \cdot \mathrm{i}_0[h,\ k] = g$
begin

lemma *dom-inj* [*simp*]:
assumes *bounded-span h k*
shows *dom* $\mathrm{i}_0[h,\ k]$ = *cod k* **and** *dom* $\mathrm{i}_1[h,\ k]$ = *cod h*
using *assms pushout-commutes* **by** *blast+*

lemma *cod-inj*:
assumes *bounded-span h k*
shows *cod* $\mathrm{i}_1[h,\ k]$ = *cod* $\mathrm{i}_0[h,\ k]$
using *assms pushout-commutes* **by** *auto*

lemma *has-bounded-pushouts*:
assumes *bounded-span h k*
shows *pushout-square* $\mathrm{i}_1[h,\ k]$ $\mathrm{i}_0[h,\ k]$ *h k*
using *assms pushout-square-def pushout-commutes pushout-universal* **by** *simp*

end

end

2.2 Categories of Transitions

theory *CategoryOfTransitions*
imports *Main Category3.EpiMonoIso CategoryWithBoundedPushouts*
ResiduatedTransitionSystem.ResiduatedTransitionSystem
begin

A category of transitions is a category with bounded pushouts in which every arrow is an epimorphism and the only isomorphisms are identities.

locale *category-of-transitions* =
elementary-category-with-bounded-pushouts +
assumes *arr-implies-epi*: *arr t* $\Longrightarrow$ *epi t*
and *iso-implies-ide*: *iso t* $\Longrightarrow$ *ide t*
begin

lemma *commutative-square-sym*:
shows *commutative-square f g h k* $\Longrightarrow$ *commutative-square g f k h*
by *auto*

lemma *inj-sym*:
shows $\mathrm{i}_0[k,\ h]$ = $\mathrm{i}_1[h,\ k]$
proof −
have ¬ *bounded-span h k* $\Longrightarrow$ *?thesis*
using *inj0-ext* [*of k h*] *inj1-ext* [*of h k*] *bounded-span-sym* **by** *metis*
moreover have *bounded-span h k* $\Longrightarrow$ *?thesis*

proof −
assume *1*: *bounded-span h k*
have $(\exists\, l.\ l \cdot \mathrm{i}_1[h,\ k] = \mathrm{i}_0[k,\ h]) \wedge (\exists\, l'.\ l' \cdot \mathrm{i}_0[k,\ h] = \mathrm{i}_1[h,\ k])$
by (*meson 1 bounded-span-sym commutative-square-sym pushout-commutes pushout-universal*)
moreover have *epi* $\mathrm{i}_0[k,\ h] \wedge$ *epi* $\mathrm{i}_1[h,\ k]$
by (*meson 1 arr-implies-epi bounded-span-sym commutative-squareE pushout-commutes*)
ultimately show *?thesis*
by (*metis arr-implies-epi iso-iff-section-and-epi comp-cod-arr comp-ide-arr epiE epi-implies-arr ide-cod iso-implies-ide comp-assoc sectionI seqE*)
qed
ultimately show *?thesis* **by** *auto*
qed

In this setting, pushouts are uniquely determined.

lemma *pushouts-unique*:
assumes *pushout-square f g h k*
shows $f = \mathrm{i}_1[h,\ k]$ **and** $g = \mathrm{i}_0[h,\ k]$
proof −
obtain *w* **where** *w*: $w \cdot \mathrm{i}_1[h,\ k] = f \wedge w \cdot \mathrm{i}_0[h,\ k] = g$
using *assms pushout-universal pushout-square-def* **by** *meson*
obtain w' **where** w': $w' \cdot f = \mathrm{i}_1[h,\ k] \wedge w' \cdot g = \mathrm{i}_0[h,\ k]$
using *assms pushout-universal pushout-square-def pushout-commutes bounded-spanI*
by *meson*
have *1*: *ide w*
proof −
have *commutative-square f g h k*
using *assms pushout-square-def* **by** *blast*
hence *ide* $(w \cdot w')$
using *assms w w' comp-cod-arr ide-char' comp-assoc*
apply (*elim commutative-squareE*)
by (*metis arr-implies-epi epiE seqE*)
thus *?thesis*
by (*metis ideD(1) ide-is-iso inv-comp-right(2) iso-iff-section-and-epi sectionI seqE arr-implies-epi iso-implies-ide*)
qed
show $f = \mathrm{i}_1[h,\ k]$ **and** $g = \mathrm{i}_0[h,\ k]$
using *1 w w'* **by** (*metis comp-ide-arr ext null-is-zero(2)*)+
qed

lemma *inj-arr-self*:
assumes *arr t*
shows $\mathrm{i}_0[t,\ t] =$ *cod t* **and** $\mathrm{i}_1[t,\ t] =$ *cod t*
proof −
have *1*: *pushout-square* (*cod t*) (*cod t*) *t t*
using *assms comp-arr-dom pushout-universal*

```
    apply (intro pushout-squareI)
      apply auto[4]
    apply auto
     apply (metis commutative-squareE inj-sym)
    by (metis ide-cod commutative-squareE comp-arr-ide)
  show i₀[t, t] = cod t and i₁[t, t] = cod t
    using 1 pushouts-unique by auto
qed

lemma inj-arr-dom:
assumes arr t
shows i₀[t, dom t] = t and i₁[t, dom t] = cod t
proof −
  have 1: pushout-square (cod t) t t (dom t)
    using assms comp-arr-dom comp-cod-arr pushout-universal
    apply (intro pushout-squareI)
      apply auto[4]
    apply auto
     apply (metis cod-dom commutative-squareE)
    by (metis arr-implies-epi commutative-squareE epiE)
  show i₀[t, dom t] = t and i₁[t, dom t] = cod t
    using 1 pushouts-unique by auto
qed

lemma eq-iff-ide-inj:
assumes span t u
shows t = u ⟷ ide i₀[t, u] ∧ ide i₀[u, t]
proof
  show t = u ⟹ ide i₀[t, u] ∧ ide i₀[u, t]
    using assms by (simp add: inj-arr-self(1))
  show ide i₀[t, u] ∧ ide i₀[u, t] ⟹ t = u
    by (metis (no-types, lifting) comp-cod-arr commutative-squareE
        ide-char ide-def inj0-ext inj-sym pushout-commutes)
qed

lemma inj-comp:
assumes bounded-span t (v · u)
shows i₀[t, v · u] = i₀[i₀[t, u], v]
and i₀[v · u, t] = i₀[v, i₀[t, u]] · i₀[u, t]
proof −
  obtain w x where wx: seq w t ∧ w · t = x · v · u
    using assms by blast
  have 1: seq w t ∧ w · t = (x · v) · u
    using wx comp-assoc by auto
  have 2: pushout-square i₀[u, t] i₀[t, u] t u
    by (metis (full-types) 1 bounded-span-def cod-comp
        commutative-squareI dom-comp has-bounded-pushouts inj-sym seqE)
  have 3: pushout-square i₀[v, i₀[t, u]] i₀[i₀[t, u], v] i₀[t, u] v
  proof −
```

```
    have 4: commutative-square w (x · v) t u
      using wx comp-assoc
      by (metis (mono-tags, lifting) cod-comp commutative-square-def
          dom-comp seqE)
    obtain l where l: l · i0[u, t] = w ∧ l · i0[t, u] = x · v
      using 2 4 pushout-square-def [of i0[u, t] i0[t, u] t u] by blast
    have bounded-span i0[t, u] v
      by (metis (full-types) 1 l bounded-spanI cod-comp
          commutative-squareI dom-comp seqE)
    thus ?thesis
      using assms has-bounded-pushouts inj-sym by auto
  qed
  show i0[t, v · u] = i0[i0[t, u], v]
    using assms 2 3 composition-of-pushouts pushouts-unique
    by (metis (full-types))
  show i0[v · u, t] = i0[v, i0[t, u]] · i0[u, t]
    using assms 2 3 composition-of-pushouts pushouts-unique inj-sym
    by metis
qed

lemma inj-prefix:
assumes arr (u · t)
shows i0[u · t, t] = u and i0[t, u · t] = cod u
proof −
  have 1: bounded-span (u · t) t
    by (metis assms ideD(1) ide-cod inj1-ext inj-arr-self(1)
        inj-comp(2) inj-sym not-arr-null seqE)
  show i0[u · t, t] = u
    using assms 1
    by (metis bounded-span-sym seqE comp-arr-dom inj-arr-dom(1)
        inj-arr-self(1) inj-comp(2))
  show i0[t, u · t] = cod u
    using 1
    by (metis assms bounded-span-sym seqE inj-arr-dom(2) inj-arr-self(1)
        inj-comp(1) inj-sym)
qed

end
```

2.3 Extensional RTS's with Composites as Categories

An extensional RTS with composites can be regarded as a category in an obvious way.

```
locale extensional-rts-with-composites-as-category =
  A: extensional-rts-with-composites
begin
```

Because we've defined RTS composition to take its arguments in diagram order, the ordering has to be reversed to match the way it is done for categories.

```
interpretation Category.partial-magma ‹λu t. t · u›
  using A.comp-null by unfold-locales metis
interpretation Category.partial-composition ‹λu t. t · u› ..

lemma null-char:
shows null = A.null
  by (metis A.comp-null(1) null-is-zero(1))

lemma ide-char:
shows ide a ⟷ A.ide a
proof
  assume a: A.ide a
  show ide a
  proof (unfold ide-def, intro conjI allI impI)
    show a · a ≠ null
      using a null-char by auto
    show ⋀t. t · a ≠ null ⟹ t · a = t
      using a
      by (metis A.comp-def A.comp-is-composite-of(2) A.composable-def
          A.composite-of-def A.cong-char null-char)
    show ⋀t. a · t ≠ null ⟹ a · t = t
      using a
      by (metis A.arr-comp A.ide-iff-trg-self A.comp-src-arr null-char
          A.comp-def A.arr-compE_EC)
  qed
  next
  assume a: ide a
  show A.ide a
    using a
    by (metis A.arr-comp A.comp-def A.comp-eqI A.cong-reflexive
        ide-def null-char)
qed

lemma src-in-domains:
assumes A.arr t
shows A.src t ∈ domains t
  unfolding domains-def
  using assms ide-char null-char by force

lemma trg-in-codomains:
assumes A.arr t
shows A.trg t ∈ codomains t
  unfolding codomains-def
  using assms ide-char null-char by force

lemma arr-char:
```

shows *arr* = *A.arr*
proof
  fix *t*
  show *arr t* ⟷ *A.arr t*
  proof
    show *A.arr t* ⟹ *arr t*
      using *src-in-domains trg-in-codomains arr-def* **by** *auto*
    assume *t*: *arr t*
    have *1*: *domains t* ≠ {} ∨ *codomains t* ≠ {}
      using *t arr-def* **by** *simp*
    show *A.arr t*
    proof (*cases domains t* ≠ {})
      case *True*
      obtain *a* **where** *a*: *a* ∈ *domains t*
        using *True* **by** *auto*
      show *A.arr t*
        using *a t A.composable-iff-seq* [*of a t*] *A.comp-def domains-def*
            null-char
        by *auto*
      next
      case *False*
      obtain *a* **where** *a*: *a* ∈ *codomains t*
        using *False 1* **by** *auto*
      show *A.arr t*
        using *a t A.composable-iff-seq* [*of t a*] *A.comp-def codomains-def*
            null-char
        by *auto*
    qed
  qed
qed

lemma *seq-char*:
shows *seq u t* ⟷ *A.seq t u*
  using *arr-char* **by** *auto*

sublocale *category* ‹λ*u t*. *t* · *u*›
proof
  show ⋀*g f*. *f* · *g* ≠ *null* ⟹ *seq g f*
    using *A.composable-iff-comp-not-null null-char seq-char* **by** *auto*
  show ⋀*f*. (*domains f* ≠ {}) = (*codomains f* ≠ {})
    using *arr-char arr-def src-in-domains trg-in-codomains empty-iff* **by** *metis*
  fix *f g h*
  show *1*: *f* · (*g* · *h*) = (*f* · *g*) · *h*
    using *A.comp-assoc*$_{EC}$ **by** *blast*
  show ⟦*seq h g*; *seq* (*g* · *h*) *f*⟧ ⟹ *seq g f*
    using *A.comp-assoc*$_{EC}$ *seq-char* **by** *auto*
  show ⟦*seq h* (*f* · *g*); *seq g f*⟧ ⟹ *seq h g*
    by (*metis 1 A.arr-compE*$_{EC}$ *arr-char*)
  show ⟦*seq g f*; *seq h g*⟧ ⟹ *seq* (*g* · *h*) *f*

```
    using seq-char by fastforce
  qed

  proposition is-category:
  shows category (λu t. t · u)
    ..

  lemma cod-char:
  shows cod = A.trg
    unfolding A.trg-def
    by (metis arr-char cod-def cod-eqI' codomains-char null-char
        A.con-implies-arr(2) trg-in-codomains A.conI A.trg-def)

  lemma dom-char:
  shows dom = A.src
    unfolding A.src-def
    by (metis A.src-def arr-char arr-def dom-eqI' dom-def null-char
        src-in-domains)

  lemma arr-implies-epi:
  assumes arr t
  shows epi t
    using assms
    by (metis arr-char epiI A.comp-cancel-left)

  lemma iso-implies-ide:
  assumes iso t
  shows ide t
    by (metis A.ide-backward-stable A.src-ide arr-char arr-inv assms
        comp-inv-arr' dom-char dom-inv ide-char' iso-is-arr A.ide-src
        A.prfx-comp seqI)

end
```

2.4 Characterization

The categories arising from extensional RTS's with composites are categories of transitions.

```
context extensional-rts-with-composites-as-category
begin

  sublocale category-of-transitions ‹λu t. t · u› ‹λh k. h \ k› ‹λh k. k \ h›
  proof
    show 1: ⋀h k. ¬ bounded-span h k ⟹ k \ h = null
    proof −
      fix h k
      have k \ h ≠ null ⟹ commutative-square (k \ h) (h \ k) h k
        by (metis (no-types, lifting) A.apex-sym A.conI A.con-imp-eq-src
```

A.diamond-commutes dom-char cod-char commutative-squareI
A.join-expansion(2) null-char seqE seq-char)
thus ¬ *bounded-span h k* ⟹ *k* \ *h* = *null*
by *auto*
qed
show ⋀*h k*. ¬ *bounded-span h k* ⟹ *h* \ *k* = *null*
using *1 bounded-span-sym* **by** *blast*
show ⋀*h k*. *bounded-span h k* ⟹ *commutative-square* (*k* \ *h*) (*h* \ *k*) *h k*
proof −
fix *h k*
assume *1*: *bounded-span h k*
hence *con*: *h* ⌢ *k*
by (*metis A.bounded-imp-con*$_E$ *A.cong-reflexive arr-char*
bounded-span-def commutative-squareE seqI)
show *commutative-square* (*k* \ *h*) (*h* \ *k*) *h k*
proof
show *cospan* (*k* \ *h*) (*h* \ *k*)
by (*simp add*: *con A.apex-sym A.con-sym arr-char cod-char*)
show *span h k*
using *1* **by** *blast*
show *dom* (*k* \ *h*) = *cod h*
using *A.join-expansion(2) con seq-char* **by** *blast*
show *h* · *k* \ *h* = *k* · *h* \ *k*
using *A.diamond-commutes* **by** *blast*
qed
qed
show ⋀*f g h k*. *commutative-square f g h k*
⟹ ∃!*l*. *k* \ *h* · *l* = *f* ∧ *h* \ *k* · *l* = *g*
proof −
fix *f g h k*
assume *1*: *commutative-square f g h k*
hence *2*: *h* · *f* = *k* · *g*
using *dom-char cod-char* **by** *auto*
hence *3*: *h* ⌢ *k*
by (*metis 1 A.bounded-imp-con*$_E$ *A.prfx-reflexive arr-char*
commutative-squareE seqI)
let *?l* = (*h* · *f*) \ (*h* ⊔ *k*)
have *4*: (*k* \ *h*) · *?l* = *f*
by (*metis 1 3 A.comp-assoc*$_{EC}$ *A.comp-resid-prfx A.induced-arrow(2)*
A.join-expansion(1) A.seq-implies-arr-comp commutative-squareE
seqI seq-char)
moreover have (*h* \ *k*) · *?l* = *g*
by (*metis 1 A.comp-resid-prfx A.induced-arrow(2)*
A.seq-implies-arr-comp calculation commutative-squareE
seqI seq-char)
ultimately have ∃ *l*. (*k* \ *h*) · *?l* = *f* ∧ (*h* \ *k*) · *?l* = *g*
by *simp*
moreover have ⋀*l'*. (*k* \ *h*) · *l'* = *f* ∧ (*h* \ *k*) · *l'* = *g* ⟹ *l'* = *?l*
by (*metis 1 4 A.comp-cancel-left arr-char commutative-square-def*)

```
    ultimately show ∃!l. (k \ h) · l = f ∧ (h \ k) · l = g by auto
  qed
  show ⋀t. arr t ⟹ epi t
    using arr-implies-epi by simp
  show ⋀t. iso t ⟹ ide t
    using iso-implies-ide by simp
qed

proposition is-category-of-transitions:
shows category-of-transitions (λu t. t · u) (λh k. h \ k) (λh k. k \ h)
  ..

end
```

Every category of transitions is derived from an underlying extensional RTS, obtained by using pushouts to define residuation.

```
locale underlying-rts =
  C: category-of-transitions
begin

  interpretation ResiduatedTransitionSystem.partial-magma ‹λh k. i₀[h, k]›
    by unfold-locales
       (metis C.inj1-ext C.inj-sym C.not-arr-null C.pushout-commutes
        C.commutative-squareE)

  lemma null-char:
  shows null = C.null
    using null-eqI C.inj0-ext C.not-arr-null C.pushout-commutes
    by (metis C.commutative-squareE)

  interpretation residuation ‹λh k. i₀[h, k]›
  proof
    show 0: ⋀t u. i₀[t, u] ≠ null ⟹ i₀[u, t] ≠ null
      by (metis C.inj0-ext C.inj-sym C.not-arr-null C.pushout-commutes
          C.commutative-squareE null-char)
    show ⋀t u. i₀[t, u] ≠ null ⟹ i₀[i₀[t, u], i₀[t, u]] ≠ null
      by (metis C.commutative-squareE C.eq-iff-ide-inj C.ideD(1) C.inj0-ext
          C.not-arr-null C.pushout-commutes null-char)
    have ∗: ⋀t u v. ⟦i₀[t, u] ≠ null; i₀[t, v] ≠ null; i₀[i₀[v, t], i₀[u, t]] ≠ null⟧
                    ⟹ ∃w. i₀[i₀[v, t], i₀[u, t]] = w · i₀[i₀[v, u], i₀[t, u]]
    proof −
      fix t u v
      assume tu: i₀[t, u] ≠ null and tv: i₀[t, v] ≠ null
      and vt-ut: i₀[i₀[v, t], i₀[u, t]] ≠ null
      have 1: (i₀[i₀[v, t], i₀[u, t]] · i₀[t, u]) · u = (i₀[i₀[u, t], i₀[v, t]] · i₀[t, v]) · v
        using tu tv vt-ut C.commutative-squareE C.comp-assoc C.inj0-ext
              C.inj-sym C.pushout-commutes null-char
        by metis
      have 2: C.commutative-square (i₀[i₀[v, t], i₀[u, t]] · i₀[t, u])
```

```
        (i₀[i₀[u, t], i₀[v, t]] · i₀[t, v]) u v
  proof −
    have C.span u v
      by (metis (mono-tags, opaque-lifting) C.commutative-square-def
          C.inj0-ext C.pushout-commutes null-char tu tv)
    moreover have C.cospan (i₀[i₀[v, t], i₀[u, t]] · i₀[t, u])
                           (i₀[i₀[u, t], i₀[v, t]] · i₀[t, v])
    proof (intro conjI)
      show 3: C.seq i₀[i₀[v, t], i₀[u, t]] i₀[t, u]
        by (metis (mono-tags, opaque-lifting) C.commutative-square-def
            C.ext C.inj0-ext C.inj-sym C.pushout-commutes C.seqI
            null-char vt-ut)
      show 4: C.seq i₀[i₀[u, t], i₀[v, t]] i₀[t, v]
        by (metis 1 3 C.dom-inj(1) C.inj0-ext C.seqE C.seqI calculation
            C.dom-comp null-char tu)
      show C.cod (i₀[i₀[v, t], i₀[u, t]] · i₀[t, u]) =
           C.cod (i₀[i₀[u, t], i₀[v, t]] · i₀[t, v])
        by (metis 3 4 C.cod-inj C.inj0-ext C.inj-sym C.cod-comp)
    qed
    moreover have C.dom (i₀[i₀[v, t], i₀[u, t]] · i₀[t, u]) = C.cod u
      by (metis C.dom-comp C.dom-inj(1) C.inj0-ext calculation(2)
          null-char tu)
    ultimately show ?thesis
      using 1 by blast
  qed
  hence C.bounded-span u v
    by blast
  obtain l where l: l · i₀[v, u] = i₀[i₀[v, t], i₀[u, t]] · i₀[t, u] ∧
                    l · i₀[u, v] = i₀[i₀[u, t], i₀[v, t]] · i₀[t, v]
    using 2 C.pushout-universal by (metis C.inj-sym)
  have 3: C.commutative-square i₀[i₀[v, t], i₀[u, t]] l i₀[t, u] i₀[v, u]
    using tu tv vt-ut l 2
    by (metis (mono-tags, lifting) C.cod-comp C.commutative-square-def
        C.dom-comp C.seqE)
  obtain w where i₀[i₀[v, t], i₀[u, t]] = w · i₀[i₀[v, u], i₀[t, u]] ∧
                 l = w · i₀[i₀[t, u], i₀[v, u]]
    using 3 C.pushout-universal by (metis C.inj-sym)
  thus ∃ w. i₀[i₀[v, t], i₀[u, t]] = w · i₀[i₀[v, u], i₀[t, u]] by auto
qed
show ⋀t u v. i₀[i₀[v, t], i₀[u, t]] ≠ null
              ⟹ i₀[i₀[v, t], i₀[u, t]] = i₀[i₀[v, u], i₀[t, u]]
proof −
  fix t u v
  assume vt-ut: i₀[i₀[v, t], i₀[u, t]] ≠ null
  have tu: i₀[t, u] ≠ null
    using vt-ut
    by (metis ‹⋀u t. i₀[t, u] ≠ null ⟹ i₀[u, t] ≠ null› null-is-zero(2))
  have tv: i₀[t, v] ≠ null
    using vt-ut
```

```
    by (metis ‹⋀u t. i₀[t, u] ≠ null ⟹ i₀[u, t] ≠ null› null-is-zero(1))
  obtain w where w: i₀[i₀[v, t], i₀[u, t]] = w · i₀[i₀[v, u], i₀[t, u]]
    using tu tv vt-ut ∗ by blast
  have vu-tu: i₀[i₀[v, u], i₀[t, u]] ≠ null
    using w null-char vt-ut by fastforce
  obtain w′ where w′: i₀[i₀[v, u], i₀[t, u]] = w′ · i₀[i₀[v, t], i₀[u, t]]
    using 0 vu-tu ∗ null-is-zero(2) by metis
  have C.ide (w · w′)
    using w w′ vt-ut
    by (metis C.comp-cod-arr C.ext C.ide-cod C.inj-prefix(1−2) C.seqE
        null-char)
  hence C.ide w
    using w w′
    by (metis C.comp-arr-ide C.ide-compE C.inj-arr-dom(1) C.inj-prefix(2)
        C.seqE)
  thus i₀[i₀[v, t], i₀[u, t]] = i₀[i₀[v, u], i₀[t, u]]
    using w C.comp-ide-arr
    by (metis C.ext null-char vt-ut)
 qed
qed

lemma con-char:
shows con t u ⟷ C.bounded-span t u
 by (metis C.commutative-squareE C.inj0-ext C.not-arr-null
     C.pushout-commutes con-def null-char)

lemma arr-char:
shows arr t ⟷ C.arr t
 by (metis C.arr-cod C.inj-arr-self(1) C.not-arr-null C.pushout-commutes
     arr-def C.commutative-squareE con-char con-def null-char)

lemma ide-char:
shows ide a ⟷ C.ide a
 by (metis C.ide-char C.ide-char′ C.ide-def C.inj-arr-self(1) arr-char ide-def
     null-char con-def ide-implies-arr)

interpretation rts ‹λh k. i₀[h, k]›
proof
 show ⋀a t. ⟦ide a; con t a⟧ ⟹ i₀[t, a] = t
   using C.inj-arr-dom con-char ide-char by fastforce
 thus ⋀a t. ⟦ide a; con a t⟧ ⟹ ide i₀[a, t]
   by (metis con-sym cube ide-def con-def)
 show ⋀t. arr t ⟹ ide (trg t)
   by (metis arr-char C.eq-iff-ide-inj ide-char resid-arr-self)
 thus ⋀t u. con t u ⟹ ∃a. ide a ∧ con a t ∧ con a u
   using con-char ide-char
   by (metis C.ide-dom C.inj-arr-dom(1) C.pushout-commutes
       C.commutative-squareE con-implies-arr(1) con-sym arr-resid-iff-con)
 show ⋀t u v. ⟦ide i₀[t, u]; con u v⟧ ⟹ con i₀[t, u] i₀[v, u]
```

by (*metis* (*no-types*, *lifting*) *C.dom-inj*(*1*) *C.ideD*(*2*) *C.inj-arr-dom*(*1*)
arr-char con-char ide-char arr-resid-iff-con con-sym ide-implies-arr)
qed

interpretation *extensional-rts* ‹$\lambda h\ k.\ \mathrm{i}_0[h,\ k]$›
by *unfold-locales*
(*metis C.ideD*(*2*) *C.inj-sym C.pushout-commutes prfx-implies-con*
C.commutative-squareE C.comp-cod-arr con-char ide-char)

lemma *src-char*:
assumes *arr t*
shows *src t* = *C.dom t*
by (*metis C.ide-dom C.inj-arr-dom*(*1*) *arr-char arr-resid-iff-con assms*
con-imp-eq-src ide-char src-ide)

lemma *trg-char*:
assumes *arr t*
shows *trg t* = *C.cod t*
by (*metis assms C.inj-arr-self*(*1*) *resid-arr-self arr-char*)

lemma *seq-char*:
shows *seq t u* ⟷ *C.seq u t*
using *seq-def sources-char targets-char*$_{W\,E}$ *arr-char src-char trg-char*
by *auto*

lemma *comp-char*:
shows *comp t u* = *u* · *t*
by (*metis C.cod-comp C.ext C.inj-arr-self*(*1*) *C.inj-prefix*(*1*−*2*)
arr-char composable-imp-seq cong-reflexive comp-def null-char
prfx-decomp seq-char)

sublocale *extensional-rts-with-composites* ‹$\lambda h\ k.\ \mathrm{i}_0[h,\ k]$›
by *unfold-locales*
(*metis C.not-arr-null composable-iff-comp-not-null comp-char null-char*
seq-char)

proposition *is-extensional-rts-with-composites*:
shows *extensional-rts-with-composites* (λ*h k.* $\mathrm{i}_0[h,\ k]$)
..

end

end

Chapter 3

RTS Constructions

This section develops several constructions on residuated transition systems, including the construction of: an RTS with no transitions (at an arbitrary type), an RTS with exactly one transition (at any type having at least two elements), free and fibered (binary) products of RTS's, and an exponential RTS. These constructions will be used in a subsequent section to construct a cartesian closed category having residuated transition systems as objects and simulations as arrows. The natural definitions of the product and exponential constructions on RTS's yield results at higher types than those of their arguments, but for a cartesian closed category we need versions of these constructions that produce results at the same type as their arguments. Since it is not possible in the case of the exponential to carry out such a construction within HOL (except for finite types), we make use of the "ZFC in HOL" axiomatic extension to HOL to obtain a type having suitable closure properties. The ZFC in HOL extension includes definitions of "smallness" for sets and types, and we show that each of the RTS constructions preserves smallness in a suitable sense. We then show that the small results (at higher type) of applying the constructions to small arguments can be mapped back, via functions injective on arrows, to isomorphic copies that "live" at the original argument type.

theory *RTSConstructions*
imports *Main Preliminaries ZFC-in-HOL.ZFC-Cardinals*
begin

3.1 Notation

Some of the theories in the HOL library that we depend on define global notation involving generic symbols that we would like to use here. It would be best if there were some way to import these theories without also having to import this notation, but for now the best we can do is to uninstall the

notation involving the symbols at issue.

no-notation *Equipollence.eqpoll* (**infixl** $\approx$ *50*)
no-notation *Equipollence.lepoll* (**infixl** $\lesssim$ *50*)
no-notation *Lattices.sup-class.sup* (**infixl** $\sqcup$ *65*)
no-notation *ZFC-Cardinals.cmult* (**infixl** $\langle\otimes\rangle$ *70*)

no-syntax *-Tuple* :: [*V*, *Vs*] $\Rightarrow$ *V* ($\langle\langle$-,/ -$\rangle\rangle$)
no-syntax *-hpattern* :: [*pttrn*, *patterns*] $\Rightarrow$ *pttrn* ($\langle\langle$-,/ -$\rangle\rangle$)

3.2 Some Constraints on a Type

3.2.1 Nondegenerate

We will call a type "nondegenerate" if it has at least two elements. This means that the type admits RTS's with a non-empty set of arrows (after using one of the elements for the required null value).

locale *nondegenerate* =
fixes *type* :: *'a itself*
assumes *is-nondegenerate*: $\exists\, x\ y ::$ *'a*. $x \neq y$

3.2.2 Lifting

A type *'a* "admits lifting" if there is an injection from the type *'a option* to *'a*.

locale *lifting* =
fixes *type* :: *'a itself*
assumes *admits-lifting*: $\exists\, l ::$ *'a option* $\Rightarrow$ *'a*. *inj l*
begin

definition *some-lift* :: *'a option* $\Rightarrow$ *'a*
where *some-lift* $\equiv$ *SOME l* :: *'a option* $\Rightarrow$ *'a*. *inj l*

lemma *inj-some-lift*:
shows *inj some-lift*
using *admits-lifting someI-ex* [*of* λ*l*. *inj l*] *some-lift-def* **by** *fastforce*

A type that admits lifting is obviously nondegenerate.

sublocale *nondegenerate*
proof (*unfold-locales*, *intro exI*)
show *some-lift None* $\neq$ *some-lift* (*Some* (*some-lift None*))
using *injD inj-some-lift* **by** *fastforce*
qed

end

3.2.3 Pairing

A type $'a$ "admits pairing" if there exists an injective "pairing function" from $'a * 'a$ to $'a$. This allows us to encode pairs of elements of $'a$ without having to pass to a higher type.

locale *pairing* =
fixes *type* :: $'a$ *itself*
assumes *admits-pairing*: $\exists\, p :: 'a * 'a \Rightarrow 'a.$ *inj p*
begin

definition *some-pair* :: $'a * 'a \Rightarrow 'a$
where *some-pair* $\equiv$ *SOME p* :: $'a * 'a \Rightarrow 'a.$ *inj p*

abbreviation *is-pair*
where *is-pair x* $\equiv$ *x* $\in$ *range some-pair*

definition *first* :: $'a \Rightarrow 'a$
where *first x* $\equiv$ *fst* (*inv some-pair x*)

definition *second* :: $'a \Rightarrow 'a$
where *second x* = *snd* (*inv some-pair x*)

lemma *inj-some-pair*:
shows *inj some-pair*
using *admits-pairing someI-ex* [*of* $\lambda p.$ *inj p*] *some-pair-def* **by** *fastforce*

lemma *first-conv*:
shows *first* (*some-pair* (*x*, *y*)) = *x*
using *first-def inj-some-pair* **by** *auto*

lemma *second-conv*:
shows *second* (*some-pair* (*x*, *y*)) = *y*
using *second-def inj-some-pair* **by** *auto*

lemma *pair-conv*:
assumes *is-pair x*
shows *some-pair* (*first x*, *second x*) = *x*
using *assms first-def second-def inj-some-pair* **by** *force*

end

A type that is nondegenerate and admits pairing also admits lifting.

locale *nondegenerate-and-pairing* =
nondegenerate + *pairing*
begin

sublocale *lifting type*
proof
obtain *c* :: $'a$ **where** *c*: $\forall\, x.$ *c* $\neq$ *some-pair* (*c*, *x*)

```
    using is-nondegenerate inj-some-pair
    by (metis (full-types) first-conv second-conv)
  let ?f = λNone ⇒ c | Some x ⇒ some-pair (c, x)
  have inj ?f
    unfolding inj-def
    by (metis (no-types, lifting) c option.case-eq-if option.collapse
        second-conv)
  thus ∃ l :: 'a option ⇒ 'a. inj l
    by blast
qed

end
```

3.2.4 Exponentiation

In order to define the exponential $[A, B]$ of an RTS A and an RTS B at a type $'a$ without having to pass to a higher type, we need the type $'a$ to be large enough to embed the set of all extensional functions that have "small" sets as their domains. Here we are using the notion of "small" provided by the `ZFC_in_HOL` extension to HOL. Now, the standard Isabelle/HOL definition of "extensional" uses the specific chosen value *undefined* as the default value for an extensional function outside of its domain, but here we need to apply this concept in cases where the value could be something else (the null value for an RTS, in particular). So, we define a notion of a function that has at most one "popular value" in its range, where a popular value is one with a "large" preimage. If such a function in addition has a small range, then it in some sense has a small encoding, which consists of its graph restricted to its domain (which must then necessarily be small), paired with the single default value that it takes outside its domain.

```
abbreviation popular-value :: ('a ⇒ 'b) ⇒ 'b ⇒ bool
where popular-value F y ≡ ¬ small {x. F x = y}

definition some-popular-value :: ('a ⇒ 'b) ⇒ 'b
where some-popular-value F ≡ SOME y. popular-value F y

abbreviation at-most-one-popular-value
where at-most-one-popular-value F ≡ ∃≤1 y. popular-value F y

definition small-function
where small-function F ≡ small (range F) ∧ at-most-one-popular-value F

lemma small-preimage-unpopular:
fixes F :: 'a ⇒ 'b
assumes small-function F
shows small {x. F x ≠ some-popular-value F}
proof (cases ∃ y. popular-value F y)
  assume 1: ¬ (∃ y. popular-value F y)
```

```
  have ⋀y. small {x. F x = y}
    using 1 by blast
  moreover have UNIV = (⋃ y∈range F. {x. F x = y})
    by auto
  ultimately have small (UNIV :: 'a set)
    using assms(1) small-function-def by (metis small-UN)
  thus ?thesis
    using smaller-than-small by blast
  next
  assume 1: ∃ y. popular-value F y
  have popular-value F (some-popular-value F)
   using 1 someI-ex [of λy. popular-value F y] some-popular-value-def by metis
  hence 2: ⋀y. y ≠ some-popular-value F ⟹ small {x. F x = y}
    using assms
    unfolding small-function-def
    by (meson Uniq-D)
  moreover have {x. F x ≠ some-popular-value F} =
               (⋃ y∈{y. y ∈ range F ∧ y ≠ some-popular-value F}. {x. F x = y})
    by auto
  ultimately show ?thesis
    using assms
    unfolding small-function-def
    by auto
qed
```

A type $'a$ "admits exponentiation" if there is an injective function that maps each small function from $'a$ to $'a$ back into $'a$.

```
locale exponentiation =
fixes type :: 'a itself
assumes admits-exponentiation:
        ∃ e :: ('a ⇒ 'a) ⇒ 'a. inj-on e (Collect small-function)
begin

  definition some-inj :: ('a ⇒ 'a) ⇒ 'a
  where some-inj ≡ SOME e :: ('a ⇒ 'a) ⇒ 'a. inj-on e (Collect small-function)

  lemma inj-some-inj:
  shows inj-on some-inj (Collect small-function)
    using some-inj-def admits-exponentiation
          someI-ex [of λe :: ('a ⇒ 'a) ⇒ 'a. inj-on e (Collect small-function)]
    unfolding small-function-def
    by presburger

  definition app :: 'a ⇒ 'a ⇒ 'a
  where app f ≡ inv-into
                 {F. small (range F) ∧
                     at-most-one-popular-value F} some-inj f

  lemma app-some-inj:
```

assumes *small-function F*
shows *app* (*some-inj F*) = *F*
by (*metis* (*mono-tags*, *lifting*) *Collect-cong assms inv-into-f-f app-def inj-some-inj mem-Collect-eq small-function-def*)

lemma *some-inj-lam-app*:
assumes *f* ∈ *some-inj* ‘ *Collect small-function*
shows *some-inj* (λ*x*. *app f x*) = *f*
using *assms f-inv-into-f*
unfolding *small-function-def*
by (*metis* (*no-types*, *lifting*) *app-def*)

end

context
begin

The type *V* (axiomatized in *ZFC-in-HOL.ZFC-in-HOL*) admits exponentiation. We show this by exhibiting a “small encoding” for small functions. We provide this fact as evidence of the nontriviality of the subsequent development, in the sense that if the existence of the type *V* is consistent with HOL, then the existence of infinite types satisfying the locale assumptions for *exponentiation* is also consistent with HOL.

interpretation *exponentiation* ‹*TYPE*(*V*)›
proof
show ∃ *e* :: (*V* ⇒ *V*) ⇒ *V*. *inj-on e* (*Collect small-function*)
proof
let *?e* = λ*F*. *vpair* (*some-popular-value F*)
(*set* ((λ*a*. *vpair a* (*F a*)) ‘ {*x*. *F x* ≠ *some-popular-value F*}))
show *inj-on ?e* (*Collect small-function*)
proof (*intro inj-onI*)
fix *F F′* :: *V* ⇒ *V*
assume *F*: *F* ∈ *Collect small-function*
assume *F′*: *F′* ∈ *Collect small-function*
assume *eq*:
vpair (*some-popular-value F*)
(*set* ((λ*a*. *vpair a* (*F a*)) ‘ {*x*. *F x* ≠ *some-popular-value F*})) =
vpair (*some-popular-value F′*)
(*set* ((λ*a*. *vpair a* (*F′ a*)) ‘ {*x*. *F′ x* ≠ *some-popular-value F′*}))
have *1*: *some-popular-value F* = *some-popular-value F′* ∧
set ((λ*a*. *vpair a* (*F a*)) ‘ {*x*. *F x* ≠ *some-popular-value F*}) =
set ((λ*a*. *vpair a* (*F′ a*)) ‘ {*x*. *F′ x* ≠ *some-popular-value F′*})
using *eq* **by** *blast*
have *2*: (λ*a*. *vpair a* (*F a*)) ‘ {*x*. *F x* ≠ *some-popular-value F*} =
(λ*a*. *vpair a* (*F′ a*)) ‘ {*x*. *F′ x* ≠ *some-popular-value F′*}
proof −
have *small* {*x*. *F x* ≠ *some-popular-value F*}
using *F small-preimage-unpopular* **by** *blast*
hence *small* ((λ*a*. *vpair a* (*F a*)) ‘ {*x*. *F x* ≠ *some-popular-value F*})

by *blast*
thus *?thesis*
by (*metis* (*full-types*) *1 F′ mem-Collect-eq replacement set-injective small-preimage-unpopular*)
qed
show $F = F'$
proof
fix x
show $F\ x = F'\ x$
using *1 2*
by (*cases* $F\ x$ = *some-popular-value* F) *force+*
qed
qed
qed
qed

lemma *V-admits-exponentiation*:
shows *exponentiation* $TYPE(V)$
..

end

3.2.5 Universe

locale *universe* = *nondegenerate-and-pairing* + *exponentiation*

The type V axiomatized in *ZFC-in-HOL.ZFC-in-HOL* is a universe.

context
begin

interpretation *nondegenerate* ‹$TYPE(V)$›
proof
obtain $f :: bool \Rightarrow V$ **where** *f*: *inj f*
using *inj-compose inj-ord-of-nat* **by** *blast*
show $\exists x\ y :: V.\ x \neq y$
by (*metis Inl-Inr-iff*)
qed

lemma *V-is-nondegenerate*:
shows *nondegenerate* $TYPE(V)$
..

interpretation *pairing* ‹$TYPE(V)$›
apply *unfold-locales*
using *inj-on-vpair* **by** *blast*

lemma *V-admits-pairing*:
shows *pairing* $TYPE(V)$
..

interpretation *exponentiation* ‹*TYPE*(*V*)›
using *V-admits-exponentiation* **by** *blast*

interpretation *universe* ‹*TYPE*(*V*)›
..

lemma *V-is-universe*:
shows *universe TYPE*(*V*)
..

end

3.3 Small RTS's

We will call an RTS "small" if its set of arrows is a small set.

locale *small-rts* =
rts +
assumes *small*: *small* (*Collect arr*)

lemma *isomorphic-to-small-rts-is-small-rts*:
assumes *small-rts A* **and** *isomorphic-rts A B*
shows *small-rts B*
proof −
interpret *A*: *small-rts A*
using *assms* **by** *blast*
interpret *B*: *rts B*
using *assms isomorphic-rts-def inverse-simulations-def* **by** *blast*
obtain *F G* **where** *FG*: *inverse-simulations A B F G*
using *assms isomorphic-rts-def* **by** *blast*
interpret *FG*: *inverse-simulations A B F G*
using *FG* **by** *blast*
show *small-rts B*
using *A.small FG.G.is-bijection-betw-arr-sets*
apply *unfold-locales*
by (*metis bij-betw-imp-surj-on replacement*)
qed

lemma *small-function-transformation*:
assumes *small-rts A* **and** *small-rts B* **and** *transformation A B F G T*
shows *small-function T*
proof −
interpret *A*: *small-rts A*
using *assms*(*1*) **by** *blast*
interpret *B*: *small-rts B*
using *assms*(*2*) **by** *blast*
interpret *T*: *transformation A B F G T*
using *assms*(*3*) **by** *blast*
have *1*: *range T* ⊆ *Collect B.arr* ∪ {*B.null*}

```
    using T.extensional T.preserves-arr by blast
  show ?thesis
  proof (unfold small-function-def, intro conjI)
    show small (range T)
      using assms(2) 1 B.small smaller-than-small by blast
    show at-most-one-popular-value T
    proof −
      have ⋀v. popular-value T v ⟹ v = B.null
      proof −
        fix v
        assume v: popular-value T v
        have v ≠ B.null ⟹ v ∈ range T
          using v
          by (metis (mono-tags, lifting) empty-Collect-eq rangeI small-empty)
        thus v = B.null
          by (metis (mono-tags, lifting) A.small Collect-mono T.extensional
              smaller-than-small v)
      qed
      thus ?thesis
        using Uniq-def by blast
    qed
  qed
qed
```

We can't simply use the previous fact to prove the following, because our definition of transformation includes extensionality conditions that are not part of the definition of simulation. So, we have to repeat the proof.

```
lemma small-function-simulation:
assumes small-rts A and small-rts B and simulation A B F
shows small-function F
proof −
  interpret A: small-rts A
    using assms(1) by blast
  interpret B: small-rts B
    using assms(2) by blast
  interpret F: simulation A B F
    using assms(3) by blast
  have 1: range F ⊆ Collect B.arr ∪ {B.null}
    using F.extensional F.preserves-reflects-arr by blast
  show ?thesis
  proof (unfold small-function-def, intro conjI)
    show small (range F)
      using assms(2) 1 B.small smaller-than-small by blast
    show at-most-one-popular-value F
    proof −
      have ⋀v. popular-value F v ⟹ v = B.null
      proof −
        fix v
        assume v: popular-value F v
```

```
      have v ≠ B.null ⟹ v ∈ range F
        using v
        by (metis (mono-tags, lifting) empty-Collect-eq rangeI small-empty)
      thus v = B.null
        by (metis (mono-tags, lifting) A.small Collect-mono F.extensional
            smaller-than-small v)
    qed
    thus ?thesis
      using Uniq-def by blast
  qed
 qed
qed

lemma small-function-resid:
fixes A :: 'a resid
assumes small-rts A
shows small-function A
and ⋀t. small-function (A t)
proof −
 interpret A: small-rts A
   using assms by blast
 show 1: small-function A
 proof (unfold small-function-def, intro conjI)
   show small (range A)
   proof −
     have range A ⊆ A ‘ Collect A.arr ∪ A ‘ {x. ¬ A.arr x}
       by blast
     moreover have small (A ‘ Collect A.arr)
       using A.small by blast
     moreover have small (A ‘ {x. ¬ A.arr x})
     proof −
       have ⋀x. ¬ A.arr x ⟹ A x = (λx. A.null)
         using A.con-implies-arr(1) by blast
       hence A ‘ {x. ¬ A.arr x} ⊆ {λx. A.null}
         by blast
       thus ?thesis
         by (meson small-empty small-insert smaller-than-small)
     qed
     ultimately show ?thesis
       by (meson small-Un smaller-than-small)
   qed
   show at-most-one-popular-value A
   proof −
     have ⋀v. popular-value A v ⟹ v ∈ A ‘ {x. ¬ A.arr x}
     proof −
       fix v
       assume v: popular-value A v
       have ¬ small {x. ¬ A.arr x ∧ A x = v}
       proof −
```

```
      have ¬ small ({x. A x = v} - {x. A.arr x ∧ A x = v})
        by (metis (mono-tags, lifting) A.small Collect-mono
            Un-Diff-cancel small-Un smaller-than-small sup-ge2 v)
      moreover have {x. A x = v} - {x. A.arr x ∧ A x = v} =
                    {x. ¬ A.arr x ∧ A x = v}
        by blast
      ultimately show ?thesis by metis
    qed
    hence v ∈ A ‘ {x. ¬ A.arr x ∧ A x = v}
      by (metis (mono-tags, lifting) empty-Collect-eq image-eqI
          mem-Collect-eq small-empty)
    thus v ∈ A ‘ {x. ¬ A.arr x} by blast
  qed
  moreover have A ‘ {x. ¬ A.arr x} ⊆ {λx. A.null}
  proof -
    have ⋀x. ¬ A.arr x ⟹ A x = (λx. A.null)
      using A.con-implies-arr(1) by blast
    thus ?thesis by blast
  qed
  ultimately show ?thesis
    by (metis (no-types, lifting) Uniq-def empty-iff singletonD
        subset-singleton-iff)
 qed
qed
show 2: ⋀t. small-function (A t)
proof -
 fix t
 show small-function (A t)
 proof (unfold small-function-def, intro conjI)
   show small (range (A t))
   proof -
     have range (A t) ⊆ Collect A.arr ∪ {A.null}
       using A.arr-resid by blast
     moreover have small (Collect A.arr ∪ {A.null})
       using A.small by simp
     ultimately show ?thesis
       using smaller-than-small by blast
   qed
   show at-most-one-popular-value (A t)
   proof -
     have ⋀v. popular-value (A t) v ⟹ v = A.null
     proof -
       fix v
       assume v: popular-value (A t) v
       have ¬ small {u. A t u = v}
         using v by blast
       hence ¬ ({u. A t u = v} ⊆ Collect A.arr)
         using A.small smaller-than-small by blast
       hence ∃u. A t u = v ∧ ¬ A.arr u
```

```
          by blast
        thus v = A.null
          using A.con-implies-arr(2) by blast
      qed
      thus ?thesis
        using Uniq-def by blast
    qed
  qed
qed

context exponentiation
begin

  lemma small-function-some-inj-resid:
  fixes A :: 'a resid
  assumes small-rts A
  shows small-function (λt. some-inj (A t))
  proof −
    interpret A: small-rts A
      using assms by blast
    show small-function (λt. some-inj (A t))
    proof (unfold small-function-def, intro conjI)
      show small (range (λt. some-inj (A t)))
      proof −
        have range (λt. some-inj (A t)) = some-inj ‘ range (λt. A t)
          by auto
        moreover have small ...
          using assms small-function-resid(1)
          by (metis replacement small-function-def)
        ultimately show ?thesis by auto
      qed
      show at-most-one-popular-value (λt. some-inj (A t))
      proof −
        have 3: ⋀t v. popular-value (λt. some-inj (A t)) v
                      ⟹ v ∈ some-inj ‘ Collect (popular-value A)
        proof −
          fix t v
          assume v: popular-value (λt. some-inj (A t)) v
          have ¬ small {t. A t = inv-into (Collect small-function) some-inj v}

          proof −
            have 1: ⋀t. A.arr t ⟷ some-inj (A t) ≠ v
            proof
              have 2: ⋀t. A.arr t ⟷ A t ≠ (λu. A.null)
                using A.con-implies-arr(1) by fastforce
              have 3: v = some-inj (λu. A.null)
                using v 2
                by (metis (mono-tags, lifting) A.small Collect-mono
```

```
            smaller-than-small)
        show ⋀t. some-inj (A t) ≠ v ⟹ A.arr t
          using 2 3 by force
        show ⋀t. A.arr t ⟹ some-inj (A t) ≠ v
          using assms 2 3 inj-some-inj app-some-inj small-function-resid(2)
          by (metis A.not-arr-null)
      qed
      have {t. A t = inv-into (Collect small-function) some-inj v} =
           {t. ¬ A.arr t}
        using 1
        by (metis (no-types, lifting) A.not-arr-null CollectD CollectI
            app-some-inj assms f-inv-into-f image-eqI inv-into-into
            small-function-resid(2))
      thus ?thesis
        using v 1 by auto
    qed
    hence inv-into (Collect small-function) some-inj v
            ∈ Collect (popular-value A)
      by auto
    moreover have some-inj
                    (inv-into (Collect small-function) some-inj v) = v
      using assms v inj-some-inj
            f-inv-into-f [of v some-inj Collect small-function]
      by (metis (mono-tags) small-function-resid(2) empty-Collect-eq
          inv-into-f-f mem-Collect-eq small-empty)
    ultimately show v ∈ some-inj ‘ Collect (popular-value A)
      by force
  qed
  show ?thesis
  proof
    fix u v
    assume u: popular-value (λx. some-inj (A x)) u
    assume v: popular-value (λx. some-inj (A x)) v
    obtain f where f: popular-value A f ∧ some-inj f = u
      using u 3 by blast
    obtain g where g: popular-value A g ∧ some-inj g = v
      using v 3 by blast
    have f = g
      using assms f g small-function-resid(1) Uniq-D
      unfolding small-function-def
      by auto fastforce
    thus u = v
      using f g by blast
  qed
 qed
qed

fun some-inj-resid :: 'a resid ⇒ 'a
```

```
  where some-inj-resid A = (some-inj (λt. some-inj (A t)))

  lemma inj-on-some-inj-resid:
  shows inj-on some-inj-resid {A :: 'a resid. small-rts A}
  proof
    fix A B :: 'a resid
    assume A: A ∈ {A. small-rts A} and B: B ∈ {B. small-rts B}
    assume eq: some-inj-resid A = some-inj-resid B
    interpret A: small-rts A
      using A by blast
    interpret B: small-rts B
      using B by blast
    show A = B
    proof −
      have some-inj (λt. some-inj (A t)) = some-inj (λt. some-inj (B t))
        using A B eq by simp
      moreover have small-function (λt. some-inj (A t))
        using A small-function-some-inj-resid by auto
      moreover have small-function (λt. some-inj (B t))
        using B small-function-some-inj-resid by auto
      ultimately have (λt. some-inj (A t)) = (λt. some-inj (B t))
        using A B inj-some-inj
        by (simp add: inj-onD)
      hence ⋀t. A t = B t
        using A B inj-some-inj small-function-resid(2)
        by (metis app-some-inj mem-Collect-eq)
      thus A = B by blast
    qed
  qed

end
```

3.4 Injective Images of RTS's

Here we show that the image of an RTS A at type $'a$, under a function from $'a$ to $'b$ that is injective on the set of arrows, is an RTS at type $'b$ that is isomorphic to A. We will use this, together with the universe assumptions, to obtain isomorphic images, of constructions such as product and exponential RTS, that yield results that "live" at the same type as their arguments.

```
  locale inj-image-rts =
    A: rts A
    for map :: 'a ⇒ 'b
    and A :: 'a resid  (infix \<^sub>A 70) +
    assumes inj-map: inj-on map (Collect A.arr ∪ {A.null})
  begin

    notation A.con    (infix ⌢A 50)
    notation A.prfx   (infix ≲A 50)
```

notation *A.cong* (**infix** $\sim_A$ *50*)

abbreviation *Null*
where *Null* $\equiv$ *map A.null*

abbreviation *Arr*
where *Arr t* $\equiv$ *t* $\in$ *map* ‘ *Collect A.arr*

abbreviation *map′*
where *map′ t* $\equiv$ *inv-into* (*Collect A.arr*) *map t*

definition *resid* :: *'b resid* (**infix** $\setminus$ *70*)
where *t* $\setminus$ *u* = (*if Arr t* $\wedge$ *Arr u then map* (*map′ t* $\setminus_A$ *map′ u*) *else Null*)

lemma *inj-map′*:
shows *inj-on map* (*Collect A.arr*)
  using *inj-map* **by** *auto*

lemma *map-null*:
shows *map A.null* $\notin$ *map* ‘ *Collect A.arr*
  using *inj-map* **by** *simp*

lemma *map′-map* [*simp*]:
assumes *A.arr t*
shows *map′* (*map t*) = *t*
  using *assms inj-map* **by** *simp*

lemma *map-map′* [*simp*]:
assumes *Arr t*
shows *map* (*map′ t*) = *t*
  using *assms f-inv-into-f* **by** *metis*

sublocale *ResiduatedTransitionSystem.partial-magma resid*
by (*metis ResiduatedTransitionSystem.partial-magma.intro map-null resid-def*)

lemma *null-char*:
shows *null* = *Null*
  by (*metis map-null null-is-zero*(*2*) *resid-def*)

sublocale *residuation resid*
proof
  show $\bigwedge$*t u. t* $\setminus$ *u* $\neq$ *null* $\Longrightarrow$ *u* $\setminus$ *t* $\neq$ *null*
    unfolding *resid-def null-char*
    by (*metis A.arr-resid A.conI A.con-sym imageI map-null mem-Collect-eq*)
  show $\bigwedge$*t u. t* $\setminus$ *u* $\neq$ *null* $\Longrightarrow$ (*t* $\setminus$ *u*) $\setminus$ (*t* $\setminus$ *u*) $\neq$ *null*
    unfolding *resid-def null-char inj-map*
    apply *simp*
    by (*metis CollectI A.arr-resid A.conI A.con-imp-arr-resid imageI*
        map′-map map-null)

show $\bigwedge v\ t\ u.\ (v \setminus t) \setminus (u \setminus t) \neq null$
$\Longrightarrow (v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$
proof −
fix *t u v*
assume *vt-ut*: $(v \setminus t) \setminus (u \setminus t) \neq null$
have *1*: $Arr\ t \wedge Arr\ u \wedge Arr\ v$
using *vt-ut*
by (*metis map-null null-char resid-def*)
have $(v \setminus t) \setminus (u \setminus t) = map\ ((map'\ v \setminus_A map'\ t) \setminus_A (map'\ u \setminus_A map'\ t))$
using *1 null-char resid-def vt-ut*
apply *auto*[*1*]
by (*metis A.arr-resid A.conI map'-map map-null vt-ut*)
also have
$... = map\ ((map'\ v \setminus_A map'\ u) \setminus_A (map'\ t \setminus_A map'\ u))$
using *A.cube* **by** *simp*
also have $... = (v \setminus u) \setminus (t \setminus u)$
using *1 null-char resid-def*
apply *auto*[*1*]
apply (*metis CollectI A.conI image-eqI map'-map map-null A.arr-resid*)
apply (*metis A.conI A.con-implies-arr*(*1*) *image-eqI mem-Collect-eq*)
by (*metis CollectI A.conI A.con-implies-arr*(*2*) *image-eqI*)
finally show $(v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$
by *blast*
qed
qed

notation *con* (**infix** $\frown$ *50*)

lemma *con-char*:
shows $t \frown u \longleftrightarrow Arr\ t \wedge Arr\ u \wedge map'\ t \frown_A map'\ u$
using *null-char con-def inj-map resid-def A.con-def*
by (*metis* (*full-types*) *image-eqI map-null mem-Collect-eq A.arr-resid*)

lemma *arr-char*:
shows $arr\ t \longleftrightarrow Arr\ t$
using *arr-def con-char inj-map* **by** *auto*

lemma $ide\text{-}char_{II}$:
shows $ide\ t \longleftrightarrow Arr\ t \wedge A.ide\ (map'\ t)$
unfolding *ide-def resid-def con-char*
using *inj-map' f-inv-into-f*
by (*metis A.ide-def inv-into-f-f mem-Collect-eq A.arr-resid*)

lemma *trg-char*:
shows $trg\ t = (if\ Arr\ t\ then\ map\ (A.trg\ (map'\ t))\ else\ null)$
unfolding *trg-def resid-def A.trg-def null-char* **by** *simp*

sublocale *rts resid*
proof

```
  show ⋀t. arr t ⟹ ide (trg t)
    using ide-char_II inj-map trg-char trg-def by fastforce
  show 1: ⋀a t. ⟦ide a; t ⌢ a⟧ ⟹ t \ a = t
    by (simp add: A.resid-arr-ide con-char f-inv-into-f ide-char_II resid-def)
  show ⋀a t. ⟦ide a; a ⌢ t⟧ ⟹ ide (a \ t)
    by (metis 1 arrE arr-resid con-sym cube ideE ideI)
  show ⋀t u. t ⌢ u ⟹ ∃a. ide a ∧ a ⌢ t ∧ a ⌢ u
    by (metis (full-types) CollectI A.con-imp-coinitial-ax ide-char_II
        image-eqI inj-image-rts.con-char inj-image-rts-axioms map'-map
        A.ide-implies-arr)
  show ⋀t u v. ⟦ide (t \ u); u ⌢ v⟧ ⟹ t \ u ⌢ v \ u
  proof −
    fix t u v
    assume ide: ide (t \ u)
    assume con: u ⌢ v
    have Arr t ∧ Arr u ∧ Arr v
      using ide con ide-char_II con-char
      by (metis arr-resid-iff-con ide-implies-arr)
    moreover have A.arr (map' t \_A map' u)
      using ide ide-char_II resid-def
      by (meson A.arr-resid arr-resid-iff-con con-char ide-implies-arr)
    moreover have A.arr (map' v \_A map' u)
      using con con-char
      by (meson A.arr-resid A.con-sym)
    ultimately show t \ u ⌢ v \ u
      using ide con ide-char_II con-char resid-def A.con-target by simp
  qed
qed

notation prfx  (infix ≲ 50)
notation cong  (infix ∼ 50)
```

The function *map* and its inverse (both suitably extensionalized) determine an isomorphism between A and its image.

```
abbreviation map_ext
where map_ext t ≡ if A.arr t then map t else null

abbreviation map'_ext
where map'_ext t ≡ if Arr t then map' t else A.null

sublocale Map: simulation A resid map_ext
  using con-char A.con-implies-arr resid-def
  by unfold-locales auto

sublocale Map': simulation resid A map'_ext
  using arr-char con-char resid-def
  by unfold-locales auto

sublocale inverse-simulations resid A map_ext map'_ext
```

using *arr-char map-null null-char*
by *unfold-locales auto*

lemma *invertible-simulation-map*:
shows *invertible-simulation A resid* map_{ext}
using *inverse-simulations-axioms inverse-simulations-sym invertible-simulation-def'*
by *fast*

lemma *invertible-simulation-map'*:
shows *invertible-simulation resid A* map'_{ext}
using *inverse-simulations-axioms inverse-simulations-sym invertible-simulation-def'*
by *fast*

lemma *inj-on-map*:
shows *inj-on* map_{ext} (*Collect A.arr*)
using *induce-bij-betw-arr-sets bij-betw-def* **by** *blast*

lemma *range-map'*:
shows map'_{ext} ` (*Collect arr*) = *Collect A.arr*
by (*metis* (*no-types*, *lifting*) *bij-betw-imp-surj-on*
inverse-simulations.induce-bij-betw-arr-sets
inverse-simulations-axioms inverse-simulations-sym)

lemma $cong\text{-}char_{II}$:
shows $t \sim u \longleftrightarrow Arr\ t \wedge Arr\ u \wedge map'\ t \sim_A map'\ u$
by (*metis* (*full-types*) *Map'.preserves-resid Map.preserves-ide con-def*
$ide\text{-}char_{II}$ *not-arr-null null-char resid-def residuation.ide-implies-arr*
residuation-axioms)

lemma *preserves-weakly-extensional-rts*:
assumes *weakly-extensional-rts A*
shows *weakly-extensional-rts resid*
by (*metis assms cong-char*$_{II}$ *ide-char*$_{II}$ *inv-into-injective rts-axioms*
weakly-extensional-rts.intro weakly-extensional-rts.weak-extensionality
weakly-extensional-rts-axioms.intro)

lemma *preserves-extensional-rts*:
assumes *extensional-rts A*
shows *extensional-rts resid*
proof
interpret *A*: *extensional-rts A*
using *assms* **by** *blast*
show $\bigwedge t\ u.\ t \sim u \Longrightarrow t = u$
using $cong\text{-}char_{II}$ $ide\text{-}char_{II}$
by (*meson A.extensional inv-into-injective*)
qed

lemma *preserves-reflects-small-rts*:
shows *small-rts A* $\longleftrightarrow$ *small-rts resid*

using *induce-bij-betw-arr-sets*
by (*metis* (*no-types*, *lifting*) *A.rts-axioms bij-betw-def rts-axioms small-image-iff small-rts.intro small-rts.small small-rts-axioms-def*)

end

lemma *inj-image-rts-comp*:
fixes $F :: 'a \Rightarrow 'b$ **and** $G :: 'b \Rightarrow 'c$
assumes *inj F* **and** *inj G*
assumes *rts X*
shows *inj-image-rts.resid* $(G \circ F)$ *X* =
inj-image-rts.resid G (*inj-image-rts.resid F X*)
proof −
interpret *X*: *rts X*
using *assms*(*3*) **by** *blast*
interpret *FX*: *inj-image-rts F X*
by (*metis X.rts-axioms assms*(*1*)
inj-image-rts-axioms-def inj-image-rts-def inj-on-subset top-greatest)
interpret *GFX*: *inj-image-rts G FX.resid*
by (*metis* (*mono-tags*, *lifting*) *FX.rts-axioms assms*(*2*) *inj-def*
inj-image-rts-axioms-def inj-image-rts-def inj-onI)
interpret *GoF-X*: *inj-image-rts* ‹*G o F*› *X*
by (*metis* (*no-types*, *opaque-lifting*) *X.rts-axioms*
assms(*1*−*2*) *inj-compose inj-image-rts-axioms-def*
inj-image-rts-def inj-on-subset top-greatest)
show *GoF-X.resid* = *GFX.resid*
proof −
have $\bigwedge t\ u$. *GoF-X.resid t u* = *GFX.resid t u*
unfolding *GoF-X.resid-def GFX.resid-def*
using *FX.arr-char FX.null-char FX.resid-def GoF-X.map'-map* **by** *auto*
thus *?thesis* **by** *blast*
qed
qed

lemma *inj-image-rts-map-comp*:
fixes $F :: 'a \Rightarrow 'b$ **and** $G :: 'b \Rightarrow 'c$
assumes *inj F* **and** *inj G*
assumes *rts X*
shows $\mathit{inj\text{-}image\text{-}rts.map}_{ext}\ (G \circ F)\ X =$
$\mathit{inj\text{-}image\text{-}rts.map}_{ext}\ G\ (\mathit{inj\text{-}image\text{-}rts.resid}\ F\ X) \circ$
$(\mathit{inj\text{-}image\text{-}rts.map}_{ext}\ F\ X)$
and $\mathit{inj\text{-}image\text{-}rts.map'}_{ext}\ (G \circ F)\ X =$
$\mathit{inj\text{-}image\text{-}rts.map'}_{ext}\ F\ X \circ$
$\mathit{inj\text{-}image\text{-}rts.map'}_{ext}\ G\ (\mathit{inj\text{-}image\text{-}rts.resid}\ F\ X)$
proof −
interpret *X*: *rts X*
using *assms*(*3*) **by** *blast*
interpret *FX*: *inj-image-rts F X*
by (*metis X.rts-axioms assms*(*1*) *inj-image-rts-axioms-def inj-image-rts-def*

inj-on-subset top-greatest)
interpret *GFX*: *inj-image-rts G FX.resid*
by (*metis FX.rts-axioms Int-UNIV-right assms*(*2*) *inj-image-rts.intro inj-image-rts-axioms.intro inj-on-Int*)
interpret *GoF-X*: *inj-image-rts* ‹*G o F*› *X*
by (*metis* (*no-types*, *opaque-lifting*) *X.rts-axioms assms*(*1*−*2*) *inj-compose inj-image-rts-axioms-def inj-image-rts-def inj-on-subset top-greatest*)
show $GoF\text{-}X.map_{ext} = GFX.map_{ext} \circ FX.map_{ext}$
using *FX.null-char GFX.null-char GoF-X.null-char* **by** *fastforce*
show $GoF\text{-}X.map'_{ext} = FX.map'_{ext} \circ GFX.map'_{ext}$
using *FX.arr-char FX.not-arr-null GoF-X.map'-map* **by** *fastforce*
qed

3.5 Empty RTS

For any type, there exists an empty RTS having that type as its arrow type. Since types in HOL are nonempty, we may use the guaranteed element *undefined* as the null value.

locale *empty-rts*
begin

definition *resid* :: *'e resid*
where *resid t u* = *undefined*

sublocale *ResiduatedTransitionSystem.partial-magma resid*
by *unfold-locales* (*metis resid-def*)

lemma *null-char*:
shows *null* = *undefined*
by (*metis null-is-zero*(*1*) *resid-def*)

sublocale *residuation resid*
apply *unfold-locales*
by (*metis resid-def*)+

lemma *arr-char*:
shows *arr t* ⟷ *False*
using *null-char resid-def*
by (*metis arrE conE*)

lemma $ide\text{-}char_{ERTS}$:
shows *ide t* ⟷ *False*
using *arr-char* **by** *force*

lemma *con-char*:
shows *con t u* ⟷ *False*
by (*simp add*: *con-def null-char resid-def*)

lemma *trg-char*:
shows *trg t = null*
by (*simp add*: *null-char resid-def trg-def*)

sublocale *rts resid*
apply *unfold-locales*
apply (*metis arr-char*)
by (*metis con-char*)+

lemma *cong-char*$_{ERTS}$:
shows *cong t u* $\longleftrightarrow$ *False*
by (*simp add*: *ide-char*$_{ERTS}$)

sublocale *small-rts resid*
apply *unfold-locales*
by (*metis Collect-empty-eq arr-char small-empty*)

lemma *is-small-rts*:
shows *small-rts resid*
..

sublocale *extensional-rts resid*
by *unfold-locales* (*simp add*: *cong-char*$_{ERTS}$)

lemma *is-extensional-rts*:
shows *extensional-rts resid*
..

lemma *src-char*:
shows *src t = null*
by (*simp add*: *arr-char src-def*)

lemma *prfx-char*:
shows *prfx t u* $\longleftrightarrow$ *False*
using *ide-char*$_{ERTS}$ **by** *metis*

lemma *composite-of-char*:
shows *composite-of t u v* $\longleftrightarrow$ *False*
using *composite-of-def prfx-char* **by** *metis*

lemma *composable-char*:
shows *composable t u* $\longleftrightarrow$ *False*
using *composable-def composite-of-char* **by** *metis*

lemma *seq-char*:
shows *seq t u* $\longleftrightarrow$ *False*
using *arr-char* **by** (*metis seqE*)

sublocale *rts-with-composites resid*

by *unfold-locales* (*simp add*: *composable-char seq-char*)

lemma *is-rts-with-composites*:
shows *rts-with-composites resid*
..

sublocale *extensional-rts-with-composites resid* **..**

lemma *is-extensional-rts-with-composites*:
shows *extensional-rts-with-composites resid*
..

lemma *comp-char*:
shows *comp t u* = *null*
by (*meson composable-char composable-iff-comp-not-null*)

There is a unique simulation from an empty RTS to any other RTS.

definition *initiator* :: *'e resid* ⇒ *'a* ⇒ *'e*
where *initiator A* ≡ (λ*t*. *ResiduatedTransitionSystem.partial-magma.null A*)

lemma *initiator-is-simulation*:
assumes *rts A*
shows *simulation resid A* (*initiator A*)
using *assms initiator-def con-char*
by (*metis rts-axioms simulation-axioms.intro simulation-def*)

lemma *universality*:
assumes *rts A*
shows ∃!*F*. *simulation resid A F*
proof
show *simulation resid A* (*initiator A*)
using *assms initiator-is-simulation* **by** *blast*
show ⋀*F*. *simulation resid A F* ⟹ *F* = *initiator A*
by (*metis HOL.ext initiator-def arr-char simulation.extensional*)
qed

end

3.6 One-Transition RTS

For any type having at least two elements, there exists a one-transition RTS having that type as its arrow type. We use the already-distinguished element *undefined* as the null value and some value distinct from *undefined* as the single transition.

locale *one-arr-rts* =
nondegenerate arr-type
for *arr-type* :: *'t itself*
begin

definition *the-arr* :: *'t*
where *the-arr* ≡ *SOME t*. *t* ≠ *undefined*

definition *resid* :: *'t resid* (**infix** $\setminus_1$ *70*)
where *resid t u* = (*if t* = *the-arr* ∧ *u* = *the-arr then the-arr else undefined*)

sublocale *ResiduatedTransitionSystem.partial-magma resid*
using *resid-def*
by *unfold-locales metis*

lemma *null-char*:
shows *null* = *undefined*
by (*metis null-is-zero*(*1*) *resid-def*)

sublocale *residuation resid*
using *null-char resid-def*
by *unfold-locales metis*+

notation *con* (**infix** $\frown_1$ *50*)

lemma *arr-char*:
shows *arr t* ⟷ *t* = *the-arr*
using *null-char resid-def is-nondegenerate*
by (*metis* (*mono-tags*, *lifting*) *arr-def con-def someI-ex the-arr-def*)

lemma *ide-char*$_{1RTS}$:
shows *ide t* ⟷ *t* = *the-arr*
using *arr-char ide-def resid-def* **by** *auto*

lemma *con-char*:
shows *con t u* ⟷ *arr t* ∧ *arr u*
by (*metis arr-char con-arr-self con-implies-arr*(*2*) *con-sym*)

lemma *trg-char*:
shows *trg t* = (*if arr t then t else null*)
by (*simp add*: *arr-char null-char resid-def trg-def*)

sublocale *rts resid*
using *con-char arr-char ide-char*$_{1RTS}$ *trg-char ideE*
by *unfold-locales metis*+

notation *prfx* (**infix** $\lesssim_1$ *50*)
notation *cong* (**infix** $\sim_1$ *50*)

lemma *cong-char*$_{1RTS}$:
shows *t* $\lesssim_1$ *u* ⟷ *arr t* ∧ *arr u*
using *arr-char ide-char*$_{1RTS}$ *not-ide-null null-char resid-def* **by** *force*

sublocale *extensional-rts resid*
using *arr-char cong-char*$_{1RTS}$
by *unfold-locales auto*

lemma *is-extensional-rts*:
shows *extensional-rts resid*
..

lemma *src-char*:
shows *src t* = (*if t* = *the-arr then t else null*)
using *arr-char ide-char*$_{1RTS}$ *src-def src-ide* **by** *presburger*

lemma *prfx-char*:
shows $t \lesssim_1 u \longleftrightarrow$ *arr t* $\wedge$ *arr u*
by (*metis cong-char*$_{1RTS}$)

lemma *composite-of-char*:
shows *composite-of t u v* $\longleftrightarrow$ *arr t* $\wedge$ *arr u* $\wedge$ *arr v*
using *composite-of-def*
by (*metis composite-of-ide-self prfx-char*)

lemma *composable-char*:
shows *composable t u* $\longleftrightarrow$ *arr t* $\wedge$ *arr u*
using *composable-def composite-of-char* **by** *auto*

lemma *seq-char*:
shows *seq t u* $\longleftrightarrow$ *arr t* $\wedge$ *arr u*
using *arr-char composable-char composable-imp-seq* **by** *auto*

sublocale *rts-with-composites resid*
by *unfold-locales* (*simp add*: *composable-char seq-char*)

lemma *is-rts-with-composites*:
shows *rts-with-composites resid*
..

sublocale *extensional-rts-with-composites resid* ..

lemma *is-extensional-rts-with-composites*:
shows *extensional-rts-with-composites resid*
..

sublocale *small-rts resid*
by (*simp add*: *Collect-cong arr-char rts-axioms small-rts.intro*
small-rts-axioms.intro)

lemma *is-small-rts*:
shows *small-rts resid*
..

lemma *comp-char*:
shows *comp t u* = (*if arr t* ∧ *arr u then the-arr else null*)
using *arr-char composable-iff-comp-not-null trg-char* **by** *auto*

For an arbitrary RTS A, there is a unique simulation from A to the one-transition RTS.

definition *terminator* :: *'a resid* ⇒ *'a* ⇒ *'t*
where *terminator A* ≡ (λ*t*. *if residuation.arr A t then the-arr else null*)

lemma *terminator-is-simulation*:
assumes *rts A*
shows *simulation A resid* (*terminator A*)
proof −
interpret *A*: *rts A*
using *assms* **by** *blast*
show *?thesis*
unfolding *terminator-def*
using *assms ide-char*$_{1RTS}$ *ideE A.con-implies-arr*
apply (*unfold-locales*)
by *auto metis*
qed

lemma *universality*:
assumes *rts A*
shows ∃!*F*. *simulation A resid F*
proof
show *simulation A resid* (*terminator A*)
using *assms terminator-is-simulation* **by** *blast*
show ⋀*F*. *simulation A resid F* ⟹ *F* = *terminator A*
unfolding *terminator-def*
by (*meson arr-char simulation.extensional*
simulation.preserves-reflects-arr)
qed

A "global transition" of an RTS A is a transformation from the one-arrow RTS to A. An important fact is that equality of simulations and of transformations is determined by their compositions with global transitions.

lemma *eq-simulation-iff*:
assumes *weakly-extensional-rts A*
and *simulation A B F* **and** *simulation A B G*
shows *F* = *G* ⟷
(∀ *Q R T*. *transformation resid A Q R T* ⟶ *F* ∘ *T* = *G* ∘ *T*)
proof
interpret *A*: *weakly-extensional-rts A*
using *assms*(*1*) *simulation-def* **by** *blast*
interpret *F*: *simulation A B F*
using *assms*(*2*) **by** *blast*
interpret *G*: *simulation A B G*

```
    using assms(3) by blast
  show F = G ⟹
        ∀ Q R T. transformation resid A Q R T ⟶ F ∘ T = G ∘ T
    by blast
  show ∀ Q R T. transformation resid A Q R T ⟶ F ∘ T = G ∘ T
                ⟹ F = G
  proof −
    have F ≠ G ⟹
          (∃ Q R T. transformation resid A Q R T ∧ F ∘ T ≠ G ∘ T)
    proof −
      assume 1: F ≠ G
      obtain t where t: A.arr t ∧ F t ≠ G t
        using 1 F.extensional G.extensional by fastforce
      interpret T: constant-transformation resid A t
        using t by unfold-locales blast
      have transformation resid A T.F T.G T.map
        using T.transformation-axioms by simp
      moreover have F ∘ T.map ≠ G ∘ T.map
        by (metis (mono-tags, lifting) arr-char comp-apply t)
      ultimately show ∃ Q R T. transformation resid A Q R T ∧
                                F ∘ T ≠ G ∘ T
        by blast
    qed
    thus ∀ Q R T. transformation resid A Q R T ⟶ F ∘ T = G ∘ T
          ⟹ F = G
      by blast
  qed
qed

lemma eq-transformation-iff:
assumes weakly-extensional-rts A and weakly-extensional-rts B
and transformation A B F G U and transformation A B F G V
shows U = V ⟷
      (∀ Q R T. transformation resid A Q R T ⟶ U ∘ T = V ∘ T)
proof
  interpret A: weakly-extensional-rts A
    using assms(1) simulation-def by blast
  interpret B: weakly-extensional-rts B
    using assms(2) simulation-def by blast
  interpret U: transformation A B F G U
    using assms(3) by blast
  interpret V: transformation A B F G V
    using assms(4) by blast
  show U = V ⟹
        ∀ Q R T. transformation resid A Q R T ⟶ U ∘ T = V ∘ T
    by blast
  show ∀ Q R T. transformation resid A Q R T ⟶ U ∘ T = V ∘ T
                ⟹ U = V
  proof −
```

```
    have U ≠ V ⟹
          (∃ Q R T. transformation resid A Q R T ∧ U ∘ T ≠ V ∘ T)
    proof −
      assume 1: U ≠ V
      obtain t where t: A.arr t ∧ U t ≠ V t
        using 1 U.extensional V.extensional by fastforce
      interpret T: constant-transformation resid A t
        using t by unfold-locales blast
      have transformation resid A T.F T.G T.map
        using T.transformation-axioms by simp
      moreover have U ∘ T.map ≠ V ∘ T.map
        by (metis (mono-tags, lifting) arr-char comp-apply t)
      ultimately show ∃ Q R T. transformation resid A Q R T ∧
                                U ∘ T ≠ V ∘ T
        by blast
    qed
    thus ∀ Q R T. transformation resid A Q R T ⟶ U ∘ T = V ∘ T
                  ⟹ U = V
      by blast
  qed
qed

end
```

3.7 Sub-RTS

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 resid  (infix \_R 70)
and Arr :: 'a ⇒ bool +
assumes inclusion: Arr t ⟹ R.arr t
and resid-closed: ⟦Arr t; Arr u; R.con t u⟧ ⟹ Arr (t \_R u)
and enough-sources: ⟦Arr t; Arr u; R.con t u⟧ ⟹
                    ∃ a. Arr a ∧ a ∈ R.sources t ∧ a ∈ R.sources u
begin

  definition resid :: 'a resid  (infix \ 70)
  where resid t u ≡ if Arr t ∧ Arr u ∧ R.con t u then t \_R u else R.null

  sublocale ResiduatedTransitionSystem.partial-magma resid
    using R.not-con-null(2) R.null-is-zero(1) resid-def
    by unfold-locales metis

  lemma null-char:
  shows null = R.null
```

by (*metis R.not-arr-null inclusion null-eqI resid-def*)

sublocale *residuation resid*
using *R.conE R.con-sym R.not-con-null*(*1*) *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 *arr-char*:
shows *arr t* ⟷ *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 *ide t* ⟷ *Arr t* ∧ *R.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* ⟷ *Arr t* ∧ *Arr u* ∧ *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* = (λ*t*. *if arr t then R.trg t 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 *prfx-char*:
shows *prfx t u* ⟷ *Arr t* ∧ *Arr u* ∧ *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*:
shows *cong t u* ⟷ *Arr t* ∧ *Arr u* ∧ *R.cong t u*
using *prfx-char* **by** *blast*

lemma *composite-of-char*:
shows *composite-of t u v* ⟷ *Arr t* ∧ *Arr u* ∧ *Arr v* ∧ *R.composite-of t u v*
proof
show *composite-of t u v* ⟹ *Arr t* ∧ *Arr u* ∧ *Arr v* ∧ *R.composite-of t u v*
by (*metis RTSConstructions.sub-rts.resid-def R.composite-of-def*

```
      R.con-sym composite-ofE con-char prfx-char prfx-implies-con sub-rts-axioms)
    show Arr t ∧ Arr u ∧ Arr v ∧ R.composite-of t u v ⟹ composite-of t u v
     by (metis (full-types) RTSConstructions.sub-rts.resid-def R.composite-of-def
         R.con-sym R.rts-axioms composite-ofI prfx-char resid-closed
         rts.prfx-implies-con sub-rts-axioms)
  qed

  lemma join-of-char:
  shows join-of t u v ⟷ Arr t ∧ Arr u ∧ Arr v ∧ R.join-of t u v
    using composite-of-char
    by (metis R.bounded-imp-con R.join-of-def join-of-def resid-closed resid-def)

  lemma preserves-weakly-extensional-rts:
  assumes weakly-extensional-rts R
  shows weakly-extensional-rts resid
    by (metis assms cong-char ide-char rts-axioms weakly-extensional-rts.intro
     weakly-extensional-rts.weak-extensionality weakly-extensional-rts-axioms.intro)

  lemma preserves-extensional-rts:
  assumes extensional-rts R
  shows extensional-rts resid
    by (meson assms extensional-rts.cong-char extensional-rts.intro
        extensional-rts-axioms.intro prfx-char rts-axioms)

end

locale sub-rts-of-weakly-extensional-rts =
  R: weakly-extensional-rts R +
  sub-rts R Arr
for R :: 'a resid  (infix \_R 70)
and Arr :: 'a ⇒ bool
begin

  sublocale weakly-extensional-rts resid
    using R.weakly-extensional-rts-axioms preserves-weakly-extensional-rts
    by blast

  lemma src-char:
  shows src = (λt. if arr t then R.src t 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** $\setminus_R$ *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*

end

3.8 Fibered Product RTS

locale *fibered-product-rts* =
A: *rts A* +
B: *rts B* +
C: *weakly-extensional-rts C* +
F: *simulation A C F* +
G: *simulation B C G*
for *A* :: *'a resid* (**infix** $\setminus_A$ *70*)
and *B* :: *'b resid* (**infix** $\setminus_B$ *70*)
and *C* :: *'c resid* (**infix** $\setminus_C$ *70*)
and *F* :: *'a* ⇒ *'c*
and *G* :: *'b* ⇒ *'c*
begin

  notation *A.con* (**infix** $\frown_A$ *50*)
  notation *B.con* (**infix** $\frown_B$ *50*)
  notation *C.con* (**infix** $\frown_C$ *50*)
  notation *A.prfx* (**infix** $\lesssim_A$ *50*)
  notation *B.prfx* (**infix** $\lesssim_B$ *50*)
  notation *C.prfx* (**infix** $\lesssim_C$ *50*)
  notation *A.cong* (**infix** $\sim_A$ *50*)
  notation *B.cong* (**infix** $\sim_B$ *50*)
  notation *C.cong* (**infix** $\sim_C$ *50*)

  abbreviation *Arr*
  where *Arr* ≡ λ*tu*. *A.arr* (*fst tu*) ∧ *B.arr* (*snd tu*) ∧ *F* (*fst tu*) = *G* (*snd tu*)

  abbreviation *Ide*
  where *Ide* ≡ λ*tu*. *A.ide* (*fst tu*) ∧ *B.ide* (*snd tu*) ∧ *F* (*fst tu*) = *G* (*snd tu*)

abbreviation *Con*
where $Con \equiv \lambda tu\ vw.\ fst\ tu \frown_A fst\ vw \wedge snd\ tu \frown_B snd\ vw \wedge$
$F\ (fst\ tu) = G\ (snd\ tu) \wedge F\ (fst\ vw) = G\ (snd\ vw)$

definition *resid* :: $('a * 'b)$ *resid* (**infix** $\backslash$ *70*)
where $tu \backslash vw =$
$(if\ Con\ tu\ vw\ then\ (fst\ tu \backslash_A fst\ vw,\ snd\ tu \backslash_B snd\ vw)$
$else\ (A.null,\ B.null))$

sublocale *ResiduatedTransitionSystem.partial-magma resid*
using *resid-def*
by *unfold-locales*
(*metis B.arr-resid-iff-con B.ex-un-null B.not-arr-null*
B.null-is-zero(*2*) *snd-conv*)

lemma *null-char*:
shows $null = (A.null,\ B.null)$
unfolding *null-def*
using *ex-un-null resid-def*
the1-equality [*of* $\lambda n.\ \forall f.\ n \backslash f = n \wedge f \backslash n = n\ (A.null,\ B.null)$]
by *simp*

sublocale *residuation resid*
proof
show $\bigwedge t\ u.\ t \backslash u \neq null \Longrightarrow u \backslash t \neq null$
by (*metis A.con-sym B.con-sym B.residuation-axioms null-char prod.inject*
resid-def residuation.con-def)
show $\bigwedge t\ u.\ t \backslash u \neq null \Longrightarrow (t \backslash u) \backslash (t \backslash u) \neq null$
by (*metis* (*no-types*, *lifting*) *A.conE A.conI A.con-imp-arr-resid B.con-def*
B.con-imp-arr-resid F.preserves-resid G.preserves-resid fst-conv null-char
resid-def snd-conv)
show $\bigwedge v\ t\ u.\ (v \backslash t) \backslash (u \backslash t) \neq null \Longrightarrow (v \backslash t) \backslash (u \backslash t) = (v \backslash u) \backslash (t \backslash u)$
proof −
fix *v t u*
assume *1*: $(v \backslash t) \backslash (u \backslash t) \neq null$
have *2*: $(v \backslash t) \backslash (u \backslash t) =$
$((fst\ v \backslash_A fst\ t) \backslash_A (fst\ u \backslash_A fst\ t),$
$(snd\ v \backslash_B snd\ t) \backslash_B (snd\ u \backslash_B snd\ t))$
using *1 resid-def null-char* **by** *auto*
also have ... $= ((fst\ v \backslash_A fst\ u) \backslash_A (fst\ t \backslash_A fst\ u),$
$(snd\ v \backslash_B snd\ u) \backslash_B (snd\ t \backslash_B snd\ u))$
using *A.cube B.cube* **by** *simp*
also have ... $= (v \backslash u) \backslash (t \backslash u)$
proof −
have $Con\ (v \backslash u)\ (t \backslash u)$
proof −
have $fst\ v \backslash_A fst\ u \frown_A fst\ t \backslash_A fst\ u$
using *1 2 A.cube*
by (*metis* (*no-types*, *lifting*) *A.con-def fst-conv null-char resid-def*)

```
      moreover have snd v \_B snd u ⌢_B snd t \_B snd u
        using 1 2 B.cube
        by (metis (no-types, lifting) B.con-def null-char resid-def snd-conv)
      ultimately show ?thesis
        using 1 resid-def
       by (metis (no-types, lifting) A.conI A.con-implies-arr(1) A.not-arr-null
           A.not-con-null(2) B.conI B.con-implies-arr(1) B.not-arr-null
           B.not-con-null(2) F.preserves-resid G.preserves-resid fst-eqD
           null-char snd-eqD)
    qed
    thus ?thesis
      using resid-def null-char by auto
  qed
  finally show (v \ t) \ (u \ t) = (v \ u) \ (t \ u)
    by blast
 qed
qed

notation con  (infix ⌢ 50)

lemma arr-char:
shows arr t ⟷ Arr t
 by (metis B.arr-def B.conE Pair-inject conE conI null-char resid-def
     A.arr-def arr-def)

lemma con-char:
shows t ⌢ u ⟷ Con t u
 by (metis B.conE null-char resid-def con-def snd-conv)

lemma ide-char_FP:
shows ide t ⟷ Ide t
 unfolding ide-def
 using con-char resid-def
 by (metis (no-types, lifting) A.ide-def fst-conv prod.exhaust-sel
     B.ide-def snd-conv)

lemma trg-char:
shows trg t = (if arr t then (A.trg (fst t), B.trg (snd t)) else null)
 by (simp add: A.trg-def B.trg-def con-char null-char resid-def
     arr-def trg-def)

sublocale rts resid
proof
 show ⋀t. arr t ⟹ ide (trg t)
   using trg-char ide-char_FP arr-char F.preserves-trg G.preserves-trg
   apply simp
   using F.preserves-trg G.preserves-trg by metis
 show 1: ⋀a t. ⟦ide a; t ⌢ a⟧ ⟹ t \ a = t
   by (simp add: A.resid-arr-ide B.resid-arr-ide con-char ide-char_FP
```

resid-def)
show $\bigwedge a\ t.\ [\![ide\ a;\ a \frown t]\!] \Longrightarrow ide\ (a \setminus t)$
by (*metis 1 arr-resid-iff-con con-sym cube ide-def*)
show $\bigwedge t\ u.\ t \frown u \Longrightarrow \exists a.\ ide\ a \wedge a \frown t \wedge a \frown u$
proof −
fix *t u*
assume *con*: $t \frown u$
obtain *a* **where** *a*: $A.ide\ a \wedge a \frown_A fst\ t \wedge a \frown_A fst\ u$
using *con con-char A.con-imp-coinitial-ax* **by** *fastforce*
obtain *b* **where** *b*: $B.ide\ b \wedge b \frown_B snd\ t \wedge b \frown_B snd\ u$
using *con con-char B.con-imp-coinitial-ax* **by** *fastforce*
have $F\ a = G\ b$
by (*metis C.src-eqI F.preserves-con F.preserves-ide G.preserves-con G.preserves-ide a b con con-char*)
thus $\exists a.\ ide\ a \wedge a \frown t \wedge a \frown u$
using *a b ide-char*$_{FP}$ *con con-char* **by** *auto*
qed
show $\bigwedge t\ u\ v.\ [\![ide\ (t \setminus u);\ u \frown v]\!] \Longrightarrow t \setminus u \frown v \setminus u$
by (*metis* (*no-types*, *lifting*) *A.con-target B.con-target arr-char arr-resid-iff-con con-char con-sym fst-conv ide-char*$_{FP}$ *ide-implies-arr resid-def snd-conv*)
qed

notation *prfx* (**infix** $\lesssim$ *50*)
notation *cong* (**infix** $\sim$ *50*)

lemma *prfx-char*:
shows $t \lesssim u \longleftrightarrow F\ (fst\ t) = G\ (snd\ t) \wedge F\ (fst\ u) = G\ (snd\ u) \wedge$
$fst\ t \lesssim_A fst\ u \wedge snd\ t \lesssim_B snd\ u$
using *A.prfx-implies-con B.prfx-implies-con ide-char*$_{FP}$ *resid-def*
by *auto*

lemma *cong-char*$_{FP}$:
shows $t \sim u \longleftrightarrow F\ (fst\ t) = G\ (snd\ t) \wedge F\ (fst\ u) = G\ (snd\ u) \wedge$
$fst\ t \sim_A fst\ u \wedge snd\ t \sim_B snd\ u$
using *prfx-char* **by** *auto*

lemma *sources-char*:
shows $sources\ t =$
$\{a.\ F\ (fst\ t) = G\ (snd\ t) \wedge F\ (fst\ a) = G\ (snd\ a) \wedge$
$fst\ a \in A.sources\ (fst\ t) \wedge snd\ a \in B.sources\ (snd\ t)\}$
using *con-char ide-char*$_{FP}$ *sources-def* **by** *auto*

lemma *targets-char*$_{FP}$:
shows $targets\ t =$
$\{a.\ F\ (fst\ t) = G\ (snd\ t) \wedge F\ (fst\ a) = G\ (snd\ a) \wedge$
$fst\ a \in A.targets\ (fst\ t) \wedge snd\ a \in B.targets\ (snd\ t)\}$
proof
show $targets\ t \subseteq \{a.\ F\ (fst\ t) = G\ (snd\ t) \wedge F\ (fst\ a) = G\ (snd\ a) \wedge$

```
                    fst a ∈ A.targets (fst t) ∧ snd a ∈ B.targets (snd t)}
    using arr-char arr-iff-has-target ide-char_FP
    apply auto[1]
     apply (metis arr-char arr-composite-of composite-of-arr-ide in-targetsE
       trg-def)
     apply (metis A.in-targetsI con-char con-implies-arr(1) fst-conv
       in-targetsE not-arr-null trg-char)
    by (metis B.in-targetsI con-char con-implies-arr(1) in-targetsE
       not-arr-null snd-conv trg-char)
  show {a. F (fst t) = G (snd t) ∧ F (fst a) = G (snd a) ∧
          fst a ∈ A.targets (fst t) ∧ snd a ∈ B.targets (snd t)} ⊆ targets t
    using A.arr-iff-has-target B.arr-iff-has-target arr-char con-char
      ide-char_FP ide-trg targets-def trg-char
    by auto
qed

definition P_0 :: 'a × 'b ⇒ 'b
where P_0 t ≡ if arr t then snd t else B.null

definition P_1 :: 'a × 'b ⇒ 'a
where P_1 t ≡ if arr t then fst t else A.null

sublocale P_0: simulation resid B P_0
  using P_0-def con-char resid-def arr-resid con-implies-arr(1-2)
  by unfold-locales auto

lemma P_0-is-simulation:
shows simulation resid B P_0
  ..

sublocale P_1: simulation resid A P_1
  using P_1-def con-char resid-def arr-resid con-implies-arr(1-2)
  by unfold-locales auto

lemma P_1-is-simulation:
shows simulation resid A P_1
  ..

lemma commutativity:
shows F o P_1 = G o P_0
  using F.extensional G.extensional P_1-def P_0-def arr-char by auto

definition tuple :: 'x resid ⇒ ('x ⇒ 'a) ⇒ ('x ⇒ 'b) ⇒ 'x ⇒ 'a × 'b
where tuple X H K ≡ λt. if residuation.arr X t then (H t, K t) else null

lemma universality:
assumes rts X and simulation X A H and simulation X B K
and F o H = G o K
shows [intro]: simulation X resid (tuple X H K)
```

and $P_1 \circ$ *tuple X H K* $= H$ **and** $P_0 \circ$ *tuple X H K* $= K$
and $\exists!HK.$ *simulation X resid HK* $\wedge$ P_1 *o HK* $= H \wedge P_0$ *o HK* $= K$
proof −
interpret X: *rts X*
using *assms*(*1*) **by** *blast*
interpret H: *simulation X A H*
using *assms*(*2*) **by** *blast*
interpret K: *simulation X B K*
using *assms*(*3*) **by** *blast*
let *?HK* $=$ *tuple X H K*
interpret *HK*: *simulation X resid ?HK*
proof
show $\bigwedge t. \neg$ *X.arr t* $\Longrightarrow$ *?HK t* $=$ *null*
unfolding *tuple-def* **by** *simp*
fix *t u*
assume *con*: *X.con t u*
have *1*: *X.arr t* $\wedge$ *X.arr u*
using *con X.con-implies-arr*(*1*) *X.con-implies-arr*(*2*) **by** *force*
show *2*: *con* (*?HK t*) (*?HK u*)
by (*metis 1 assms*(*4*) *comp-apply con con-char fst-conv tuple-def*
H.preserves-con K.preserves-con snd-conv)
show *?HK* (*X t u*) $=$ *resid* (*?HK t*) (*?HK u*)
using *1 2 con resid-def null-char con-char arr-char*
by (*simp add*: *tuple-def*)
qed
show *simulation X resid* (*tuple X H K*)
..
show $P_1 \circ$ *?HK* $= H$
proof
fix *t*
show ($P_1 \circ$ *?HK*) *t* $= H\ t$
using P_1*-def HK.preserves-reflects-arr*
by (*simp add*: *H.extensional tuple-def*)
qed
moreover show $P_0 \circ$ *?HK* $= K$
proof
fix *t*
show ($P_0 \circ$ *?HK*) *t* $= K\ t$
using P_0*-def HK.preserves-reflects-arr*
by (*simp add*: *K.extensional tuple-def*)
qed
ultimately
have *simulation X resid ?HK* $\wedge$ $P_1 \circ$ *?HK* $= H \wedge P_0 \circ$ *?HK* $= K$
using *HK.simulation-axioms* **by** *simp*
moreover have $\bigwedge M.$ *simulation X resid M* $\wedge$ $P_1 \circ M = H \wedge P_0 \circ M = K$
$\Longrightarrow M =$ *?HK*
unfolding P_1*-def* P_0*-def tuple-def*
using *simulation.extensional simulation.preserves-reflects-arr*
by *fastforce*

ultimately show $\exists!HK.$ *simulation X resid HK* $\wedge$
P_1 *o HK* $= H \wedge P_0$ *o HK* $= K$
by *auto*
qed

lemma *preserves-weakly-extensional-rts*:
assumes *weakly-extensional-rts A* **and** *weakly-extensional-rts B*
shows *weakly-extensional-rts resid*
using *assms*
by *unfold-locales*
(*metis con-char ide-char*$_{FP}$ *prod.exhaust-sel prfx-implies-con*
weakly-extensional-rts.con-ide-are-eq)

lemma *preserves-extensional-rts*:
assumes *extensional-rts A* **and** *extensional-rts B*
shows *extensional-rts resid*
using *assms*
by *unfold-locales*
(*metis* (*no-types*, *lifting*) *extensional-rts.extensional fst-conv ide-char*$_{FP}$
ide-implies-arr not-arr-null null-char prod.exhaust-sel resid-def snd-conv)

lemma *preserves-small-rts*:
assumes *small-rts A* **and** *small-rts B*
shows *small-rts resid*
proof
interpret *A*: *small-rts A*
using *assms*(*1*) **by** *blast*
interpret *B*: *small-rts B*
using *assms*(*2*) **by** *blast*
show *small* (*Collect arr*)
proof −
have *1*: *Collect arr* $\subseteq$ {*t*. *A.arr* (*fst t*) $\wedge$ *B.arr* (*snd t*)}
using *arr-char* **by** *blast*
obtain φ
where φ: *inj-on* φ (*Collect A.arr*) $\wedge$ φ ‘ *Collect A.arr* $\in$ *range elts*
using *A.small small-def* **by** *metis*
obtain ψ
where ψ: *inj-on* ψ (*Collect B.arr*) $\wedge$ ψ ‘ *Collect B.arr* $\in$ *range elts*
using *B.small small-def* **by** *metis*
let *?*$\varphi\psi = \lambda$*ab. vpair* (φ (*fst ab*)) (ψ (*snd ab*))
have *inj-on ?*$\varphi\psi$ (*Collect arr*)
using *1* φ ψ *arr-char inj-on-def* [*of* φ *Collect A.arr*]
inj-on-def [*of* ψ *Collect B.arr*] *prod.expand*
by (*intro inj-onI*) *force*
moreover have *?*$\varphi\psi$ ‘ *Collect arr* $\in$ *range elts*
proof −
have *?*$\varphi\psi$ ‘ *Collect arr* $\subseteq$

```
          elts (vtimes (set (φ ‘ Collect A.arr)) (set (ψ ‘ Collect B.arr)))
        using A.small B.small arr-char by auto
      thus ?thesis
        by (meson down-raw)
    qed
    ultimately show ?thesis
      by (meson small-def)
  qed
qed

end

locale fibered-product-of-weakly-extensional-rts =
  A: weakly-extensional-rts A +
  B: weakly-extensional-rts B +
  fibered-product-rts
begin

  sublocale weakly-extensional-rts resid
    using A.weakly-extensional-rts-axioms B.weakly-extensional-rts-axioms
        preserves-weakly-extensional-rts
    by auto

  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 ¬ arr t ⟹ ?thesis
      using src-def by presburger
    assume t: arr t
    show ?thesis
    proof (intro src-eqI)
      show ide (if arr t then (A.src (fst t), B.src (snd t)) else null)
        using t ide-char_FP
        by simp (metis A.src-eqI B.src-eqI con-arr-src(2) con-char ide-src)
      show (if arr t then (A.src (fst t), B.src (snd t)) else null) ⌢ t
        using t con-char arr-char
        by simp
          (metis A.arr-def A.src-eqI B.arr-def B.src-eqI
              con-imp-coinitial-ax ide-char_FP)
    qed
  qed

end

locale fibered-product-of-extensional-rts =
```

```
  A: extensional-rts A +
  B: extensional-rts B +
  fibered-product-of-weakly-extensional-rts
begin

  sublocale fibered-product-of-weakly-extensional-rts A B ..
  sublocale extensional-rts resid
  using A.extensional-rts-axioms B.extensional-rts-axioms preserves-extensional-rts
    by blast

  lemma is-extensional-rts:
  shows extensional-rts resid
    ..

end

locale fibered-product-of-small-rts =
  A: small-rts A +
  B: small-rts B +
  fibered-product-rts
begin

  sublocale small-rts resid
    by (simp add: A.small-rts-axioms B.small-rts-axioms preserves-small-rts)

  lemma is-small-rts:
  shows small-rts resid
    ..

end
```

3.9 Product RTS

It is possible to define a product construction for RTS's as a special case of the fibered product, but some inconveniences result from that approach. In addition, we have already defined a product construction in *ResiduatedTransitionSystem.ResiduatedTransitionSystem*. So, we will build on that existing construction.

```
definition pointwise-tuple :: ('x ⇒ 'a) ⇒ ('x ⇒ 'b) ⇒ 'x ⇒ 'a × 'b  (⟨⟨-, -⟩⟩)
where pointwise-tuple H K ≡ (λt. (H t, K t))

context product-rts
begin

  definition P₀ :: 'a × 'b ⇒ 'b
  where P₀ t ≡ if arr t then snd t else B.null

  definition P₁ :: 'a × 'b ⇒ 'a
```

where $P_1\ t \equiv$ *if arr t then fst t else A.null*

sublocale P_0: *simulation resid B* P_0
using P_0*-def con-char resid-def arr-resid con-implies-arr*(*1*−*2*)
by *unfold-locales auto*

lemma P_0*-is-simulation*:
shows *simulation resid B* P_0
..

sublocale P_1: *simulation resid A* P_1
using P_1*-def con-char resid-def arr-resid con-implies-arr*(*1*−*2*)
by *unfold-locales auto*

lemma P_1*-is-simulation*:
shows *simulation resid A* P_1
..

abbreviation *tuple* :: $('x \Rightarrow 'a) \Rightarrow ('x \Rightarrow 'b) \Rightarrow 'x \Rightarrow 'a \times 'b$
where *tuple* ≡ *pointwise-tuple*

lemma *universality*:
assumes *simulation X A H* **and** *simulation X B K*
shows [*intro*]: *simulation X resid* $\langle\langle H,\ K\rangle\rangle$
and $P_1 \circ \langle\langle H,\ K\rangle\rangle = H$ **and** $P_0 \circ \langle\langle H,\ K\rangle\rangle = K$
and $\exists!HK.$ *simulation X resid HK* $\wedge$ P_1 *o HK* $= H \wedge P_0$ *o HK* $= K$
proof −
interpret *H*: *simulation X A H*
using *assms*(*1*) **by** *blast*
interpret *K*: *simulation X B K*
using *assms*(*2*) **by** *blast*
interpret *HK*: *simulation X resid* $\langle\langle H,\ K\rangle\rangle$
proof
show $\bigwedge t.\ \neg$ *H.A.arr t* $\Longrightarrow \langle\langle H,\ K\rangle\rangle\ t =$ *null*
by (*simp add*: *H.extensional K.extensional pointwise-tuple-def*)
fix *t u*
assume *con*: *H.A.con t u*
have *1*: *H.A.arr t* $\wedge$ *H.A.arr u*
using *con H.A.con-implies-arr*(*1*−*2*) **by** *force*
show *con* $(\langle\langle H,\ K\rangle\rangle\ t)\ (\langle\langle H,\ K\rangle\rangle\ u)$
using *1 con con-char H.preserves-con K.preserves-con pointwise-tuple-def*
by (*metis fst-conv snd-conv*)
thus $\langle\langle H,\ K\rangle\rangle$ *(X t u) = resid* $(\langle\langle H,\ K\rangle\rangle\ t)\ (\langle\langle H,\ K\rangle\rangle\ u)$
unfolding *pointwise-tuple-def*
using *1 con resid-def null-char con-char arr-char*
by *simp*
qed
show *simulation X resid* $\langle\langle H,\ K\rangle\rangle$ **..**
show $P_1 \circ \langle\langle H,\ K\rangle\rangle = H$

```
    unfolding pointwise-tuple-def
    using P₁-def HK.preserves-reflects-arr
    by (auto simp add: H.extensional)
  moreover show P₀ ∘ ⟨⟨H, K⟩⟩ = K
    unfolding pointwise-tuple-def
    using P₀-def HK.preserves-reflects-arr
    by (auto simp add: K.extensional)
  ultimately have simulation X resid ⟨⟨H, K⟩⟩ ∧
                  P₁ ∘ ⟨⟨H, K⟩⟩ = H ∧ P₀ ∘ ⟨⟨H, K⟩⟩ = K
    using HK.simulation-axioms by simp
  moreover have ⋀M. simulation X resid M ∧ P₁ ∘ M = H ∧ P₀ ∘ M = K
                    ⟹ M = ⟨⟨H, K⟩⟩
  proof
    fix M t
    assume 1: simulation X resid M ∧ P₁ ∘ M = H ∧ P₀ ∘ M = K
    interpret M: simulation X resid M
      using 1 by blast
    show M t = ⟨⟨H, K⟩⟩ t
      using 1 M.extensional M.preserves-reflects-arr
      unfolding P₁-def P₀-def pointwise-tuple-def
      by force
  qed
  ultimately
  show ∃!HK. simulation X resid HK ∧ P₁ o HK = H ∧ P₀ o HK = K
    by auto
qed

lemma proj-joint-monic:
assumes simulation X resid F and simulation X resid G
and P₀ ∘ F = P₀ ∘ G and P₁ ∘ F = P₁ ∘ G
shows F = G
  using assms(1−4) P₀-is-simulation P₁-is-simulation universality(4)
  by blast

lemma tuple-proj:
assumes simulation X resid F
shows ⟨⟨P₁ ∘ F, P₀ ∘ F⟩⟩ = F
  by (meson P₀-is-simulation P₁-is-simulation proj-joint-monic assms
      universality(1−3) simulation-comp)

lemma proj-tuple:
assumes simulation X A F and simulation X B G
shows P₁ ∘ ⟨⟨F, G⟩⟩ = F and P₀ ∘ ⟨⟨F, G⟩⟩ = G
  using assms(1−2) universality(2−3) by auto

lemma preserves-weakly-extensional-rts:
assumes weakly-extensional-rts A and weakly-extensional-rts B
shows weakly-extensional-rts resid
  by (metis assms(1−2) ide-char prfx-char prod.exhaust-sel rts-axioms
```

```
    weakly-extensional-rts.intro weakly-extensional-rts.weak-extensionality
    weakly-extensional-rts-axioms.intro)

lemma preserves-extensional-rts:
assumes extensional-rts A and extensional-rts B
shows extensional-rts resid
proof −
  interpret A: extensional-rts A
    using assms(1) by blast
  interpret B: extensional-rts B
    using assms(2) by blast
  show ?thesis
    by unfold-locales
      (metis A.extensional B.extensional cong-char prod.collapse)
qed

lemma preserves-small-rts:
assumes small-rts A and small-rts B
shows small-rts resid
proof
  interpret A: small-rts A
    using assms(1) by blast
  interpret B: small-rts B
    using assms(2) by blast
  show small (Collect arr)
  proof −

    interpret One: one-arr-rts ‹TYPE(bool)›
      by unfold-locales auto
    interpret simulation A One.resid ‹One.terminator A›
      using One.terminator-is-simulation A.rts-axioms by blast
    interpret simulation B One.resid ‹One.terminator B›
      using One.terminator-is-simulation B.rts-axioms by blast
    interpret AxB: fibered-product-of-small-rts A B One.resid
                     ‹One.terminator A› ‹One.terminator B› ..
    have Collect arr ⊆ Collect AxB.arr
      using AxB.arr-char arr-char One.terminator-def
      by (metis Collect-mono)
    moreover have small (Collect AxB.arr)
      using AxB.small by blast
    ultimately show ?thesis
      using smaller-than-small by blast
  qed
qed

end

locale product-of-extensional-rts =
  A: extensional-rts A +
```

```
  B: extensional-rts B +
  product-rts
begin

  sublocale product-of-weakly-extensional-rts A B ..

  sublocale extensional-rts resid
  using A.extensional-rts-axioms B.extensional-rts-axioms preserves-extensional-rts
    by blast

  lemma is-extensional-rts:
  shows extensional-rts resid
    ..

  lemma proj-tuple2:
  assumes transformation X A F G S and transformation X B H K T
  shows P1 ∘ ⟨⟨S, T⟩⟩ = S and P0 ∘ ⟨⟨S, T⟩⟩ = T
  proof −
    interpret S: transformation X A F G S
      using assms(1) by blast
    interpret T: transformation X B H K T
      using assms(2) by blast
    show P1 ∘ ⟨⟨S, T⟩⟩ = S
    proof
      fix t
      show (P1 ∘ ⟨⟨S, T⟩⟩) t = S t
        unfolding pointwise-tuple-def P1-def
        using S.extensional S.preserves-arr T.preserves-arr
        apply auto[1]
        by metis+
    qed
    show P0 ∘ ⟨⟨S, T⟩⟩ = T
    proof
      fix t
      show (P0 ∘ ⟨⟨S, T⟩⟩) t = T t
        unfolding pointwise-tuple-def P0-def
        using T.extensional S.preserves-arr T.preserves-arr
        apply auto[1]
        by metis+
    qed
  qed

  lemma universality2:
  assumes simulation X resid F and simulation X resid G
  and transformation X A (P1 ∘ F) (P1 ∘ G) S
  and transformation X B (P0 ∘ F) (P0 ∘ G) T
  shows [intro]: transformation X resid F G ⟨⟨S, T⟩⟩
  and P1 ∘ ⟨⟨S, T⟩⟩ = S and P0 ∘ ⟨⟨S, T⟩⟩ = T
  and ∃!ST. transformation X resid F G ST ∧ P1 o ST = S ∧ P0 o ST = T
```

proof −
interpret X: *rts X*
using *assms(1) simulation-def* **by** *auto*
interpret A: *weakly-extensional-rts A*
using *assms(3) transformation-def* **by** *blast*
interpret B: *weakly-extensional-rts B*
using *assms(4) transformation-def* **by** *blast*
interpret X: *weakly-extensional-rts X*
using *assms(4) transformation-def* **by** *blast*
interpret F: *simulation X resid F*
using *assms(1)* **by** *blast*
interpret F: *simulation-to-weakly-extensional-rts X resid F* **..**
interpret G: *simulation X resid G*
using *assms(2)* **by** *blast*
interpret G: *simulation-to-weakly-extensional-rts X resid G* **..**
interpret $P_1 o F$: *composite-simulation X resid A F* P_1 **..**
interpret $P_1 o F$: *simulation-to-weakly-extensional-rts X A* $P_1 o F.map$ **..**
interpret $P_1 o G$: *composite-simulation X resid A G* P_1 **..**
interpret $P_1 o G$: *simulation-to-weakly-extensional-rts X A* $P_1 o F.map$ **..**
interpret $P_0 o F$: *composite-simulation X resid B F* P_0 **..**
interpret $P_0 o F$: *simulation-to-weakly-extensional-rts X A* $P_1 o F.map$ **..**
interpret $P_0 o G$: *composite-simulation X resid B G* P_0 **..**
interpret $P_0 o F$: *simulation-to-weakly-extensional-rts X A* $P_1 o F.map$ **..**
interpret S: *transformation X A* $\langle P_1 \circ F \rangle$ $\langle P_1 \circ G \rangle$ S
using *assms(3)* **by** *blast*
interpret S: *transformation-to-extensional-rts X A* $\langle P_1 \circ F \rangle$ $\langle P_1 \circ G \rangle$ S **..**
interpret T: *transformation X B* $\langle P_0 \circ F \rangle$ $\langle P_0 \circ G \rangle$ T
using *assms(4)* **by** *blast*
interpret T: *transformation-to-extensional-rts X B* $\langle P_0 \circ F \rangle$ $\langle P_0 \circ G \rangle$ T **..**
interpret ST: *transformation X resid F G* $\langle\langle S,\ T \rangle\rangle$
proof
show $\bigwedge t.\ \neg\ X.arr\ t \Longrightarrow \langle\langle S,\ T \rangle\rangle\ t = null$
by (*simp add*: *S.extensional T.extensional pointwise-tuple-def*)
fix t
assume t: *X.ide t*
have *arr*: *arr* $(\langle\langle S,\ T \rangle\rangle\ t)$
unfolding *pointwise-tuple-def*
using *t S.preserves-arr T.preserves-arr* **by** *auto*
show *src* $(\langle\langle S,\ T \rangle\rangle\ t) = F\ t$
proof −
have $F\ t = ((P_1 \circ F)\ t, (P_0 \circ F)\ t)$
using *t tuple-proj F.simulation-axioms pointwise-tuple-def*
by *metis*
thus *?thesis*
using *arr src-char arr-char*
by (*simp add*: *S.preserves-src T.preserves-src t pointwise-tuple-def*)
qed
show *trg* $(\langle\langle S,\ T \rangle\rangle\ t) = G\ t$
proof −

```
  have G t = ((P₁ ∘ G) t, (P₀ ∘ G) t)
    using t tuple-proj G.simulation-axioms pointwise-tuple-def
    by metis
  thus ?thesis
    using arr trg-char arr-char
    by (simp add: S.preserves-trg T.preserves-trg t pointwise-tuple-def)
qed
next
fix t
assume t: X.arr t
have con1: S (X.src t) ⌢_A P₁ (F t)
proof −
  have S t = A.join (S (X.src t)) (P₁ (F t))
    using t S.naturality3'_E by auto
  hence A.prfx (S (X.src t)) (S t) ∧ A.prfx (P₁ (F t)) (S t)
    using t A.join-sym A.arr-prfx-join-self S.preserves-arr
          A.joinable-iff-arr-join
    by metis
  thus ?thesis
    using t S.preserves-arr A.con-arr-self A.con-prfx(1−2) by blast
qed
moreover have con0: T (X.src t) ⌢_B P₀ (F t)
proof −
  have T t = B.join (T (X.src t)) (P₀ (F t))
    using t T.naturality3'_E by auto
  hence B.prfx (T (X.src t)) (T t) ∧ B.prfx (P₀ (F t)) (T t)
    using t B.join-sym B.arr-prfx-join-self T.preserves-arr
          B.joinable-iff-arr-join
    by metis
  thus ?thesis
    using t T.preserves-arr B.con-arr-self B.con-prfx(1−2) by blast
qed
show ⟨⟨S, T⟩⟩ (X.src t) \ F t = ⟨⟨S, T⟩⟩ (X.trg t)
proof −
  have ⟨⟨S, T⟩⟩ (X.src t) \ F t =
       (S (X.src t), T (X.src t)) \ (P₁ (F t), P₀ (F t))
    using t tuple-proj [of X F] F.simulation-axioms
          pointwise-tuple-def [of S T]
          pointwise-tuple-def [of P₁oF.map P₀oF.map]
    apply auto[1]
    by metis
  also have ... = (S (X.src t) \_A P₁ (F t), T (X.src t) \_B P₀ (F t))
    using t con0 con1 resid-def by auto
  also have ... = (S (X.trg t), T (X.trg t))
    using t S.naturality1 T.naturality1 by auto
  also have ... = ⟨⟨S, T⟩⟩ (X.trg t)
    using t pointwise-tuple-def by metis
  finally show ?thesis by blast
qed
```

```
  show F t \ ⟨⟨S, T⟩⟩ (X.src t) = G t
  proof −
    have F t \ ⟨⟨S, T⟩⟩ (X.src t) =
          (P₁ (F t), P₀ (F t)) \ (S (X.src t), T (X.src t))
      using t tuple-proj [of X F] F.simulation-axioms
            pointwise-tuple-def [of S T]
            pointwise-tuple-def [of P₁oF.map P₀oF.map]
      apply auto[1]
      by metis
    also have ... = (P₁ (F t) \_A S (X.src t), P₀ (F t) \_B T (X.src t))
      using t con0 con1 resid-def A.con-sym B.con-sym by auto
    also have ... = (P₁oG.map t, P₀oG.map t)
      using t S.naturality2 T.naturality2 by auto
    also have ... = G t
      using t tuple-proj G.simulation-axioms pointwise-tuple-def by metis
    finally show ?thesis by blast
  qed
  show join-of (⟨⟨S, T⟩⟩ (X.src t)) (F t) (⟨⟨S, T⟩⟩ t)
  proof −
    have A.join-of (S (X.src t)) (P₁oF.map t) (S t)
      using t con1 S.naturality3 [of t] by blast
    moreover have B.join-of (T (X.src t)) (P₀oF.map t) (T t)
      using t con0 T.naturality3 [of t] by blast
    ultimately show ?thesis
      unfolding pointwise-tuple-def
      using t join-of-char(1−2) F.preserves-reflects-arr P₀-def P₁-def
      by auto
  qed
qed
show 1: transformation X (\) F G ⟨⟨S, T⟩⟩ ..
show 2: P₁ ∘ ⟨⟨S, T⟩⟩ = S and 3: P₀ ∘ ⟨⟨S, T⟩⟩ = T
  using proj-tuple2 S.transformation-axioms T.transformation-axioms
  by blast+
show ∃!ST. transformation X (\) F G ST ∧ P₁ ∘ ST = S ∧ P₀ ∘ ST = T
proof −
  have ⋀ST. ⟦transformation X (\) F G ST; P₁ ∘ ST = S; P₀ ∘ ST = T⟧
              ⟹ ST = ⟨⟨S, T⟩⟩
  proof −
    fix ST
    assume ST: transformation X resid F G ST
    assume 0: P₀ ∘ ST = T
    assume 1: P₁ ∘ ST = S
    interpret ST: transformation X resid F G ST
      using ST by blast
    show ST = ⟨⟨S, T⟩⟩
    proof
      fix t
      show ST t = ⟨⟨S, T⟩⟩ t
        unfolding pointwise-tuple-def
```

```
          using 0 1 ST.preserves-arr ST.extensional P₀-def P₁-def
          apply auto[1]
          by (metis prod.exhaust-sel)
      qed
    qed
    thus ?thesis
      using 1 2 3 by metis
  qed
qed

lemma proj-joint-monic2:
assumes transformation X resid F G S and transformation X resid F G T
and P₀ ∘ S = P₀ ∘ T and P₁ ∘ S = P₁ ∘ T
shows S = T
  using assms transformation-whisker-left [of X resid F G S] universality2(4)
        A.weakly-extensional-rts-axioms B.weakly-extensional-rts-axioms
        P₁.simulation-axioms P₀.simulation-axioms
  by (metis transformation-def)

lemma join-char:
shows join t u =
       (if joinable t u
        then (A.join (fst t) (fst u), B.join (snd t) (snd u))
        else null)
  by (metis A.join-is-join-of B.join-is-join-of fst-conv join-is-join-of
      join-of-char(1−2) join-of-unique joinable-iff-join-not-null snd-conv)

lemma join-simp:
assumes joinable t u
shows join t u = (A.join (fst t) (fst u), B.join (snd t) (snd u))
  using assms join-char by auto

end

locale product-of-small-rts =
  A: small-rts A +
  B: small-rts B +
  product-rts
begin

  sublocale small-rts resid
    using A.small-rts-axioms B.small-rts-axioms preserves-small-rts
    by blast

  lemma is-small-rts:
  shows small-rts resid
    ..

end
```

lemma *simulation-tuple* [*intro*]:
assumes *simulation X A H* **and** *simulation X B K*
and *Y* = *product-rts.resid A B*
shows *simulation X Y* ⟨⟨*H*, *K*⟩⟩
using *assms product-rts.universality*(*1*) [*of A B X H K*]
by (*simp add*: *product-rts.intro simulation-def*)

lemma *simulation-product* [*intro*]:
assumes *simulation A B H* **and** *simulation C D K*
and *X* = *product-rts.resid A C* **and** *Y* = *product-rts.resid B D*
shows *simulation X Y* (*product-simulation.map A C H K*)
using *assms*
by (*meson product-rts-def product-rts-def product-simulation.intro*
product-simulation.is-simulation simulation-def simulation-def)

lemma *comp-pointwise-tuple*:
shows ⟨⟨*H*, *K*⟩⟩ ∘ *L* = ⟨⟨*H* ∘ *L*, *K* ∘ *L*⟩⟩
unfolding *pointwise-tuple-def*
by *auto*

lemma *comp-product-simulation-tuple2*:
assumes *simulation A A′ F* **and** *simulation B B′ G*
and *transformation X A* H_0 H_1 *H* **and** *transformation X B* K_0 K_1 *K*
shows *product-simulation.map A B F G* ∘ ⟨⟨*H*, *K*⟩⟩ = ⟨⟨*F* ∘ *H*, *G* ∘ *K*⟩⟩
proof −
interpret *X*: *weakly-extensional-rts X*
using *assms*(*3*) *transformation.axioms*(*1*) **by** *blast*
interpret *A*: *rts A*
using *assms*(*1*) *simulation.axioms*(*1*) **by** *blast*
interpret *B*: *rts B*
using *assms*(*2*) *simulation.axioms*(*1*) **by** *blast*
interpret *A′*: *rts A′*
using *assms*(*1*) *simulation.axioms*(*2*) **by** *blast*
interpret *B′*: *rts B′*
using *assms*(*2*) *simulation.axioms*(*2*) **by** *blast*
interpret *AxB*: *product-rts A B* **..**
interpret *A′xB′*: *product-rts A′ B′* **..**
interpret *F*: *simulation A A′ F*
using *assms* **by** *blast*
interpret *G*: *simulation B B′ G*
using *assms* **by** *blast*
interpret *FxG*: *product-simulation A B A′ B′ F G* **..**
interpret *H*: *transformation X A* H_0 H_1 *H*
using *assms*(*3*) **by** *blast*
interpret *K*: *transformation X B* K_0 K_1 *K*
using *assms*(*4*) **by** *blast*
show *?thesis*
proof

```
    fix x
    show (FxG.map ∘ ⟨⟨H, K⟩⟩) x = ⟨⟨F ∘ H, G ∘ K⟩⟩ x
      using FxG.extensional pointwise-tuple-def F.extensional G.extensional
            H.extensional K.extensional H.preserves-arr K.preserves-arr
      by (cases X.arr x) (auto simp add: pointwise-tuple-def)
  qed
qed

lemma comp-product-simulation-tuple:
assumes simulation A A' F and simulation B B' G
and simulation X A H and simulation X B K
shows product-simulation.map A B F G ∘ ⟨⟨H, K⟩⟩ = ⟨⟨F ∘ H, G ∘ K⟩⟩
proof −

  interpret X: rts X
    using assms(3) simulation.axioms(1) by blast
  interpret A: rts A
    using assms(1) simulation.axioms(1) by blast
  interpret B: rts B
    using assms(2) simulation.axioms(1) by blast
  interpret A': rts A'
    using assms(1) simulation.axioms(2) by blast
  interpret B': rts B'
    using assms(2) simulation.axioms(2) by blast
  interpret AxB: product-rts A B ..
  interpret A'xB': product-rts A' B' ..
  interpret F: simulation A A' F
    using assms by blast
  interpret G: simulation B B' G
     using assms by blast
  interpret FxG: product-simulation A B A' B' F G ..
  interpret H: simulation X A H
    using assms(3) by blast
  interpret K: simulation X B K
    using assms(4) by blast
  show ?thesis
  proof
    fix x
    show (FxG.map ∘ ⟨⟨H, K⟩⟩) x = ⟨⟨F ∘ H, G ∘ K⟩⟩ x
      using FxG.extensional pointwise-tuple-def F.extensional G.extensional
            H.extensional K.extensional H.preserves-reflects-arr
            K.preserves-reflects-arr
      by (cases X.arr x) (auto simp add: pointwise-tuple-def)
  qed
qed



locale product-transformation =
```

```
  A1: weakly-extensional-rts A1 +
  A0: weakly-extensional-rts A0 +
  B1: extensional-rts B1 +
  B0: extensional-rts B0 +
  A1xA0: product-rts A1 A0 +
  B1xB0: product-rts B1 B0 +
  F1: simulation A1 B1 F1 +
  F0: simulation A0 B0 F0 +
  G1: simulation A1 B1 G1 +
  G0: simulation A0 B0 G0 +
  T1: transformation A1 B1 F1 G1 T1 +
  T0: transformation A0 B0 F0 G0 T0
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
and G1 :: 'a1 ⇒ 'b1
and G0 :: 'a0 ⇒ 'b0
and T1 :: 'a1 ⇒ 'b1
and T0 :: 'a0 ⇒ 'b0
begin

  sublocale F1: simulation-to-weakly-extensional-rts A1 B1 F1 ..
  sublocale F0: simulation-to-weakly-extensional-rts A0 B0 F0 ..
  sublocale G1: simulation-to-weakly-extensional-rts A1 B1 G1 ..
  sublocale G0: simulation-to-weakly-extensional-rts A0 B0 G0 ..

  sublocale A1xA0: product-of-weakly-extensional-rts A1 A0 ..
  sublocale B1xB0: product-of-extensional-rts B1 B0 ..
  sublocale F1xF0: product-simulation A1 A0 B1 B0 F1 F0 ..
  sublocale G1xG0: product-simulation A1 A0 B1 B0 G1 G0 ..

  abbreviation (input) map₀ :: 'a1 × 'a0 ⇒ 'b1 × 'b0
  where map₀ ≡ (λa. (T1 (fst a), T0 (snd a)))

  sublocale TC: transformation-by-components
                A1xA0.resid B1xB0.resid F1xF0.map G1xG0.map map₀
  proof
    show ⋀a. A1xA0.ide a ⟹ B1xB0.src (map₀ a) = F1xF0.map a
      unfolding F1xF0.map-def
      using T1.preserves-arr T0.preserves-arr T1.preserves-src T0.preserves-src
            B1xB0.src-char
      by auto
    show ⋀a. A1xA0.ide a ⟹ B1xB0.trg (map₀ a) = G1xG0.map a
      unfolding G1xG0.map-def
      using T1.preserves-arr T0.preserves-arr T1.preserves-trg T0.preserves-trg
            B1xB0.trg-char
```

```
    by auto
  fix t
  assume t: A1xA0.arr t
  show B1xB0.resid (map₀ (A1xA0.src t)) (F1xF0.map t) =
       map₀ (A1xA0.trg t)
    unfolding F1xF0.map-def
    using t B1xB0.resid-def A1xA0.src-char A1xA0.trg-char
          T1.naturality1 T0.naturality1 A1.con-arr-src(2) T1.preserves-con(2)
          A0.con-arr-src(2) T0.preserves-con(2)
    by auto
  show B1xB0.resid (F1xF0.map t) (map₀ (A1xA0.src t)) =
       G1xG0.map t
    unfolding F1xF0.map-def G1xG0.map-def
    using t B1xB0.resid-def A1xA0.src-char A1xA0.trg-char
          T1.naturality2 T0.naturality2
    apply auto[1]
     apply (metis B1.conI B1.not-arr-null G1.preserves-reflects-arr)
    by (metis B0.conI B0.not-arr-null G0.preserves-reflects-arr)
  show B1xB0.joinable (map₀ (A1xA0.src t)) (F1xF0.map t)
    unfolding F1xF0.map-def
    using t A1xA0.src-char T1.naturality3 T0.naturality3
    apply auto[1]
    by (metis B0.joinable-def B1.joinable-def B1xB0.join-of-char(2)
       fst-eqD snd-eqD)
qed

definition map :: 'a1 × 'a0 ⇒ 'b1 × 'b0
where map ≡ TC.map

sublocale transformation
          A1xA0.resid B1xB0.resid F1xF0.map G1xG0.map map
  unfolding map-def ..

lemma is-transformation:
shows transformation A1xA0.resid B1xB0.resid F1xF0.map G1xG0.map map
  ..

lemma map-simp:
shows map t = B1xB0.join (map₀ (A1xA0.src t)) (F1xF0.map t)
  unfolding map-def TC.map-def by blast

lemma map-simp-ide:
assumes A1xA0.ide t
shows map t = map₀ (A1xA0.src t)
  unfolding map-def
  using assms TC.map-simp-ide by simp

end
```

lemma *comp-product-transformation-tuple*:
assumes *transformation-to-extensional-rts A1 B1 F1 G1 T1*
and *transformation-to-extensional-rts A0 B0 F0 G0 T0*
and *simulation X A1 H1* **and** *simulation X A0 H0*
shows *product-transformation.map A1 A0 B1 B0 F1 F0 T1 T0* $\circ$ $\langle\langle H1, H0\rangle\rangle$ =
$\langle\langle T1 \circ H1, T0 \circ H0\rangle\rangle$
proof −
interpret *X*: *rts X*
using *assms*(*3*) *simulation.axioms*(*1*) **by** *blast*
interpret *A1*: *weakly-extensional-rts A1*
using *assms*(*1*) *transformation-to-extensional-rts.axioms*(*1*)
transformation.axioms(*1*)
by *blast*
interpret *A0*: *weakly-extensional-rts A0*
using *assms*(*2*) *transformation-to-extensional-rts.axioms*(*1*)
transformation.axioms(*1*)
by *blast*
interpret *B1*: *extensional-rts B1*
using *assms*(*1*) *transformation-to-extensional-rts.axioms*(*2*) **by** *blast*
interpret *B0*: *extensional-rts B0*
using *assms*(*2*) *transformation-to-extensional-rts.axioms*(*2*) **by** *blast*
interpret *F1*: *simulation A1 B1 F1*
using *assms*(*1*) *transformation-to-extensional-rts.axioms*(*1*)
transformation.axioms(*3*) *simulation.axioms*(*3*)
by *blast*
interpret *F0*: *simulation A0 B0 F0*
using *assms*(*2*) *transformation-to-extensional-rts.axioms*(*1*)
transformation.axioms(*3*) *simulation.axioms*(*3*)
by *blast*
interpret *G1*: *simulation A1 B1 G1*
using *assms*(*1*) *transformation-to-extensional-rts.axioms*(*1*)
transformation.axioms(*4*) *simulation.axioms*(*3*)
by *blast*
interpret *G0*: *simulation A0 B0 G0*
using *assms*(*2*) *transformation-to-extensional-rts.axioms*(*1*)
transformation.axioms(*4*) *simulation.axioms*(*3*)
by *blast*
interpret *T1*: *transformation-to-extensional-rts A1 B1 F1 G1 T1*
using *assms*(*1*) **by** *blast*
interpret *T0*: *transformation-to-extensional-rts A0 B0 F0 G0 T0*
using *assms*(*2*) **by** *blast*
interpret *A1xA0*: *product-of-weakly-extensional-rts A1 A0* **..**
interpret *B1xB0*: *product-of-extensional-rts B1 B0* **..**
interpret *F1xF0*: *product-simulation A1 A0 B1 B0 F1 F0* **..**
interpret *G1xG0*: *product-simulation A1 A0 B1 B0 G1 G0* **..**
interpret *T1xT0*: *product-transformation A1 A0 B1 B0 F1 F0 G1 G0 T1 T0*
..
interpret *H1*: *simulation X A1 H1*
using *assms*(*3*) **by** *blast*

```
  interpret H0: simulation X A0 H0
    using assms(4) by blast
  show ?thesis
  proof
    fix x
    show (T1xT0.map ∘ ⟨⟨H1, H0⟩⟩) x = ⟨⟨T1 ∘ H1, T0 ∘ H0⟩⟩ x
    proof −
      have X.arr x ⟹
             B1xB0.join
               (T1 (fst (A1xA0.src (H1 x, H0 x))),
                T0 (snd (A1xA0.src (H1 x, H0 x))))
               (F1 (H1 x), F0 (H0 x)) =
           (T1 (H1 x), T0 (H0 x))
      proof −
        assume x: X.arr x
        have B1xB0.join
               (T1 (fst (A1xA0.src (H1 x, H0 x))),
                T0 (snd (A1xA0.src (H1 x, H0 x))))
               (F1 (H1 x), F0 (H0 x)) =
             B1xB0.join
               (T1 (A1.src (H1 x)), T0 (A0.src (H0 x)))
               (F1 (H1 x), F0 (H0 x))
          using x A1xA0.src-char by auto
        also have ... = (T1 (H1 x), T0 (H0 x))
          using x B1xB0.join-char
          by (simp add: B1xB0.join-of-char(2) T0.naturality3'_E(1−2)
              T1.naturality3'_E(1−2))
        finally show ?thesis by blast
      qed
      thus ?thesis
        using pointwise-tuple-def H1.extensional H0.extensional
              T1.extensional T0.extensional T1xT0.extensional T1xT0.map-def
              T1xT0.TC.map-def comp-product-simulation-tuple
              F1xF0.product-simulation-axioms G1xG0.product-simulation-axioms
        by (cases X.arr x) (auto simp add: pointwise-tuple-def)
    qed
  qed
qed

lemma simulation-interchange:
assumes simulation A A' F and simulation B B' G
and simulation A' A'' F' and simulation B' B'' G'
shows product-simulation.map A B (F' ∘ F) (G' ∘ G) =
      product-simulation.map A' B' F' G' ∘ product-simulation.map A B F G
proof −
  interpret F: simulation A A' F
    using assms(1) by blast
  interpret G: simulation B B' G
    using assms(2) by blast
```

interpret F': *simulation* A' A'' F'
using *assms*(3) **by** *blast*
interpret G': *simulation* B' B'' G'
using *assms*(4) **by** *blast*
interpret $F'oF$: *composite-simulation* A A' A'' F F' **..**
interpret $G'oG$: *composite-simulation* B B' B'' G G' **..**
interpret FxG: *product-simulation* A B A' B' F G **..**
interpret $F'xG'$: *product-simulation* A' B' A'' B'' F' G' **..**
interpret $F'oFxG'oG$: *product-simulation* A B A'' B'' $\langle F' \circ F \rangle$ $\langle G' \circ G \rangle$ **..**
show $F'oFxG'oG.map = F'xG'.map \circ FxG.map$
unfolding $F'oFxG'oG.map\text{-}def$ $FxG.map\text{-}def$ $F'xG'.map\text{-}def$
using $F.extensional$ $G.extensional$ $F'.extensional$ $G'.extensional$ **by** *auto*
qed

3.9.1 Associators

For any RTS's A, B, and C, there exists an invertible "associator" simulation from the product RTS $(A \times B) \times C)$ to the product RTS $A \times (B \times C)$.

locale *ASSOC* =
A: *rts* A +
B: *rts* B +
C: *rts* C
for A :: $'a$ *resid*
and B :: $'b$ *resid*
and C :: $'c$ *resid*
begin

sublocale AxB: *product-rts* A B **..**
sublocale BxC: *product-rts* B C **..**
sublocale $AxB\text{-}xC$: *product-rts* $AxB.resid$ C **..**
sublocale $Ax\text{-}BxC$: *product-rts* A $BxC.resid$ **..**

The following definition is expressed in a form that makes it evident that it defines a simulation.

definition *map* :: $('a \times 'b) \times 'c \Rightarrow 'a \times 'b \times 'c$
where $map \equiv Ax\text{-}BxC.tuple$
$(AxB.P_1 \circ AxB\text{-}xC.P_1)$
$(BxC.tuple\ (AxB.P_0 \circ AxB\text{-}xC.P_1)\ AxB\text{-}xC.P_0)$

sublocale *simulation* $AxB\text{-}xC.resid$ $Ax\text{-}BxC.resid$ *map*
unfolding *map-def*
using $AxB.P_0.simulation\text{-}axioms$ $AxB.P_1.simulation\text{-}axioms$
$AxB\text{-}xC.P_0.simulation\text{-}axioms$ $AxB\text{-}xC.P_1.simulation\text{-}axioms$
by (*intro simulation-tuple simulation-comp*) *auto*

lemma *is-simulation*:
shows *simulation* $AxB\text{-}xC.resid$ $Ax\text{-}BxC.resid$ *map*
..

The following explicit formula is more convenient for calculations.

lemma *map-eq*:
shows *map* = (λ*x*. *if AxB-xC.arr x*
then (*fst* (*fst x*), (*snd* (*fst x*), *snd x*))
else Ax-BxC.null)
unfolding *map-def pointwise-tuple-def AxB.P*$_0$*-def AxB.P*$_1$*-def*
AxB-xC.P$_0$*-def AxB-xC.P*$_1$*-def*
by *auto*

definition *map′* :: *′a* × *′b* × *′c* ⇒ (*′a* × *′b*) × *′c*
where *map′* ≡ *AxB-xC.tuple*
(*AxB.tuple Ax-BxC.P*$_1$ (*BxC.P*$_1$ ∘ *Ax-BxC.P*$_0$))
(*BxC.P*$_0$ ∘ *Ax-BxC.P*$_0$)

sublocale *inv*: *simulation Ax-BxC.resid AxB-xC.resid map′*
unfolding *map′-def*
using *BxC.P*$_0$*.simulation-axioms BxC.P*$_1$*.simulation-axioms*
Ax-BxC.P$_0$*.simulation-axioms Ax-BxC.P*$_1$*.simulation-axioms*
by (*intro simulation-tuple simulation-comp*) *auto*

lemma *inv-is-simulation*:
shows *simulation Ax-BxC.resid AxB-xC.resid map′*
..

lemma *map′-eq*:
shows *map′* =
(λ*x*. *if Ax-BxC.arr x*
then ((*fst x*, *fst* (*snd x*)), *snd* (*snd x*))
else AxB-xC.null)
unfolding *map′-def pointwise-tuple-def BxC.P*$_0$*-def BxC.P*$_1$*-def*
Ax-BxC.P$_0$*-def Ax-BxC.P*$_1$*-def*
by *auto*

lemma *inverse-simulations-map′-map*:
shows *inverse-simulations AxB-xC.resid Ax-BxC.resid map′ map*
proof
show *map* ∘ *map′* = *I Ax-BxC.resid*
unfolding *map-eq map′-eq* **by** *auto*
show *map′* ∘ *map* = *I AxB-xC.resid*
unfolding *map-eq map′-eq* **by** *auto*
qed

end

3.10 Exponential RTS

The exponential [*A*, *B*] of RTS's *A* and *B* has states corresponding to simulations from *A* to *B* and transitions corresponding to transformations between

such simulations. Since our definition of transformation has assumed that A and B are weakly extensional, we need to include those assumptions here. In addition, the definition of residuation for the exponential RTS uses the assumption of uniqueness of joins, so we actually assume that B is extensional. Things become rather inconvenient if this assumption is not made, and I have not investigated whether relaxing it is possible.

```
locale consistent-transformations =
  A: weakly-extensional-rts A +
  B: extensional-rts B +
  F: simulation A B F +
  G: simulation A B G +
  H: simulation A B H +
  σ: transformation A B F G σ +
  τ: transformation A B F H τ
for A :: 'a resid      (infix \<^sub>A 70)
and B :: 'b resid      (infix \<^sub>B 70)
and F :: 'a ⇒ 'b
and G :: 'a ⇒ 'b
and H :: 'a ⇒ 'b
and σ :: 'a ⇒ 'b
and τ :: 'a ⇒ 'b +
assumes con: A.ide a ⟶ B.con (σ a) (τ a)
begin

  sublocale σ: transformation-to-extensional-rts A B F G σ ..
  sublocale τ: transformation-to-extensional-rts A B F H τ ..

  sublocale sym: consistent-transformations A B F H G τ σ
    by (meson B.residuation-axioms con consistent-transformations-axioms
        consistent-transformations-axioms.intro consistent-transformations-def
        residuation.con-sym)
```

The "apex" determined by consistent transformations σ and τ is the simulation whose value at a transition t of A may be visualized as the apex of a rectangular parallelepiped, which is formed with t as its base, the components at *src c* of the transformations associated with the two transitions and their residuals, and the images of t under these transformations.

```
  abbreviation apex :: 'a ⇒ 'b
  where apex ≡ (λt. if A.arr t
                then H t \<^sub>B (σ (A.src t) \<^sub>B τ (A.src t))
                else B.null)

  abbreviation resid :: 'a ⇒ 'b
  where resid ≡ (λt. if A.arr t
                 then B.join (σ (A.src t) \<^sub>B τ (A.src t)) (H t)
                 else B.null)

end
```

For unknown reasons, it is necessary to close and re-open the context here in order to obtain access to *sym* as a sublocale.

```
context consistent-transformations
begin

lemma sym-apex-eq:
shows sym.apex = apex
proof
  fix t
  show sym.apex t = apex t
    by (metis (full-types) σ.naturality2 τ.naturality2 B.cube)
qed
```

The apex associated with two consistent transformations is a simulation. The proof that it preserves residuation can be visualized in terms of a three-dimensional figure consisting of four rectangular parallelepipeds connected into an overall diamond shape, with $Dom\ \tau\ t$ and $Dom\ \sigma\ u$ (which is equal to $Dom\ \tau\ u$) and their residuals at the base of the overall diamond and with $Apex\ \sigma\ \tau\ t$ and $Apex\ \sigma\ \tau\ u$ at its peak.

```
interpretation apex: simulation A B apex
proof
  show ⋀t. ¬ A.arr t ⟹ apex t = B.null
    by simp
  show ⋀t u. t ⌢A u ⟹ apex t ⌢B apex u
  proof −
    fix t u
    assume con: t ⌢A u
    have apex t ⌢B apex u ⟷
         H t \B (σ (A.src t) \B τ (A.src t)) ⌢B
         H u \B (σ (A.src u) \B τ (A.src u))
      using A.con-implies-arr(1−2) con by simp
    also have ... ⟷ (F t \B τ (A.src t)) \B
                        (σ (A.src t) \B τ (A.src t)) ⌢B
                    (F u \B τ (A.src t)) \B
                        (σ (A.src t) \B τ (A.src t))
      using A.con-implies-arr(1−2) τ.naturality2 con A.con-imp-eq-src
      by (metis (mono-tags, lifting))
    also have ... ⟷ (F t \B F u) \B τ (A.trg u) ⌢B
                    (σ (A.src t) \B F u) \B τ (A.trg u)
       using A.con-imp-eq-src τ.naturality1 con B.con-def B.cube by presburger
    also have ... ⟷ (F t \B F u) \B τ (A.trg u) ⌢B
                    σ (A.trg u) \B τ (A.trg u)
      using A.con-imp-eq-src σ.naturality1 con by presburger
    also have ... ⟷ F (t \A u) \B τ (A.trg u) ⌢B
                    σ (A.trg u) \B τ (A.trg u)
      using F.preserves-resid con by presburger
    also have ... ⟷ True
      by (metis A.arr-resid A.ide-trg A.src-resid B.con-def B.con-sym
```

```
        B.cube σ.naturality1 τ.naturality1 con sym.con)
    finally show apex t ⌢B apex u by blast
  qed
  show ⋀t u. t ⌢A u ⟹ apex (t \A u) = apex t \B apex u
  using σ.naturality1 τ.naturality1 τ.naturality2 B.cube A.arr-resid A.arr-src-iff-arr
        A.con-imp-eq-src A.con-implies-arr(1−2) A.src-resid H.preserves-resid
    by auto (metis (mono-tags, lifting))
qed

lemma simulation-apex:
shows simulation A B apex
  ..

lemma resid-ide:
assumes A.ide a
shows resid a = σ a \B τ a
  by (metis (full-types) A.arr-def A.ideE A.weakly-extensional-rts-axioms
      B.join-prfx(2) apex.preserves-ide assms weakly-extensional-rts.src-ide)

interpretation resid: transformation ‹(\A)› ‹(\B)› H apex resid
proof −
  have 2: ⋀f. A.ide f ⟹ B.joinable (σ (A.src f) \B τ (A.src f)) (H f)
    using B.joinable-iff-arr-join con resid-ide by auto
  show transformation (\A) (\B) H apex resid
  proof
    show ⋀f. ¬ A.arr f ⟹ resid f = B.null
      by force
    show 3: ⋀f. A.ide f ⟹ B.src (resid f) = H f
      using 2 B.src-join τ.preserves-trg con by auto
    show ⋀f. A.ide f ⟹ B.trg (resid f) = apex f
      using B.apex-arr-prfx(2) apex.preserves-ide resid-ide by force
    show ⋀f. A.arr f ⟹ H f \B resid (A.src f) = apex f
      using resid-ide by force
    show ⋀f. A.arr f ⟹ resid (A.src f) \B H f = resid (A.trg f)
      using A.ide-src A.ide-trg B.cube σ.naturality1 τ.naturality1
            τ.naturality2 resid-ide
      by auto metis
    show ⋀f. A.arr f ⟹ B.join-of (resid (A.src f)) (H f) (resid f)
    proof −
      fix f
      assume f: A.arr f
      have 1: B.joinable (σ (A.src f) \B τ (A.src f)) (H f)
      proof −
        have (σ (A.src f) ⊔B F f) \B τ (A.src f) =
              (σ (A.src f) \B τ (A.src f)) ⊔B (F f \B τ (A.src f))
          by (metis A.prfx-reflexive A.trg-def B.conI B.con-with-join-if(2)
              B.not-arr-null B.resid-join_E(3) H.preserves-reflects-arr
              σ.naturality1-ax σ.naturality3'_E(2) τ.naturality1-ax τ.naturality2
              f sym.con)
```

```
      hence B.join-of (σ (A.src f) \_B τ (A.src f)) (H f)
                      ((σ (A.src f) \_B τ (A.src f)) ⊔_B (F f \_B τ (A.src f)))
    using f A.ide-trg B.arr-resid-iff-con B.conE B.con-sym B.con-with-join-if(1)
          B.join-def B.join-is-join-of H.preserves-reflects-arr σ.naturality1
          σ.naturality3'_E(2) τ.naturality1 τ.naturality2
      by (metis sym.con)
    thus ?thesis
      using B.joinable-def by blast
  qed
  show B.join-of (resid (A.src f)) (H f) (resid f)
    using 1 A.ide-src B.join-is-join-of resid-ide f by auto
 qed
qed

lemma transformation-resid:
shows transformation (\_A) (\_B) H apex resid
  ..

end
```

Now we can define the exponential $[A,\ B]$ of RTS's A and B.

```
locale exponential-rts =
A: weakly-extensional-rts A +
B: extensional-rts B
for A :: 'a resid      (infix \_A 70)
and B :: 'b resid      (infix \_B 70)
begin

  notation A.con   (infix ⌢_A 50)
  notation A.prfx  (infix ≲_A 50)
  notation B.con   (infix ⌢_B 50)
  notation B.join  (infix ⊔_B 52)
  notation B.prfx  (infix ≲_B 50)

  datatype ('aa, 'bb) arr =
    Null
  | MkArr ‹'aa ⇒ 'bb› ‹'aa ⇒ 'bb› ‹'aa ⇒ 'bb›

  abbreviation MkIde :: ('a ⇒ 'b) ⇒ ('a, 'b) arr
  where MkIde a ≡ MkArr a a a

  fun Dom :: ('a, 'b) arr ⇒ 'a ⇒ 'b
  where Dom (MkArr F - -) = F
      | Dom - = undefined

  fun Cod :: ('a, 'b) arr ⇒ 'a ⇒ 'b
  where Cod (MkArr - G -) = G
      | Cod - = undefined
```

```
fun Map :: ('a, 'b) arr ⇒ 'a ⇒ 'b
where Map (MkArr - - τ) = τ
  | Map - - = undefined

abbreviation Arr :: ('a, 'b) arr ⇒ bool
where Arr ≡ λτ. τ ≠ Null ∧ transformation A B (Dom τ) (Cod τ) (Map τ)

abbreviation Ide :: ('a, 'b) arr ⇒ bool
where Ide ≡ λτ. τ ≠ Null ∧
              identity-transformation A B (Dom τ) (Cod τ) (Map τ)
```

In order to define consistency for transitions of the exponential, we at least need to have pointwise consistency of the components of the corresponding transitions. Surprisingly, this is sufficient.

```
abbreviation Con :: ('a, 'b) arr ⇒ ('a, 'b) arr ⇒ bool
where Con ≡ λσ τ. Arr σ ∧ Arr τ ∧ Dom σ = Dom τ ∧
              (∀ a. A.ide a ⟶ B.con (Map σ a) (Map τ a))

lemma Con-sym:
assumes Con σ τ
shows Con τ σ
  using assms B.con-sym by auto

lemma Con-implies-consistent-transformations:
assumes Con σ τ
shows consistent-transformations
        A B (Dom σ) (Cod σ) (Cod τ) (Map σ) (Map τ)
proof −
  interpret σ: transformation A B ‹Dom σ› ‹Cod σ› ‹Map σ›
    using assms by auto
  interpret τ: transformation A B ‹Dom σ› ‹Cod τ› ‹Map τ›
    using assms by auto
  show ?thesis
    using assms
    apply intro-locales
    by (simp add: consistent-transformations-axioms-def)
qed

definition Apex :: ('a, 'b) arr ⇒ ('a, 'b) arr ⇒ 'a ⇒ 'b
where Apex σ τ = (λt. if A.arr t
                      then Cod τ t \_B (Map σ (A.src t) \_B Map τ (A.src t))
                      else B.null)

lemma Apex-sym:
assumes Con σ τ
shows Apex σ τ = Apex τ σ
  unfolding Apex-def
  using assms Con-implies-consistent-transformations
```

consistent-transformations.sym-apex-eq
[of A B Dom σ Cod σ Cod τ Map σ Map τ]
by *auto*

lemma *Apex-is-simulation* [*intro*]:
assumes *Con σ τ*
shows *simulation A B (Apex σ τ)*
unfolding *Apex-def*
using *assms Con-implies-consistent-transformations*
consistent-transformations.simulation-apex
by *blast*

abbreviation *Resid* :: *('a, 'b) arr ⇒ ('a, 'b) arr ⇒ 'a ⇒ 'b*
where *Resid σ τ ≡ (λt. if A.arr t*
then (Map σ (A.src t)) \B Map τ (A.src t)) ⊔B Cod τ t
else B.null)

definition *resid* :: *('a, 'b) arr resid* (**infix** \ *70*)
where *σ \ τ =*
(if Con σ τ
then MkArr (Cod τ)
(consistent-transformations.apex A B (Cod τ) (Map σ) (Map τ))
(consistent-transformations.resid A B (Cod τ) (Map σ) (Map τ))
else Null)

lemma *Dom-resid'*:
assumes *Con σ τ*
shows *Dom (σ \ τ) = Cod τ*
using *assms resid-def* **by** *auto*

lemma *Cod-resid'*:
assumes *Con σ τ*
shows *Cod (σ \ τ) = Apex σ τ*
unfolding *Apex-def*
using *assms resid-def Con-implies-consistent-transformations* **by** *auto*

lemma *Map-resid'*:
assumes *Con σ τ*
shows *Map (σ \ τ) = (λt. if A.arr t*
then Map σ (A.src t) \B Map τ (A.src t) ⊔B Cod τ t
else B.null)
using *assms resid-def* **by** *auto*

lemma *Map-resid-ide'*:
assumes *Con σ τ* **and** *A.ide a*
shows *Map (σ \ τ) a = Map σ a \B Map τ a*
unfolding *resid-def*
using *assms Con-implies-consistent-transformations*
consistent-transformations.resid-ide

```
      [of A B Dom σ Cod σ Cod τ Map σ Map τ]
  by auto

lemma transformation-Map-resid:
assumes Con σ τ
shows transformation (\_A) (\_B) (Cod τ) (Apex σ τ) (Map (σ \ τ))
  using assms Apex-def resid-def Con-implies-consistent-transformations
      consistent-transformations.transformation-resid
        [of A B Dom σ Cod σ Cod τ Map σ Map τ]
  by auto

sublocale ResiduatedTransitionSystem.partial-magma resid
proof
  show ∃!n. ∀t. n \ t = n ∧ t \ n = n
    using resid-def by metis
qed

lemma null-char:
shows null = Null
  by (metis null-is-zero(2) resid-def)

sublocale residuation resid
proof
  show ⋀σ τ. σ \ τ ≠ null ⟹ τ \ σ ≠ null
    using resid-def null-char Con-sym
    by (metis (no-types, lifting) arr.simps(2))
  show ⋀σ τ. σ \ τ ≠ null ⟹ (σ \ τ) \ (σ \ τ) ≠ null
  proof −
    fix σ τ
    assume 1: σ \ τ ≠ null
    have Con σ τ
      using 1
      by (metis (no-types, opaque-lifting) null-char resid-def)
    hence Con (σ \ τ) (σ \ τ)
      using 1 null-char Dom-resid′ Cod-resid′ transformation-Map-resid
      apply auto[1]
      by (metis A.ideE transformation.preserves-con(1))
    thus (σ \ τ) \ (σ \ τ) ≠ null
      by (simp add: null-char resid-def)
  qed
  show ⋀ϱ σ τ. (σ \ τ) \ (ϱ \ τ) ≠ null
                ⟹ (σ \ τ) \ (ϱ \ τ) = (σ \ ϱ) \ (τ \ ϱ)
  proof −
    fix ϱ σ τ
    assume στ-ϱσ: (σ \ τ) \ (ϱ \ τ) ≠ null
    have στ: Con σ τ
      using στ-ϱσ resid-def
      by (metis (no-types, opaque-lifting) null-char)
    have ϱτ: Con ϱ τ
```

using $\sigma\tau$-$\varrho\sigma$ *resid-def*
by (*metis* (*no-types*, *opaque-lifting*) *null-char*)
have $\sigma\tau$-$\varrho\tau$: *Con* $(\sigma \setminus \tau)$ $(\varrho \setminus \tau)$
using $\sigma\tau$-$\varrho\sigma$ *resid-def*
by (*metis* (*no-types*, *opaque-lifting*) *null-char*)
interpret σ: *transformation A B* ‹*Dom* σ› ‹*Cod* σ› ‹*Map* σ›
using $\sigma\tau$ **by** *blast*
interpret τ: *transformation A B* ‹*Dom* σ› ‹*Cod* τ› ‹*Map* τ›
using $\sigma\tau$ **by** *auto*
interpret ϱ: *transformation A B* ‹*Dom* σ› ‹*Cod* ϱ› ‹*Map* ϱ›
using $\sigma\tau$ $\varrho\tau$ **by** *auto*
have *1*: $\bigwedge a$. *A.ide a* $\Longrightarrow$ *Map* σ *a* $\setminus_B$ *Map* τ *a* $\frown_B$ *Map* ϱ *a* $\setminus_B$ *Map* τ *a*
using $\sigma\tau$ $\varrho\tau$ *Map-resid-ide'* $\sigma\tau$-$\varrho\tau$ **by** *force*
show $(\sigma \setminus \tau) \setminus (\varrho \setminus \tau) = (\sigma \setminus \varrho) \setminus (\tau \setminus \varrho)$
proof −
interpret $\sigma\tau$: *transformation A B* ‹*Cod* τ› ‹*Apex* σ τ› ‹*Map* $(\sigma \setminus \tau)$›
using $\sigma\tau$-$\varrho\sigma$ $\sigma\tau$ $\varrho\tau$ *resid-def transformation-Map-resid* [*of* σ τ] **by** *blast*
interpret $\varrho\tau$: *transformation A B* ‹*Cod* τ› ‹*Apex* ϱ τ› ‹*Map* $(\varrho \setminus \tau)$›
using $\sigma\tau$-$\varrho\sigma$ $\sigma\tau$ $\varrho\tau$ *resid-def transformation-Map-resid* [*of* ϱ τ] **by** *blast*
have $\sigma\varrho$: *Con* σ ϱ
by (*metis* $\sigma\tau$ $\varrho\tau$ *1 B.resid-reflects-con*)
have $\tau\varrho$: *Con* τ ϱ
by (*metis* $\varrho\tau$ *B.con-sym*)
interpret $\sigma\varrho$: *transformation A B* ‹*Cod* ϱ› ‹*Apex* σ ϱ› ‹*Map* $(\sigma \setminus \varrho)$›
using $\sigma\varrho$ *transformation-Map-resid* **by** *blast*
interpret $\tau\varrho$: *transformation A B* ‹*Cod* ϱ› ‹*Apex* τ ϱ› ‹*Map* $(\tau \setminus \varrho)$›
using $\tau\varrho$ *transformation-Map-resid* **by** *blast*
have $\sigma\varrho$-$\tau\varrho$: *Con* $(\sigma \setminus \varrho)$ $(\tau \setminus \varrho)$
proof (*intro conjI*)
show $\sigma \setminus \varrho \neq$ *Null*
using $\sigma\varrho$ *resid-def* **by** *force*
show $\tau \setminus \varrho \neq$ *Null*
by (*metis* $\sigma\tau$-$\varrho\sigma$ ‹$\bigwedge\tau$ σ. $\sigma \setminus \tau \neq$ *null* $\Longrightarrow$ $\tau \setminus \sigma \neq$ *null*› *null-char*
null-is-zero(*2*))
show *Dom* $(\sigma \setminus \varrho)$ = *Dom* $(\tau \setminus \varrho)$
using $\sigma\varrho$ $\tau\varrho$ *exponential-rts.resid-def exponential-rts-axioms* **by** *force*
show *transformation A B* (*Dom* $(\sigma \setminus \varrho)$) (*Cod* $(\sigma \setminus \varrho)$) (*Map* $(\sigma \setminus \varrho)$)
using $\sigma\varrho$ $\sigma\varrho$.*transformation-axioms Dom-resid' Cod-resid'* **by** *auto*
show *transformation A B* (*Dom* $(\tau \setminus \varrho)$) (*Cod* $(\tau \setminus \varrho)$) (*Map* $(\tau \setminus \varrho)$)
using $\tau\varrho$ $\tau\varrho$.*transformation-axioms Dom-resid' Cod-resid'* **by** *auto*
show $\forall\, a$. *A.ide a* $\longrightarrow$ *Map* $(\sigma \setminus \varrho)$ *a* $\frown_B$ *Map* $(\tau \setminus \varrho)$ *a*
using *1 B.con-def B.cube Map-resid-ide'* $\sigma\varrho$ $\tau\varrho$ **by** *force*
qed
interpret $\sigma\tau$-$\varrho\tau$: *transformation A B* ‹*Apex* ϱ τ› ‹*Apex* $(\sigma \setminus \tau)$ $(\varrho \setminus \tau)$›
‹*Map* $((\sigma \setminus \tau) \setminus (\varrho \setminus \tau))$›
by (*metis* $\varrho\tau$ *Cod-resid'* $\sigma\tau$-$\varrho\tau$ *transformation-Map-resid*)
interpret $\sigma\varrho$-$\tau\varrho$: *transformation A B* ‹*Apex* τ ϱ› ‹*Apex* $(\sigma \setminus \varrho)$ $(\tau \setminus \varrho)$›
‹*Map* $((\sigma \setminus \varrho) \setminus (\tau \setminus \varrho))$›
using $\sigma\varrho$-$\tau\varrho$ *transformation-Map-resid*

```
  by (metis Cod-resid' τ ϱ)
show (σ \ τ) \ (ϱ \ τ) = (σ \ ϱ) \ (τ \ ϱ)
proof −
  have 2: ⋀a. A.ide a ⟹
              Map ((σ \ τ) \ (ϱ \ τ)) a = Map ((σ \ ϱ) \ (τ \ ϱ)) a
    using B.cube Map-resid-ide' ϱτ σϱ σϱ-τϱ στ στ-ϱτ τϱ by auto
  have 3: Apex (σ \ ϱ) (τ \ ϱ) = Apex (σ \ τ) (ϱ \ τ)
  proof −
    have (λt. if A.arr t
              then Cod (τ \ ϱ) t \B
                   (Map (σ \ ϱ) (A.src t) \B Map (τ \ ϱ) (A.src t))
              else B.null) =
         (λt. if A.arr t
              then Cod (ϱ \ τ) t \B
                   (Map (σ \ τ) (A.src t) \B Map (ϱ \ τ) (A.src t))
              else B.null)
    proof
      fix t
      show (if A.arr t
              then Cod (τ \ ϱ) t \B
                   (Map (σ \ ϱ) (A.src t) \B Map (τ \ ϱ) (A.src t))
              else B.null) =
           (if A.arr t
              then Cod (ϱ \ τ) t \B
                   (Map (σ \ τ) (A.src t) \B Map (ϱ \ τ) (A.src t))
              else B.null)
      proof (cases A.arr t)
        show ¬ A.arr t ⟹ ?thesis
          by auto
        assume t: A.arr t
        show ?thesis
          by (metis 2 A.ide-src Apex-def Apex-sym σϱ-τϱ.naturality2
              στ-ϱτ.naturality2 τϱ)
      qed
    qed
    thus ?thesis
      using Apex-def by simp
  qed
  have ⋀x y. ⟦x ≠ null; y ≠ null;
              Dom x = Dom y; Cod x = Cod y; Map x = Map y⟧
                ⟹ x = y
    by (metis Cod.elims Dom.simps(1) Map.simps(1) null-char)
  moreover have (σ \ ϱ) \ (τ \ ϱ) ≠ null
    using στ-ϱτ exponential-rts.resid-def exponential-rts-axioms null-char
    by force
  moreover have Dom ((σ \ τ) \ (ϱ \ τ)) = Dom ((σ \ ϱ) \ (τ \ ϱ))
    using ϱτ Apex-sym Cod-resid' Dom-resid' σϱ-τϱ στ-ϱτ τϱ by auto
  moreover have Cod ((σ \ τ) \ (ϱ \ τ)) = Cod ((σ \ ϱ) \ (τ \ ϱ))
    using Cod-resid' σϱ-τϱ στ-ϱτ 3 by auto
```

```
      moreover have Map ((σ \ τ) \ (ϱ \ τ)) = Map ((σ \ ϱ) \ (τ \ ϱ))
      using ϱτ 2 3 στ-ϱτ.transformation-axioms στ-ϱτ.transformation-axioms
            transformation-eqI B.extensional-rts-axioms Apex-sym
        by metis
      ultimately show ?thesis
        using στ-ϱσ null-char by blast
    qed
  qed
 qed
qed

notation con  (infix ⌢ 50)

lemma con-char:
shows σ ⌢ τ ⟷ Con σ τ
  using con-def resid-def null-char by auto

lemma arr-char:
shows arr τ ⟷ Arr τ
  by (metis A.ide-implies-arr B.arr-def arrE arrI con-char
      transformation.preserves-arr)

lemma Dom-resid [simp]:
assumes σ ⌢ τ
shows Dom (σ \ τ) = Cod τ
  using assms Dom-resid′ con-char by auto

lemma Cod-resid [simp]:
assumes σ ⌢ τ
shows Cod (σ \ τ) = Apex σ τ
  using assms Cod-resid′ con-char by auto

lemma Map-resid [simp]:
assumes σ ⌢ τ
shows Map (σ \ τ) = (λt. if A.arr t
                      then Map σ (A.src t) \B Map τ (A.src t) ⊔B Cod τ t
                      else B.null)
  using assms Map-resid′ con-char by auto

lemma Map-resid-ide [simp]:
assumes con σ τ and A.ide a
shows Map (σ \ τ) a = Map σ a \B Map τ a
  using Map-resid-ide′ assms(1−2) con-char by blast

lemma resid-Map:
assumes con ϱ σ and t ⌢A u
shows Map ϱ t \B Map σ u = Map (ϱ \ σ) (t \A u)
proof −
  interpret ϱ: transformation A B ‹Dom ϱ› ‹Cod ϱ› ‹Map ϱ›
```

```
  using assms(1) arr-char con-implies-arr(1) by blast
interpret ϱ: transformation-to-extensional-rts A B ‹Dom ϱ› ‹Cod ϱ› ‹Map ϱ›
  ..
interpret σ: transformation A B ‹Dom σ› ‹Cod σ› ‹Map σ›
  using assms(1) arr-char con-implies-arr(2) by blast
interpret σ: transformation-to-extensional-rts A B
              ‹Dom σ› ‹Cod σ› ‹Map σ›
  ..
interpret ϱσ: transformation A B ‹Cod σ› ‹Apex ϱ σ› ‹Map (ϱ \ σ)›
  using assms con-char transformation-Map-resid by auto
interpret ϱσ: transformation-to-extensional-rts A B
               ‹Cod σ› ‹Apex ϱ σ› ‹Map (ϱ \ σ)›
  ..
have Map ϱ t \B Map σ u = Map ϱ t \B (Map σ (A.src u) ⊔B Dom σ u)
  by (metis σ.naturality3'E(1))
also have ... =
          (Map ϱ t \B Map σ (A.src u)) \B (Dom σ u \B Map σ (A.src u))
  using B.resid-joinE(2) [of Map σ (A.src u) Dom σ u Map ϱ t]
  by (metis A.con-implies-arr(2) B.conI B.con-sym-ax B.con-with-join-if(2)
      B.join-sym B.joinable-iff-join-not-null B.null-is-zero(2) σ.naturality3'E(2)
      assms(2))
also have ... = (Map ϱ t \B Map σ (A.src u)) \B Cod σ u
  using σ.naturality2 by presburger
also have ... =
          ((Map ϱ (A.src t) ⊔B Dom ϱ t) \B Map σ (A.src u)) \B Cod σ u
  by (metis ϱ.naturality3'E(1))
also have ... =
          (((Map ϱ (A.src t) \B Map σ (A.src u)) ⊔B
            (Dom ϱ t \B Map σ (A.src u))))
            \B Cod σ u
  using B.resid-joinE(3)
by (metis (mono-tags, lifting) A.con-imp-eq-src A.ide-trg A.residuation-axioms
      B.con-sym B.con-with-join-if(2) B.joinable-implies-con ϱ.naturality1
    ϱ.naturality3'E(2) σ.naturality1 σ.naturality3'E(2) assms(1−2) con-char
      residuation.con-implies-arr(1))
also have ... = (Map (ϱ \ σ) (A.src t) ⊔B Cod σ t) \B Cod σ u
by (metis (mono-tags, lifting) A.con-imp-eq-src A.con-implies-arr(1) A.ide-src
      Map-resid-ide σ.naturality2 assms(1−2) con-char)
also have ... = Map (ϱ \ σ) (A.src t) \B Cod σ u ⊔B Cod σ t \B Cod σ u
by (metis (full-types) assms(2) A.con-implies-arr(1) B.conE B.conI B.con-sym-ax
      B.resid-joinE(3) ϱσ.naturality3'E(1−2) ϱσ.preserves-con(2))
also have ... = Map (ϱ \ σ) (A.src t) \B Cod σ u ⊔B Cod σ (t \A u)
  using assms(2) by fastforce
also have ... = Map (ϱ \ σ) (A.trg u) ⊔B Cod σ (t \A u)
  using A.con-imp-eq-src ϱσ.naturality1 assms(2) by presburger
also have ... = Map (ϱ \ σ) (t \A u)
  by (metis A.src-resid ϱσ.naturality3'E(1) assms(2))
finally show ?thesis by simp
qed
```

```
lemma resid-def':
shows σ \ τ =
      (if σ ⌢ τ
       then MkArr (Cod τ) (Apex σ τ)
                  (λt. if A.arr t
                       then Map σ (A.src t) \_B Map τ (A.src t) ⊔_B Cod τ t
                       else B.null)
       else null)
  using Apex-def resid-def con-char null-char Dom-resid Cod-resid Map-resid
  by auto

lemma trg-simp:
assumes arr τ
shows trg τ = MkArr (Cod τ) (Cod τ) (Cod τ)
proof −
  interpret τ: transformation A B ‹Dom τ› ‹Cod τ› ‹Map τ›
    using assms arr-char by blast
  have trg τ = τ \ τ
    using assms trg-def by blast
  also have ... = MkArr (Cod τ) (Cod τ) (Cod τ)
  proof −
    have Dom (τ \ τ) = Cod τ
      using Dom-resid assms by blast
    moreover have Cod (τ \ τ) = Cod τ
    proof −
      have (λt. if A.arr t
                then Cod τ t \_B (Map τ (A.src t) \_B Map τ (A.src t))
                else B.null) =
           Cod τ
      proof
        fix t
        show (if A.arr t
              then Cod τ t \_B (Map τ (A.src t) \_B Map τ (A.src t))
              else B.null) =
             Cod τ t
          by (metis A.con-arr-src(1) A.ide-src A.resid-arr-src B.trg-def
            τ.G.extensional τ.G.preserves-resid τ.preserves-trg)
      qed
      thus ?thesis
        using resid-def Apex-def τ.transformation-axioms assms
              arr-char null-char τ.preserves-arr
        by auto
    qed
    moreover have Map (τ \ τ) = Cod τ
    proof −
      have (λt. if A.arr t
                then Map τ (A.src t) \_B Map τ (A.src t) ⊔_B Cod τ t
                else B.null) =
```

```
        Cod τ
    proof
      fix t
      show (if A.arr t
            then Map τ (A.src t) \_B Map τ (A.src t) ⊔_B Cod τ t
            else B.null) =
           Cod τ t
        by (metis B.arr-resid-iff-con B.join-src B.src-resid B.trg-def
            τ.G.extensional τ.G.preserves-reflects-arr τ.naturality2)
    qed
    thus ?thesis
      using resid-def τ.transformation-axioms assms arr-char
            A.ide-implies-arr B.residuation-axioms τ.preserves-arr
            residuation.arrE
      by auto
  qed
  ultimately show ?thesis
    using assms arr-char
    by (metis Cod.simps(1) Dom.simps(1) Map.simps(1) arr.exhaust arrE
        conE null-char)
qed
finally show ?thesis by blast
qed

lemma trg-char:
shows trg = (λτ. if arr τ then MkIde (Cod τ) else null)
  using trg-simp trg-def by fastforce

lemma Map-trg [simp]:
assumes arr τ
shows Map (trg τ) = Cod τ
  using assms trg-simp by auto

lemma resid-Map-self:
assumes arr σ and t ⌢_A u
shows Map σ t \_B Map σ u = Cod σ (t \_A u)
  using assms resid-Map [of σ σ t u]
  by (metis Map-trg arrE trg-def)

lemma ide-char_ERTS [iff]:
shows ide τ ⟷ τ ≠ null ∧ simulation A B (Map τ) ∧
                Dom τ = Map τ ∧ Cod τ = Map τ
proof
  show ide τ ⟹ τ ≠ null ∧ simulation A B (Map τ) ∧
                  Dom τ = Map τ ∧ Cod τ = Map τ
    by (metis Dom-resid Map.simps(1) arr-char ideE ide-implies-arr
        null-char transformation-def trg-simp trg-def)
  show τ ≠ null ∧ simulation A B (Map τ) ∧
        Dom τ = Map τ ∧ Cod τ = Map τ
```

```
        ⟹ ide τ
  proof −
    assume 1: τ ≠ null ∧ simulation A B (Map τ) ∧
              Dom τ = Map τ ∧ Cod τ = Map τ
    interpret τ: simulation A B ‹Map τ›
      using 1 by blast
    interpret τ: simulation-as-transformation A B ‹Map τ› ..
    show ide τ
      by (metis 1 Cod.elims Dom.simps(1) Map.simps(1)
          τ.transformation-axioms arr-char arr-def ide-def null-char
          trg-simp trg-def)
  qed
qed

sublocale rts resid
proof
  show ⋀t. arr t ⟹ ide (trg t)
    using arr-char ide-char_ERTS trg-simp
    by (metis (full-types) Cod.simps(1) arr.simps(2) con-def con-imp-arr-resid
      con-implies-arr(2) ide-def null-char trg-def)
  show ⋀a t. ⟦ide a; t ⌢ a⟧ ⟹ t \ a = t
  proof −
    fix a t
    assume a: ide a
    assume con: t ⌢ a
    interpret a: identity-transformation A B ‹Dom a› ‹Cod a› ‹Map a›
      using a ide-char_ERTS
      by (metis con con-char identity-transformation-axioms-def
          identity-transformation-def simulation.preserves-ide)
    interpret t: transformation A B ‹Dom a› ‹Cod t› ‹Map t›
      using con con-char by metis
    have t \ a = MkArr (Cod a) (Apex t a)
                       (λta. if A.arr ta
                             then (Map t (A.src ta) \_B Map a (A.src ta)) ⊔_B
                                  Cod a ta
                             else B.null)
      using con ide-char_ERTS con-char resid-def Apex-def
      by simp metis
    moreover have Apex t a = Cod t
    proof −
      have (λu. if A.arr u
                then Cod a u \_B (Map t (A.src u) \_B Map a (A.src u))
                else B.null) =
           Cod t
      proof
        fix u
        show (if A.arr u
              then Cod a u \_B (Map t (A.src u) \_B Map a (A.src u))
              else B.null) =
```

```
        Cod t u
      by (metis A.src-src A.trg-src a a.src-eq-trg ide-char_ERTS
          t.G.extensional t.naturality1 t.naturality2)
    qed
    thus ?thesis
      using a con ide-char_ERTS con-char Apex-def by simp
  qed
  moreover have (λu. if A.arr u
                     then Map t (A.src u) \_B Map a (A.src u) ⊔_B Cod a u
                     else B.null) =
                Map t
  proof
    fix u
    show (if A.arr u
          then Map t (A.src u) \_B Map a (A.src u) ⊔_B Cod a u
          else B.null) =
         Map t u
      by (metis (full-types) A.arr-src-iff-arr A.ide-src A.src-src
          B.join-is-join-of B.join-of-unique B.joinable-def B.resid-arr-src
          a ide-char_ERTS t.naturality3 t.preserves-arr t.preserves-src
          t.transformation-axioms transformation.extensional)
  qed
  ultimately show t \ a = t
    using con con-char
    by (metis Cod.elims Dom.simps(1) Map.simps(1) a.src-eq-trg)
qed
thus ⋀a t. ⟦ide a; a ⌢ t⟧ ⟹ ide (a \ t)
  by (metis arrE arr-resid con-sym cube ideE ideI)
show ⋀t u. t ⌢ u ⟹ ∃a. ide a ∧ a ⌢ t ∧ a ⌢ u
proof −
  fix t u
  assume con: t ⌢ u
  interpret t: transformation A B ‹Dom t› ‹Cod t› ‹Map t›
    using con con-char by blast
  interpret u: transformation A B ‹Dom t› ‹Cod u› ‹Map u›
    using con con-char by auto
  interpret Dom-t: transformation A B ‹Dom t› ‹Dom t› ‹Dom t›
    by (metis Cod.simps(1) Dom.simps(1) Map.simps(1) arr.simps(2)
        arr-char ide-char_ERTS ide-implies-arr null-char
        t.F.simulation-axioms)
  interpret Dom-t: identity-transformation A B ‹Dom t› ‹Dom t› ‹Dom t›
    by unfold-locales auto
  have ide (MkIde (Dom t))
    by (simp add: null-char t.F.simulation-axioms)
  moreover have MkIde (Dom t) ⌢ t
    using con con-char Dom-t.transformation-axioms Dom-t.identity
          t.transformation-axioms
    by simp
       (metis A.ide-iff-src-self A.ide-implies-arr B.not-ide-null
```

```
        B.conI t.G.preserves-ide t.naturality2)
    moreover have MkIde (Dom t) ⌢ u
      using con con-char Dom-t.transformation-axioms Dom-t.identity
            u.transformation-axioms
      by simp
         (metis A.ide-iff-src-self A.ide-implies-arr B.not-ide-null
          B.conI u.G.preserves-ide u.naturality2)
    ultimately show ∃a. ide a ∧ a ⌢ t ∧ a ⌢ u by blast
  qed
  show ⋀t u v. ⟦ide (t \ u); u ⌢ v⟧ ⟹ t \ u ⌢ v \ u
    by (metis (no-types, opaque-lifting) Dom-resid conI con-implies-arr(1)
        cube ideE resid-arr-self conE con-imp-arr-resid trg-simp)
qed

lemma is-rts:
shows rts resid
  ..

sublocale extensional-rts resid
proof
  fix t u
  assume cong: cong t u
  interpret t: transformation A B ‹Dom t› ‹Cod t› ‹Map t›
    using cong con-char prfx-implies-con by blast
  interpret u: transformation A B ‹Dom t› ‹Cod u› ‹Map u›
    using cong con-char prfx-implies-con by metis
  interpret tu: transformation A B ‹Cod u› ‹Apex t u› ‹Map (t \ u)›
    using cong transformation-Map-resid
    by (metis con-char prfx-implies-con)
  interpret tu: identity-transformation A B ‹Cod u› ‹Apex t u› ‹Map (t \ u)›
    by unfold-locales (meson ide-char_ERTS cong simulation.preserves-ide)
  interpret ut: transformation A B ‹Cod t› ‹Apex t u› ‹Map (u \ t)›
    using cong transformation-Map-resid Apex-sym
    by (metis con-char prfx-implies-con)
  interpret ut: identity-transformation A B ‹Cod t› ‹Apex t u› ‹Map (u \ t)›
    by unfold-locales (meson ide-char_ERTS cong simulation.preserves-ide)
  have 1: ⋀a. A.ide a ⟹ Map t a = Map u a
    by (metis B.cong-char Map-resid-ide congE cong tu.identity ut.identity)
  show t = u
    using t.transformation-axioms u.transformation-axioms
          transformation-eqI
    by (metis (no-types, lifting) 1 B.extensional-rts-axioms Cod.elims
        Dom.simps(1) Map.simps(1) resid-def cong not-ide-null null-char
        tu.src-eq-trg ut.src-eq-trg)
qed

lemma is-extensional-rts:
shows extensional-rts resid
  ..
```

lemma *conI*$_{ERTS}$ [*intro*]:
assumes *coinitial* σ τ
and $\bigwedge a.$ *A.ide a* $\Longrightarrow$ *Map* σ *a* $\frown_B$ *Map* τ *a*
shows $\sigma \frown \tau$
using *assms con-char*
by (*metis* (*full-types*) *coinitialE*$_{WE}$ *con-arr-src*(*1*))

lemma *conE*$_{ERTS}$ [*elim*]:
assumes $\sigma \frown \tau$
and $[\![$*coinitial* σ τ; $\bigwedge t$ *u. A.con t u* $\Longrightarrow$ *Map* σ *t* $\frown_B$ *Map* τ *u*$]\!] \Longrightarrow T$
shows *T*
by (*metis A.arr-resid-iff-con B.arr-resid-iff-con arr-char*
arr-resid-iff-con assms(*1*) *assms*(*2*) *con-imp-coinitial resid-Map*
transformation.preserves-arr)

lemma *arrI* [*intro*]:
assumes $f \neq$ *null* **and** *transformation A B* (*Dom f*) (*Cod f*) (*Map f*)
shows *arr f*
using *assms arr-char null-char* **by** *simp*

lemma *arrE* [*elim*]:
assumes *arr f*
and $[\![f \neq$ *null*; *transformation A B* (*Dom f*) (*Cod f*) (*Map f*)$]\!] \Longrightarrow T$
shows *T*
using *assms arr-char null-char* **by** *simp*

lemma *arr-MkArr* [*iff*]:
shows *arr* (*MkArr F G* τ) $\longleftrightarrow$ *transformation A B F G* τ
using *arr-char null-char transformation-def* **by** *fastforce*

lemma *src-simp*:
assumes *arr* τ
shows *src* τ = *MkIde* (*Dom* τ)
by (*metis Dom-resid arr-src-iff-arr assms con-arr-src*(*1*) *resid-arr-src*
trg-simp trg-src)

lemma *src-char*:
shows *src* = ($\lambda\tau$. *if arr* τ *then MkIde* (*Dom* τ) *else null*)
using *src-simp src-def* **by** *auto*

lemma *Map-src* [*simp*]:
assumes *arr* τ
shows *Map* (*src* τ) = *Dom* τ
using *assms src-simp* **by** *auto*

lemma *ide-MkIde* [*iff*]:
shows *ide* (*MkIde F*) $\longleftrightarrow$ *simulation A B F*
using *ide-char*$_{ERTS}$ *null-char* **by** *auto*

lemma *MkArr-Map*:
assumes $\tau \neq Null$
shows $\tau = MkArr\ (Dom\ \tau)\ (Cod\ \tau)\ (Map\ \tau)$
using *assms* **by** (*cases* τ) *auto*

lemma *MkIde-Dom*:
assumes *arr* τ
shows $MkIde\ (Dom\ \tau) = src\ \tau$
using *assms arr-char src-char* **by** *auto*

lemma *MkIde-Cod*:
assumes *arr* τ
shows $MkIde\ (Cod\ \tau) = trg\ \tau$
using *assms arr-char trg-char* **by** *auto*

lemma *MkIde-Map*:
assumes *ide a*
shows $MkIde\ (Map\ a) = a$
using *assms trg-char trg-ide* **by** *auto*

lemma *arr-eqI*:
assumes *arr* σ **and** *arr* τ **and** $Dom\ \sigma = Dom\ \tau$ **and** $Cod\ \sigma = Cod\ \tau$
and $\bigwedge a.\ A.ide\ a \Longrightarrow Map\ \sigma\ a = Map\ \tau\ a$
shows $\sigma = \tau$
proof −
interpret σ: *transformation A B* ‹*Dom* σ› ‹*Cod* σ› ‹*Map* σ›
using *assms*(*1*) **by** *blast*
interpret σ: *transformation-to-extensional-rts A B*
‹*Dom* σ› ‹*Cod* σ› ‹*Map* σ› **..**
interpret τ: *transformation A B* ‹*Dom* τ› ‹*Cod* τ› ‹*Map* τ›
using *assms*(*2*) **by** *blast*
interpret τ: *transformation-to-extensional-rts A B*
‹*Dom* τ› ‹*Cod* τ› ‹*Map* τ› **..**
have $Map\ \sigma = Map\ \tau$
by (*metis assms*(*3*−*5*) *B.extensional-rts-axioms* σ*.transformation-axioms*
τ*.transformation-axioms transformation-eqI*)
thus *?thesis*
using *assms*
by (*metis MkArr-Map arr-char*)
qed

lemma *seq-char*:
shows $seq\ \sigma\ \tau \longleftrightarrow Arr\ \sigma \wedge Arr\ \tau \wedge Cod\ \sigma = Dom\ \tau$
using *arr-char src-char trg-char*
by (*metis* (*no-types*, *lifting*) *Map-src Map-trg* $seqE_{WE}$ $seqI_{WE}$(*1*))

notation *prfx* (**infix** $\lesssim$ *50*)

lemma *prfx-char*:
shows $\sigma \lesssim \tau \longleftrightarrow arr\ \sigma \wedge arr\ \tau \wedge Dom\ \sigma = Dom\ \tau \wedge$
$(\forall a.\ A.ide\ a \longrightarrow Map\ \sigma\ a \lesssim_B Map\ \tau\ a)$
proof
show $\sigma \lesssim \tau \Longrightarrow arr\ \sigma \wedge arr\ \tau \wedge Dom\ \sigma = Dom\ \tau \wedge$
$(\forall a.\ A.ide\ a \longrightarrow Map\ \sigma\ a \lesssim_B Map\ \tau\ a)$
by (*metis Map-resid-ide con-char con-implies-arr*(1−2) *ide-char*$_{ERTS}$
prfx-implies-con simulation.preserves-ide)
show $arr\ \sigma \wedge arr\ \tau \wedge Dom\ \sigma = Dom\ \tau \wedge$
$(\forall a.\ A.ide\ a \longrightarrow Map\ \sigma\ a \lesssim_B Map\ \tau\ a) \Longrightarrow \sigma \lesssim \tau$
proof −
assume *1*: $arr\ \sigma \wedge arr\ \tau \wedge Dom\ \sigma = Dom\ \tau \wedge$
$(\forall a.\ A.ide\ a \longrightarrow Map\ \sigma\ a \lesssim_B Map\ \tau\ a)$
have *2*: $\sigma \frown \tau$
using *1 con-char arr-char B.prfx-implies-con* **by** *force*
interpret $\sigma\tau$: *transformation A B* ‹*Cod* τ› ‹*Apex* $\sigma\ \tau$› ‹*Map* $(\sigma \setminus \tau)$›
using *2 con-char transformation-Map-resid* **by** *auto*
interpret $\sigma\tau$: *simulation A B* ‹*Map* $(\sigma \setminus \tau)$›
proof
show $\bigwedge t.\ \neg\ A.arr\ t \Longrightarrow Map\ (\sigma \setminus \tau)\ t = B.null$
using $\sigma\tau$*.extensional* **by** *blast*
show *3*: $\bigwedge t\ u.\ t \frown_A u \Longrightarrow Map\ (\sigma \setminus \tau)\ t \frown_B Map\ (\sigma \setminus \tau)\ u$
using $\sigma\tau$*.preserves-con* **by** *blast*
show $\bigwedge t\ u.\ t \frown_A u \Longrightarrow$
$Map\ (\sigma \setminus \tau)\ (t \setminus_A u) = Map\ (\sigma \setminus \tau)\ t \setminus_B Map\ (\sigma \setminus \tau)\ u$
by (*metis 1 2 3 A.arr-resid A.con-arr-src*(1) *A.ide-src A.resid-arr-src*
B.resid-arr-ide arr-resid-iff-con Map-resid-ide resid-Map-self)
qed
show $\sigma \lesssim \tau$
proof
show $\sigma \setminus \tau \neq null \wedge simulation\ A\ B\ (Map\ (\sigma \setminus \tau)) \wedge$
$Dom\ (\sigma \setminus \tau) = Map\ (\sigma \setminus \tau) \wedge Cod\ (\sigma \setminus \tau) = Map\ (\sigma \setminus \tau)$
proof (*intro conjI*)
show $\sigma \setminus \tau \neq null$
using *2* **by** *blast*
show *simulation A B* $(Map\ (\sigma \setminus \tau))$
..
show *3*: $Dom\ (\sigma \setminus \tau) = Map\ (\sigma \setminus \tau)$
proof −
have $Dom\ (\sigma \setminus \tau) = Cod\ \tau$
using *2 con-char* **by** *auto*
also have ... $= Map\ (\sigma \setminus \tau)$
proof −
have $\bigwedge t.\ A.arr\ t \Longrightarrow$
$Map\ \sigma\ (A.src\ t) \setminus_B Map\ \tau\ (A.src\ t) = B.src\ (Cod\ \tau\ t)$
by (*metis 1 2 A.ide-src B.src-ide B.src-join-of*(1−2) *Map-resid-ide*
$\sigma\tau$*.naturality3*)
hence $\bigwedge t.\ Map\ (\sigma \setminus \tau)\ t = Cod\ \tau\ t$

```
            using 1 2 B.join-src στ.F.extensional con-char by auto
          thus ?thesis by auto
        qed
        finally show ?thesis by blast
      qed
      show Cod (σ \ τ) = Map (σ \ τ)
      proof −
        have Cod (σ \ τ) = Apex σ τ
          using 2 con-char by auto
        also have ... = Map (σ \ τ)
        proof
          fix t
          show Apex σ τ t = Map (σ \ τ) t
            using 1 Map-resid [of σ τ]
            by (metis (no-types, lifting) 2 3 A.con-arr-src(1) A.resid-arr-src
                Apex-def Dom-resid στ.naturality2 στ.preserves-resid)
        qed
        finally show ?thesis by blast
      qed
    qed
  qed
qed

lemma Map-preserves-prfx:
assumes σ ≲ τ and A.arr t
shows Map σ t ≲B Map τ t
  by (metis A.rts-axioms Map-resid-ide assms(1−2) prfx-char resid-Map
      rts.cong-reflexive rts.prfx-implies-con rts-axioms)
```

Joins in an Exponential RTS

```
notation join  (infix ⊔ 52)

lemma join-char:
shows joinable σ τ ⟷
      arr σ ∧ arr τ ∧ Dom σ = Dom τ ∧
      (∀ t. A.arr t ⟶
            B.joinable (Map σ (A.src t) ⊔B Map τ (A.src t)) (Dom σ t))
and σ ⊔ τ =
    (if joinable σ τ
     then MkArr (Dom τ) (Apex σ τ)
               (λt. (Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom τ t)
     else null)
proof (intro iffI)
  have ∗: joinable σ τ ⟹
          arr σ ∧ arr τ ∧ Dom σ = Dom τ ∧ σ ⊔ τ ≠ Null ∧
          Dom (σ ⊔ τ) = Dom σ ∧ Cod (σ ⊔ τ) = Apex σ τ ∧
          Map (σ ⊔ τ) =
```

```
        (λt. (Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom τ t) ∧
        transformation A B (Dom σ) (Apex σ τ) (Map (join σ τ)) ∧
        (∀a. A.ide a ⟶ B.joinable (Map σ a ⊔B Map τ a) (Dom τ a)) ∧
        (∀t. (Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom τ t =
             Map (σ ⊔ τ) t)
proof (intro conjI)
  assume 1: joinable σ τ
  show σ: arr σ
    using 1 con-implies-arr(1) joinable-implies-con by blast
  show τ: arr τ
    using 1 con-implies-arr(2) joinable-implies-con by blast
  show Dom σ = Dom τ
    using 1 con-char joinable-implies-con by presburger
  have 2: join-of σ τ (σ ⊔ τ)
    using 1 join-is-join-of by simp
  show σ ⊔ τ ≠ Null
    using 1 joinable-iff-join-not-null null-char by force
  show Dom: Dom (σ ⊔ τ) = Dom σ
  by (metis (no-types, opaque-lifting) 1 arr-prfx-join-self not-ide-null null-char
      resid-def)
  show Cod: Cod (σ ⊔ τ) = Apex σ τ
    using 1 trg-simp Cod-resid trg-join joinable-implies-con arr-resid
    by (metis arr.inject joinable-iff-arr-join)
  interpret στ: transformation A B ‹Dom σ› ‹Apex σ τ› ‹Map (σ ⊔ τ)›
    using Dom Cod 1 joinable-implies-con con-char arr-char
    by (metis joinable-iff-arr-join)
  interpret στ: transformation-to-extensional-rts A B
                ‹Dom σ› ‹Apex σ τ› ‹Map (σ ⊔ τ)› ..
  show transformation A B (Dom σ) (Apex σ τ) (Map (σ ⊔ τ)) ..
  have 3: ⋀a. A.ide a ⟹ Map σ a ⊔B Map τ a = Map (σ ⊔ τ) a
  proof (intro B.join-eqI)
    fix a
    assume a: A.ide a
    show Map σ a ≲B Map (σ ⊔ τ) a
      using 1 a arr-prfx-join-self prfx-char by blast
    show Map τ a ≲B Map (σ ⊔ τ) a
      using join-sym
      by (meson 2 a composite-ofE join-ofE prfx-char)
    show Map (σ ⊔ τ) a \B Map τ a = Map σ a \B Map τ a
      by (metis 1 2 Map-resid-ide a composite-ofE con-prfx-composite-of(1)
          extensional join-ofE joinable-implies-con con-sym)
    show Map (σ ⊔ τ) a \B Map σ a = Map τ a \B Map σ a
      by (metis 1 Map-resid-ide ‹Dom σ = Dom τ› σ τ a arr-prfx-join-self
          join-src joinable-iff-arr-join joinable-implies-con prfx-implies-con
          resid-join_E(3) resid-src-arr con-sym src-simp trg-def)
  qed
  show ∀a. A.ide a ⟶ B.joinable (Map σ a ⊔B Map τ a) (Dom τ a)
    by (metis 3 A.ide-iff-src-self A.ide-implies-arr ‹Dom σ = Dom τ›
        στ.naturality3'_E(2))
```

```
  show ∀ t. (Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom τ t =
          Map (σ ⊔ τ) t
    by (metis 3 A.ide-src B.joinable-iff-join-not-null B.joinable-implies-con
        B.null-is-zero(2) B.residuation-axioms ‹Dom σ = Dom τ›
        στ.F.extensional στ.naturality3'E(1) residuation.conE)
  thus Map (σ ⊔ τ) =
        (λt. (Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom τ t)
    by auto
qed
show σ ⊔ τ =
      (if joinable σ τ
       then MkArr (Dom τ) (Apex σ τ)
                  (λt. (Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom τ t)
       else null)
  using ∗ MkArr-Map joinable-iff-join-not-null by fastforce
show joinable σ τ ⟹
        arr σ ∧ arr τ ∧ Dom σ = Dom τ ∧
        (∀ t. A.arr t ⟶
              B.joinable (Map σ (A.src t) ⊔B Map τ (A.src t)) (Dom σ t))
  using ∗
by (metis (no-types, lifting) B.joinable-iff-arr-join transformation.preserves-arr)
show arr σ ∧ arr τ ∧ Dom σ = Dom τ ∧
      (∀ t. A.arr t ⟶
            B.joinable (Map σ (A.src t) ⊔B Map τ (A.src t)) (Dom σ t))
        ⟹ joinable σ τ
proof −
  assume 1: arr σ ∧ arr τ ∧ Dom σ = Dom τ ∧
            (∀ t. A.arr t ⟶
                  B.joinable (Map σ (A.src t) ⊔B Map τ (A.src t)) (Dom σ t))
  interpret σ: transformation A B ‹Dom σ› ‹Cod σ› ‹Map σ›
    using 1 arr-char by blast
  interpret τ: transformation A B ‹Dom σ› ‹Cod τ› ‹Map τ›
    using 1 arr-char by auto
  interpret Apex: simulation A B ‹Apex σ τ›
    using 1 arr-char Apex-is-simulation
    by (metis A.ide-iff-src-self A.ide-implies-arr B.con-implies-arr(1)
        B.joinable-iff-join-not-null B.joinable-implies-con B.not-arr-null)
  let ?Map = λa. Map σ a ⊔B Map τ a
  interpret στ: transformation-by-components
                  A B ‹Dom σ› ‹Apex σ τ› ?Map
  proof
    show 2: ⋀a. A.ide a ⟹ B.src (?Map a) = Dom σ a
      by (metis (full-types) 1 A.ide-implies-arr A.src-ide B.join-sym
          B.joinable-iff-join-not-null B.src-join B.src-src σ.preserves-src)
    show ⋀a. A.ide a ⟹ B.trg (?Map a) = Apex σ τ a
      by (metis (full-types) 2 A.ide-iff-src-self A.ide-implies-arr Apex-def
          B.con-arr-src(2) B.joinable-iff-arr-join B.joinable-iff-join-not-null
          B.not-ide-null B.null-is-zero(2) B.resid-join_E(1) B.resid-src-arr
          B.src-trg B.trg-def σ.F.preserves-ide τ.naturality2)
```

```
  show ⋀t. A.arr t ⟹
            (Map σ (A.src t) ⊔B Map τ (A.src t)) \B Dom σ t =
            Map σ (A.trg t) ⊔B Map τ (A.trg t)
    by (metis (full-types) 1 B.conE B.conI B.con-sym-ax B.join-def
        B.joinable-implies-con B.null-is-zero(2) B.resid-join_E(3)
        σ.naturality1 τ.naturality1)
  show ⋀t. A.arr t ⟹
          Dom σ t \B (Map σ (A.src t) ⊔B Map τ (A.src t)) = Apex σ τ t
    by (metis 1 Apex-def B.con-implies-arr(2) B.joinable-implies-con
        τ.naturality2 B.joinable-iff-arr-join B.resid-join_E(1) B.con-sym)
  show ⋀t. A.arr t ⟹
           B.joinable (Map σ (A.src t) ⊔B Map τ (A.src t)) (Dom σ t)
    using 1 by blast
qed
let ?Cod-στ = λt. if A.arr t
                  then Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))
                  else B.null
let ?Map-στ = λt. if A.arr t
                  then (Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom σ t
                  else B.null
have στ': transformation A B (Dom τ) ?Cod-στ ?Map-στ
  using 1 Apex-def στ.map-eq στ.transformation-axioms by presburger
let ?στ = MkArr (Dom σ) (Apex σ τ) στ.map
have στ: arr ?στ
  using arr-char στ.transformation-axioms by blast
have con-σ-στ: σ ⌢ ?στ
  using 1 con-char σ.transformation-axioms στ.transformation-axioms
    στ.map-simp-ide
  by auto
    (metis (no-types, lifting) A.ide-implies-arr B.arr-prfx-join-self
     B.joinable-iff-join-not-null B.not-arr-null B.prfx-implies-con
     transformation.preserves-arr)
have con-τ-στ: τ ⌢ ?στ
  using 1 con-char τ.transformation-axioms στ.transformation-axioms
    στ.map-simp-ide
  by auto
    (metis (no-types, lifting) A.ide-implies-arr B.arr-prfx-join-self B.conI
     B.join-sym B.joinable-iff-join-not-null B.not-arr-null B.not-ide-null
     transformation.preserves-arr)
have 4: Apex σ ?στ = Apex σ τ
proof
  fix t
  have A.arr t ⟹
         (Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))) \B
           (Map σ (A.src t) \B (Map σ (A.src t) ⊔B Map τ (A.src t))) =
          Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))
  proof −
    assume t: A.arr t
    have 4: Map σ (A.src t) ≲B
```

```
          (Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom τ (A.src t)
      using t 1 B.arr-prfx-join-self B.joinable-iff-join-not-null στ.map-def
          στ.map-simp-ide
      by (metis (no-types, lifting) A.arr-src-iff-arr A.ide-src A.src-src)
    have Map σ (A.src t) \B
          ((Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom τ (A.src t)) =
        B.src (Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t)))
      by (metis (no-types, lifting) 1 4 A.ide-src A.src-src
          Apex.preserves-reflects-arr B.ide-iff-src-self B.ide-implies-arr
          B.not-arr-null B.src-resid στ.map-def στ.map-simp-ide στ.naturality2
          Apex-def B.conI t)
    thus (Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))) \B
          (Map σ (A.src t) \B (Map σ (A.src t) ⊔B Map τ (A.src t))) =
        Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))
      by (metis (no-types, lifting) 1 4 A.ide-src A.src-src B.arr-src-iff-arr
          B.prfx-implies-con B.resid-arr-src στ.map-def στ.map-simp-ide
          B.arr-resid t)
  qed
  thus Apex σ ?στ t = Apex σ τ t
    unfolding Apex-def by simp
qed
have 5: Apex τ ?στ = Apex σ τ
proof
  fix t
  have A.arr t ⟹
        (Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))) \B
          (Map τ (A.src t) \B (Map σ (A.src t) ⊔B Map τ (A.src t))) =
        Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))
  proof −
    assume t: A.arr t
    show (Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))) \B
          (Map τ (A.src t) \B (Map σ (A.src t) ⊔B Map τ (A.src t))) =
        Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))
    proof −
      have Map τ (A.src t) \B (Map σ (A.src t) ⊔B Map τ (A.src t)) =
          B.src (Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t)))
      proof −
        have Map τ (A.src t) \B (Map σ (A.src t) ⊔B Map τ (A.src t)) =
            (Map τ (A.src t) \B Map τ (A.src t)) \B
              (Map σ (A.src t) \B Map τ (A.src t))
          by (metis (no-types, lifting) 1 A.arr-src-if-arr A.ide-src
              B.arr-prfx-join-self B.join-sym B.joinable-iff-join-not-null
              B.prfx-implies-con B.resid-join_E(1) στ.map-def στ.map-simp-ide
              t)
        also have ... = B.trg (Map σ (A.src t) \B Map τ (A.src t))
          using B.apex-sym B.cube B.trg-def by auto
        also have
            ... = B.src (Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t)))
          by (metis Apex.preserves-reflects-arr Apex-def B.arr-resid-iff-con
```

```
          B.src-resid t)
      finally show ?thesis by blast
    qed
    thus ?thesis
      by (metis B.arr-resid-iff-con B.con-def B.con-implies-arr(1)
          B.resid-arr-src)
  qed
 qed
 thus Apex τ ?στ t = Apex σ τ t
  unfolding Apex-def by simp
qed
have σ ⊔ τ = MkArr (Dom σ) (Apex σ τ) στ.map
proof (intro join-eqI)
 show σ ≲ MkArr (Dom σ) (Apex σ τ) στ.map
 proof −
  have σ \ MkArr (Dom σ) (Apex σ τ) στ.map = trg ?στ
  proof −
   have Dom (σ \ MkArr (Dom σ) (Apex σ τ) στ.map) = Dom (trg ?στ)
    using στ con-σ-στ trg-simp by fastforce
   moreover
   have Cod (σ \ MkArr (Dom σ) (Apex σ τ) στ.map) = Cod (trg ?στ)
    using στ 4 con-σ-στ trg-simp by fastforce
   moreover
   have Map (σ \ MkArr (Dom σ) (Apex σ τ) στ.map) = Map (trg ?στ)
   proof −
    have Map (σ \ MkArr (Dom σ) (Apex σ τ) στ.map) = Apex σ τ
    proof
     fix t
     show Map (σ \ MkArr (Dom σ) (Apex σ τ) στ.map) t =
          Apex σ τ t
     proof (cases A.arr t)
      show ¬ A.arr t ⟹ ?thesis
       by (simp add: Apex.extensional con-σ-στ)
      assume t: A.arr t
      have Map σ (A.src t) \B
             (Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Apex σ τ t =
           Apex σ τ t
       by (metis Apex.preserves-reflects-arr B.arr-prfx-join-self
           B.arr-resid-iff-con B.con-implies-arr(2) B.con-target
           B.join-prfx(1) B.joinable-iff-arr-join
           B.joinable-implies-con B.resid-ide-arr στ.joinable
           στ.naturality2 t)
      thus ?thesis
       by (simp add: con-σ-στ t)
     qed
    qed
    thus ?thesis
     using στ trg-simp by force
   qed
```

```
    ultimately show ?thesis
      using στ con-σ-στ resid-def' trg-simp by fastforce
  qed
  thus ?thesis
    using στ ide-trg by presburger
qed
show τ ≲ MkArr (Dom σ) (Apex σ τ) στ.map
proof −
  have τ \ MkArr (Dom σ) (Apex σ τ) στ.map = trg ?στ
  proof −
   have Dom (τ \ MkArr (Dom σ) (Apex σ τ) στ.map) = Dom (trg ?στ)
     using στ con-τ-στ trg-def by fastforce
    moreover have Cod (τ \ MkArr (Dom σ) (Apex σ τ) στ.map) =
                  Cod (trg ?στ)
     using στ 5 Cod.simps(1) Cod-resid con-τ-στ trg-simp by presburger
    moreover have Map (τ \ MkArr (Dom σ) (Apex σ τ) στ.map) =
                  Map (trg ?στ)
    proof −
      have Map (τ \ MkArr (Dom σ) (Apex σ τ) στ.map) = Apex σ τ
      proof
        fix t
        show Map (τ \ MkArr (Dom σ) (Apex σ τ) στ.map) t =
             Apex σ τ t
        proof (cases A.arr t)
          show ¬ A.arr t ⟹ ?thesis
            by (simp add: Apex.extensional con-τ-στ)
          assume t: A.arr t
          have Map (τ \ MkArr (Dom σ) (Apex σ τ) στ.map) t =
               Map τ (A.src t) \B στ.map (A.src t) ⊔B Apex σ τ t
            using στ Map-resid
            by (simp add: con-τ-στ t)
          also have ... = Map τ (A.src t) \B
                            ((Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B
                               Dom σ (A.src t))
                            ⊔B Apex σ τ t
            using t στ.map-def by simp
          also have ... = ((Map τ (A.src t) \B
                             (Map σ (A.src t) ⊔B Map τ (A.src t))) \B
                             (Dom σ (A.src t) \B
                                (Map σ (A.src t) ⊔B Map τ (A.src t))))
                             ⊔B Apex σ τ t
            by (metis (no-types, lifting) t A.arr-src-iff-arr
                A.ide-src A.src-src A.trg-src B.cube στ.map-def
                στ.map-simp-ide στ.naturality1 τ.naturality1)
          also have ... = (((Map τ (A.src t) \B Map τ (A.src t)) \B
                             (Map σ (A.src t) \B Map τ (A.src t))) \B
                            (Dom σ (A.src t) \B
                               (Map σ (A.src t) ⊔B Map τ (A.src t))))
                            ⊔B Apex σ τ t
```

```
      by (metis (no-types, lifting) A.arr-src-iff-arr A.ide-src
          B.arr-prfx-join-self B.join-sym
          B.joinable-iff-join-not-null B.not-arr-null
          B.prfx-implies-con B.resid-join_E(1) στ.map-simp-ide
          στ.preserves-arr t)
    also have ... = ((Cod τ (A.src t)) \_B
                     (Map σ (A.src t) \_B Map τ (A.src t))) \_B
                    (Dom σ (A.src t) \_B
                      (Map σ (A.src t) ⊔_B Map τ (A.src t))))
                    ⊔_B Apex σ τ t
      by (simp add: t τ.preserves-trg B.resid-arr-self)
    also have ... = (Apex σ τ (A.src t) \_B
                    (Dom σ (A.src t) \_B
                      (Map σ (A.src t) ⊔_B Map τ (A.src t))))
                    ⊔_B Apex σ τ t
      by (simp add: Apex-def τ.G.extensional)
    also have ... = (Apex σ τ (A.src t) \_B Apex σ τ (A.src t)) ⊔_B
                    Apex σ τ t
      by (metis A.arr-src-iff-arr A.src-src στ.naturality2 t)
    also have ... = Apex σ τ (A.src t) ⊔_B Apex σ τ t
      by (metis A.ideE A.ide-src Apex.preserves-resid t)
    also have ... = Apex σ τ t
      by (metis A.ide-src Apex.preserves-reflects-arr
          B.arr-resid-iff-con B.join-src B.src-resid στ.naturality2
          στ.preserves-trg t)
    finally show ?thesis by blast
  qed
 qed
 thus ?thesis
  using στ trg-simp by force
qed
ultimately show ?thesis
 using στ con-τ-στ resid-def' trg-simp by fastforce
qed
thus ?thesis
 using στ ide-trg by presburger
qed
have 5: ⋀a. A.ide a ⟹ Map σ a ⌢_B Map τ a
 by (metis A.ide-iff-src-self B.not-ide-null B.null-is-zero(2)
     B.residuation-axioms Apex.preserves-ide Apex-def residuation.con-def)
have 6: ∀ a. A.ide a ⟶
          (Map σ a ⊔_B Map τ a) ⊔_B Dom τ a ⌢_B Map σ a
 by (metis (no-types, lifting) 1 A.ide-iff-src-self
     B.arr-prfx-join-self B.conI B.con-sym B.join-def B.not-arr-null
     B.prfx-implies-con B.src-def σ.F.preserves-ide στ.map-eq
     στ.map-simp-ide στ.preserves-src)
show MkArr (Dom σ) (Apex σ τ) στ.map \ τ = σ \ τ
proof −
 have ⋀a. A.ide a ⟹
```

```
          (Map σ a ⊔B Map τ a) ⊔B Dom τ a ⌢B Map τ a
  using B.con-sym στ.map-def con-τ-στ con-char by force
moreover
have transformation A B (Dom τ) ?Cod-στ ?Map-στ
  using στ' by simp
moreover
have (λt. if A.arr t
          then Cod τ t \B
                 (Map (MkArr (Dom σ) ?Cod-στ ?Map-στ) (A.src t) \B
                   Map τ (A.src t))
          else B.null) =
     ?Cod-στ
proof
  fix t
  show (if A.arr t
        then Cod τ t \B
               (Map (MkArr (Dom σ) ?Cod-στ ?Map-στ) (A.src t) \B
                 Map τ (A.src t))
        else B.null) =
       (if A.arr t
        then Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))
        else B.null)
  proof −
    have A.arr t ⟹
           Cod τ t \B
           (((Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom σ (A.src t))
               \B Map τ (A.src t)) =
           Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))
    proof −
      assume t: A.arr t
      show Cod τ t \B
           (((Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom σ (A.src t))
               \B Map τ (A.src t)) =
           Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))
      proof −
        have Cod τ t \B
             (((Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B Dom σ (A.src t))
                 \B Map τ (A.src t)) =
             Cod τ t \B
               ((Map σ (A.src t) ⊔B Map τ (A.src t)) \B
                 Map τ (A.src t))
          using στ.map-def στ.map-simp-ide t by force
        also have ... = Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t)
⊔B
                                    Map τ (A.src t) \B Map τ (A.src t))
          by (metis 1 B.arr-prfx-join-self B.conE B.con-sym-ax B.join-def
              B.join-sym B.null-is-zero(2) B.prfx-implies-con
              B.resid-join_E(3) t)
        also have ... = Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t)
```

```
⊔B
                                        Cod τ (A.src t))
          using A.ide-src A.trg-src B.trg-def τ.preserves-trg t by presburger
          also have ... = Cod τ t \B (Map σ (A.src t) \B Map τ (A.src t))
            by (metis 5 A.ide-src B.arr-resid-iff-con B.join-src B.join-sym
                B.src-resid τ.preserves-trg t)
          finally show ?thesis by blast
        qed
      qed
      thus ?thesis by auto
    qed
  qed
  moreover
  have (λt. if A.arr t
            then (Map (MkArr (Dom σ) ?Cod-στ ?Map-στ) (A.src t) \B
                      Map τ (A.src t))
                 ⊔B Cod τ t
            else B.null) =
       (λt. if A.arr t
            then Map σ (A.src t) \B Map τ (A.src t) ⊔B Cod τ t
            else B.null)
  proof
    fix t
    show (if A.arr t
          then Map (MkArr (Dom σ) ?Cod-στ ?Map-στ) (A.src t) \B
               Map τ (A.src t)
               ⊔B Cod τ t
          else B.null) =
         (if A.arr t
          then Map σ (A.src t) \B Map τ (A.src t) ⊔B Cod τ t
          else B.null)
    proof −
      have A.arr t ⟹
           ((Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B
              Dom σ (A.src t)) \B
            Map τ (A.src t) ⊔B Cod τ t =
           Map σ (A.src t) \B Map τ (A.src t) ⊔B Cod τ t
      proof −
        assume t: A.arr t
        show ((Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B
                Dom σ (A.src t)) \B
              Map τ (A.src t) ⊔B Cod τ t =
             Map σ (A.src t) \B Map τ (A.src t) ⊔B Cod τ t
        proof −
          have ((Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B
                  Dom σ (A.src t)) \B
                Map τ (A.src t) ⊔B Cod τ t =
               ((Map σ (A.src t) ⊔B Map τ (A.src t)) \B
                  Map τ (A.src t))
```

```
                  ⊔B Cod τ t
          using στ.map-def στ.map-simp-ide t by fastforce
        also have ... = (Map σ (A.src t) \B Map τ (A.src t) ⊔B
                           Map τ (A.src t) \B Map τ (A.src t))
                         ⊔B Cod τ t
          by (metis 1 B.arr-prfx-join-self B.conE B.con-sym-ax B.join-def
          B.join-sym B.null-is-zero(2) B.prfx-implies-con B.resid-joinE(3)
              t)
        also have ... =
                  (Map σ (A.src t) \B Map τ (A.src t) ⊔B Cod τ (A.src t))
                      ⊔B Cod τ t
          by (simp add: τ.preserves-trg B.resid-arr-self t)
        also have ... = (Map σ (A.src t) \B Map τ (A.src t)) ⊔B Cod τ t
          by (metis 5 A.ide-src B.arr-resid-iff-con B.join-src B.join-sym
              B.src-resid τ.preserves-trg t)
        finally show ?thesis by blast
      qed
    qed
    thus ?thesis by auto
  qed
 qed
 ultimately show ?thesis
  unfolding resid-def στ.map-eq Apex-def
  using 1 5 arr-char σ.transformation-axioms
  by auto
qed
show MkArr (Dom σ) (Apex σ τ) στ.map \ σ = τ \ σ
proof −
 have (λt. if A.arr t
           then Cod σ t \B
                  (Map (MkArr (Dom σ) ?Cod-στ ?Map-στ) (A.src t) \B
                    Map σ (A.src t))
           else B.null) =
      (λt. if A.arr t
           then Cod σ t \B (Map τ (A.src t) \B Map σ (A.src t))
           else B.null)
 proof
  fix t
  show (if A.arr t
        then Cod σ t \B
               (Map (MkArr (Dom σ) ?Cod-στ ?Map-στ) (A.src t) \B
                 Map σ (A.src t))
        else B.null) =
       (if A.arr t
        then Cod σ t \B (Map τ (A.src t) \B Map σ (A.src t))
        else B.null)
  proof (cases A.arr t)
   show ¬ A.arr t ⟹ ?thesis
    by simp
```

```
    assume t: A.arr t
    show ?thesis
    proof −
      have Cod σ t \_B
             (((Map σ (A.src t) ⊔_B Map τ (A.src t)) ⊔_B
                 Dom σ (A.src t)) \_B
               Map σ (A.src t)) =
           Cod σ t \_B
             ((Map σ (A.src t) ⊔_B Map τ (A.src t)) \_B
               Map σ (A.src t))
        using στ.map-def στ.map-simp-ide t by fastforce
      also have ... = Cod σ t \_B
                        ((Map σ (A.src t) \_B Map σ (A.src t)) ⊔_B
                          (Map τ (A.src t) \_B Map σ (A.src t)))
        by (metis 1 B.arr-prfx-join-self B.conE B.con-sym-ax B.join-def
            B.null-is-zero(2) B.prfx-implies-con B.resid-join_E(3) t)
      also have ... = Cod σ t \_B
                        (Cod σ (A.src t) ⊔_B
                          (Map τ (A.src t) \_B Map σ (A.src t)))
       using A.ide-src A.trg-src B.trg-def σ.preserves-trg t by presburger
      also have ... = Cod σ t \_B (Map τ (A.src t) \_B Map σ (A.src t))
        by (metis (no-types, lifting) 5 A.ide-src B.arr-resid-iff-con
            B.conI B.join-src B.src-resid σ.preserves-trg B.con-sym-ax t)
      finally show ?thesis by auto
    qed
  qed
qed
moreover
have (λt. if A.arr t
          then (Map (MkArr (Dom σ) ?Cod-στ ?Map-στ) (A.src t) \_B
                  Map σ (A.src t))
               ⊔_B Cod σ t
          else B.null) =
     (λt. if A.arr t
          then Map τ (A.src t) \_B Map σ (A.src t) ⊔_B Cod σ t
          else B.null)
proof
  fix t
  show (if A.arr t
        then (Map (MkArr (Dom σ) ?Cod-στ ?Map-στ) (A.src t) \_B
                Map σ (A.src t))
             ⊔_B Cod σ t
        else B.null) =
       (if A.arr t
        then Map τ (A.src t) \_B Map σ (A.src t) ⊔_B Cod σ t
        else B.null)
  proof (cases A.arr t)
    show ¬ A.arr t ⟹ ?thesis
      by simp
```

```
          assume t: A.arr t
          show ?thesis
          proof −
            have (((Map σ (A.src t) ⊔B Map τ (A.src t)) ⊔B
                      Dom σ (A.src t)) \B
                    Map σ (A.src t)) ⊔B Cod σ t =
                  ((Map σ (A.src t) ⊔B Map τ (A.src t)) \B
                    Map σ (A.src t)) ⊔B Cod σ t
              using στ.map-def στ.map-simp-ide t by fastforce
            also have ... = (Map σ (A.src t) \B Map σ (A.src t) ⊔B
                             Map τ (A.src t) \B Map σ (A.src t))
                             ⊔B Cod σ t
              by (metis Apex.preserves-reflects-arr B.arr-prfx-join-self B.conI
                  B.joinable-iff-join-not-null B.not-arr-null B.not-ide-null
                  B.null-is-zero(2) B.resid-joinE(3) στ.naturality2 t)
            also have ... = (Cod σ (A.src t) ⊔B
                             (Map τ (A.src t) \B Map σ (A.src t)))
                             ⊔B Cod σ t
              using A.ide-src A.trg-src B.trg-def σ.preserves-trg t
              by presburger
            also have ... = (Map τ (A.src t) \B Map σ (A.src t)) ⊔B Cod σ t
              by (metis (no-types, lifting) 5 A.ide-src B.arr-resid-iff-con
                  B.conI B.join-src B.src-resid σ.preserves-trg B.con-sym-ax t)
            finally show ?thesis by auto
          qed
        qed
      qed
      ultimately show ?thesis
        unfolding resid-def στ.map-eq Apex-def
        using 1 5 6 στ' σ.transformation-axioms arr-char B.con-sym
        by simp
    qed
  qed
  thus joinable σ τ
    by (metis στ joinable-iff-join-not-null not-arr-null)
 qed
qed

lemma Dom-join:
assumes joinable σ τ
shows Dom (σ ⊔ τ) = Dom σ
 using assms join-char by auto

lemma Cod-join:
assumes joinable σ τ
shows Cod (σ ⊔ τ) = Apex σ τ
 using assms join-char by auto

lemma Map-join:
```

assumes *joinable* σ τ
shows *Map* $(\sigma \sqcup \tau) =$
$(\lambda t.\ (Map\ \sigma\ (A.src\ t)) \sqcup_B Map\ \tau\ (A.src\ t)) \sqcup_B Dom\ \tau\ t)$
using *assms join-char* **by** *auto*

end

3.10.1 Exponential of Small RTS's

locale *exponential-of-small-rts* =
A: *small-rts A* +
B: *small-rts B* +
exponential-rts
begin

lemma *small-Collect-fun*:
shows *small* $\{F.\ F$ ‘ *Collect A.arr* $\subseteq$ *Collect B.arr* $\wedge$
F ‘ $(UNIV - Collect\ A.arr) \subseteq \{B.null\}\}$
proof −
let *?*$\mathcal{F} = \{F.\ F$ ‘ *Collect A.arr* $\subseteq$ *Collect B.arr* $\wedge$
F ‘ $(UNIV - Collect\ A.arr) \subseteq \{B.null\}\}$
obtain φ **where** φ: *inj-on* φ *(Collect A.arr)* $\wedge$ φ ‘ *Collect A.arr* $\in$ *range elts*
using *A.small small-def* **by** *metis*
obtain ψ **where** ψ: *inj-on* ψ *(Collect B.arr)* $\wedge$ ψ ‘ *Collect B.arr* $\in$ *range elts*
using *B.small small-def* **by** *metis*
let *?graph* $= \lambda F :: 'a \Rightarrow 'b.$ *set* $((\lambda x.\ vpair\ (\varphi\ x)\ (\psi\ (F\ x)))$ ‘ *Collect A.arr)*
have *?graph* ‘ *?*$\mathcal{F} \subseteq$ *elts* *(VPow (vtimes (set* $(\varphi$ ‘ *Collect A.arr))*
(set $(\psi$ ‘ *Collect B.arr))))*
using *A.small B.small small-def*
by (*simp add*: *image-subset-iff set-image-le-iff*)
moreover have *inj-on ?graph ?*$\mathcal{F}$
proof (*intro inj-onI*)
fix *F G*
assume *F*: $F \in$ *?*$\mathcal{F}$ **and** *G*: $G \in$ *?*$\mathcal{F}$
and *eq*: *?graph F* = *?graph G*
show $F = G$
proof
fix x
show $F\ x = G\ x$
proof (*cases A.arr x*)
show $\neg$ *A.arr x* $\Longrightarrow$ *?thesis*
using *F G*
by (*simp add*: *image-subset-iff*)
assume *x*: *A.arr x*
have *?graph F* = *?graph G*
using *eq* **by** *simp*
hence $(\lambda x.\ vpair\ (\varphi\ x)\ (\psi\ (F\ x)))$ ‘ *Collect A.arr* =
$(\lambda x.\ vpair\ (\varphi\ x)\ (\psi\ (G\ x)))$ ‘ *Collect A.arr*
using *A.small* **by** *auto*

```
      hence ∃x'. A.arr x' ∧ vpair (φ x) (ψ (F x)) = vpair (φ x') (ψ (G x'))
        using x by blast
      hence vpair (φ x) (ψ (F x)) = vpair (φ x) (ψ (G x))
        by (metis x φ inj-onD mem-Collect-eq vpair-inject)
      hence ψ (F x) = ψ (G x)
        by blast
      thus ?thesis
        using x F G ψ inj-onD [of ψ Collect B.arr F x G x] by blast
    qed
  qed
 qed
 ultimately show ?thesis
  by (meson down-raw small-def)
qed

lemma small-Collect-simulation:
shows small (Collect (simulation A B))
proof −
 have ⋀F. simulation A B F ⟹
            F ‘ Collect A.arr ⊆ Collect B.arr ∧
            F ‘ (UNIV − Collect A.arr) ⊆ {B.null}
  apply (intro conjI)
   apply (simp add: image-subset-iff simulation.preserves-reflects-arr)
  using simulation.extensional by fastforce
 thus ?thesis
 by (metis (no-types, lifting) Collect-mono small-Collect-fun smaller-than-small)
qed

lemma small-Collect-transformation:
assumes simulation A B F and simulation A B G
shows small (Collect (transformation A B F G))
proof −
 have ⋀τ. transformation A B F G τ ⟹
            τ ‘ Collect A.arr ⊆ Collect B.arr ∧
            τ ‘ (UNIV − Collect A.arr) ⊆ {B.null}
  by (metis (mono-tags, lifting) DiffD2 image-subsetI mem-Collect-eq
      singleton-iff transformation.extensional transformation.preserves-arr)
 thus ?thesis
  by (metis (no-types, lifting) Collect-mono small-Collect-fun
      smaller-than-small)
qed

sublocale small-rts resid
proof
 have small (⋃FG∈Collect (simulation A B) × Collect (simulation A B).
                  {FG} × Collect (transformation A B (fst FG) (snd FG)))
 proof −
  have small (Collect (simulation A B) × Collect (simulation A B))
   using small-Collect-simulation by fastforce
```

```
    moreover
    have ⋀FG. FG ∈ Collect (simulation A B) × Collect (simulation A B) ⟹
                small ({FG} × Collect (transformation A B (fst FG) (snd FG)))
      using small-Collect-transformation by force
    ultimately show ?thesis by blast
  qed
  moreover have (λt. ((Dom t, Cod t), Map t)) ‘ Collect arr ⊆
        (⋃FG∈Collect (simulation A B) × Collect (simulation A B).
          {FG} × Collect (transformation A B (fst FG) (snd FG)))
  proof
    fix T
    assume T: T ∈ (λt. ((Dom t, Cod t), Map t)) ‘ Collect arr
    obtain t where t: arr t ∧ T = ((Dom t, Cod t), Map t)
      using T by blast
    have simulation A B (Dom t) ∧ simulation A B (Cod t) ∧
         transformation A B (Dom t) (Cod t) (Map t)
      by (meson arr-char t transformation-def)
    thus T ∈
          (⋃FG∈Collect (simulation A B) × Collect (simulation A B).
          {FG} × Collect (transformation A B (fst FG) (snd FG)))
      using t by simp
  qed
  ultimately have small ((λt. ((Dom t, Cod t), Map t)) ‘ Collect arr)
    using smaller-than-small by blast
  moreover have inj-on (λt. ((Dom t, Cod t), Map t)) (Collect arr)
    using not-arr-null null-char MkArr-Map
    by (intro inj-onI) (metis fst-conv mem-Collect-eq snd-eqD)
  ultimately show small (Collect arr) by auto
qed

lemma is-small-rts:
shows small-rts resid
  ..

end
```

3.10.2 Exponential into RTS with Composites

```
locale exponential-into-rts-with-composites =
  A: rts A +
  B: rts-with-composites B +
  exponential-rts
begin

  interpretation B: extensional-rts B ..
  interpretation B: extensional-rts-with-composites B ..

  notation B.comp  (infixr ·B 55)
```

```
abbreviation COMP :: ('a, 'b) arr ⇒ ('a, 'b) arr ⇒ ('a, 'b) arr
where COMP t u ≡ MkArr (Dom t) (Cod u)
                   (λx. Map t (A.src x) ·_B Map u (A.src x) ⊔_B Dom t x)

lemma composite-of-iff:
shows composite-of t u v ⟷ seq t u ∧ v = COMP t u
proof
  show ⋀v. seq t u ∧ v = COMP t u ⟹ composite-of t u v
  proof (elim conjE)
    fix v
    assume tu: seq t u
    interpret T: transformation A B ‹Dom t› ‹Cod t› ‹Map t›
      using tu arr-char by blast
    interpret U: transformation A B ‹Cod t› ‹Cod u› ‹Map u›
      using tu arr-char src-simp
      by (metis Map-src Map-trg seqE_W_E)
    interpret TU: transformation-by-components A B ‹Dom t› ‹Cod u›
                    ‹λa. B.comp (Map t a) (Map u a)›
    proof
      show ⋀a. A.ide a ⟹ B.src (B.comp (Map t a) (Map u a)) = Dom t a
        by (metis A.ide-implies-arr B.rts-with-composites-axioms B.seqI_W_E(2)
            T.preserves-src T.preserves-trg U.preserves-arr U.preserves-src
            exponential-rts-axioms exponential-rts-def
            extensional-rts-with-composites.src-comp_EC
            extensional-rts-with-composites-def)
      show 1: ⋀a. A.ide a ⟹ B.trg (B.comp (Map t a) (Map u a)) = Cod u a
        by (metis A.ide-implies-arr B.arr-src-iff-arr B.composable-iff-arr-comp
            B.trg-comp T.F.preserves-reflects-arr U.preserves-trg
            ‹⋀a. A.ide a ⟹ B.src (B.comp (Map t a) (Map u a)) = Dom t a›)
      fix x
      assume x: A.arr x
      show B.comp (Map t (A.src x)) (Map u (A.src x)) \_B Dom t x =
           B.comp (Map t (A.trg x)) (Map u (A.trg x))
      proof −
        have B.comp (Map t (A.src x)) (Map u (A.src x)) \_B Dom t x =
             B.comp (Map t (A.src x) \_B Dom t x)
                    (Map u (A.src x) \_B (Dom t x \_B Map t (A.src x)))
          by (metis (full-types) 1 A.arr-src-iff-arr A.ide-src B.arr-resid-iff-con
              B.arr-trg-iff-arr B.comp-def B.con-compI(2) B.not-arr-null
              T.transformation-axioms U.G.preserves-reflects-arr
              U.transformation-axioms exponential-rts-axioms exponential-rts-def
              extensional-rts.resid-comp(2) transformation.naturality2-ax x)
        also have ... = B.comp (Map t (A.trg x)) (Map u (A.trg x))
          using T.naturality1 T.naturality2 U.naturality1 by presburger
        finally
        show B.comp (Map t (A.src x)) (Map u (A.src x)) \_B Dom t x =
             B.comp (Map t (A.trg x)) (Map u (A.trg x))
          by blast
      qed
```

```
  show 1: Dom t x \_B B.comp (Map t (A.src x)) (Map u (A.src x)) =
          Cod u x
    using x
    by (metis A.arr-trg-iff-arr A.ide-trg B.arr-resid-iff-con B.arr-trg-iff-arr
      B.resid-comp(1) T.naturality2 U.G.preserves-reflects-arr U.naturality2
        ‹B.comp (Map t (A.src x)) (Map u (A.src x)) \_B Dom t x =
         B.comp (Map t (A.trg x)) (Map u (A.trg x))›
        ‹⋀a. A.ide a ⟹ B.trg (B.comp (Map t a) (Map u a)) = Cod u a›)
  show B.joinable (B.comp (Map t (A.src x)) (Map u (A.src x))) (Dom t x)
    using x B.joinable-iff-con
    by (metis 1 B.conI B.con-sym-ax B.not-arr-null
        U.G.preserves-reflects-arr)
qed
have composite-of t u (MkArr (Dom t) (Cod u) TU.map)
proof
  have 1: arr (MkArr (Dom t) (Cod u) TU.map)
    using arr-char TU.transformation-axioms by blast
  have src (MkArr (Dom t) (Cod u) TU.map) = src t
    using 1 tu src-simp
    by (metis (no-types, lifting) Dom.simps(1) seqE_WE)
  have 3: trg (MkArr (Dom t) (Cod u) TU.map) = trg u
    using 1 trg-simp tu
    by (metis (no-types, lifting) Cod.simps(1) seqE_WE)
  have ∀a. A.ide a ⟶
             Map t a ≲_B Map (MkArr (Dom t) (Cod u) TU.map) a
    using TU.map-simp-ide
    by (metis (no-types, lifting) A.ide-implies-arr B.prfx-comp
        Map.simps(1) TU.preserves-arr)
  thus 2: t ≲ MkArr (Dom t) (Cod u) TU.map
    using 1 src-simp tu prfx-char [of t MkArr (Dom t) (Cod u) TU.map]
    by auto
  have MkArr (Dom t) (Cod u) TU.map \ t = u
  proof −
    have Dom (MkArr (Dom t) (Cod u) TU.map \ t) = Dom u
      by (metis (mono-tags, lifting) 2 Dom-resid conI con-sym-ax
          exponential-rts.Map-src exponential-rts.Map-trg
          exponential-rts.ide-char_ERTS exponential-rts-axioms seqE_WE tu)
    moreover have Cod (MkArr (Dom t) (Cod u) TU.map \ t) = Cod u
      by (metis (no-types, lifting) 1 2 Cod.simps(1) Map-trg apex-sym
          arr-resid conI con-sym-ax not-ide-null src-ide src-resid trg-ide)
    moreover have Map (MkArr (Dom t) (Cod u) TU.map \ t) = Map u
    proof −
      have ⋀a. A.ide a ⟹
                 Map (MkArr (Dom t) (Cod u) TU.map \ t) a = Map u a
        by (metis (no-types, lifting) 2 A.ide-implies-arr B.comp-resid-prfx
            Map.simps(1) Map-resid-ide TU.map-simp-ide TU.preserves-arr
            conI con-sym-ax not-ide-null)
      thus ?thesis
        using transformation-eqI
```

```
                [of A B Dom u Cod u
                    Map (MkArr (Dom t) (Cod u) TU.map \ t) Map u]
              2 arr-char U.transformation-axioms
        by (metis (no-types, lifting) B.extensional-rts-axioms Dom-resid
            arr-resid calculation(1−2) conI con-sym-ax not-ide-null)
    qed
    ultimately show ?thesis
      using 1
      by (metis (no-types, lifting) 2 MkArr-Map arr-char con-sym-ax
          not-ide-null null-char seqE_W_E tu)
  qed
  thus cong (MkArr (Dom t) (Cod u) TU.map \ t) u
    by (metis (full-types) 1 3 ide-trg trg-def)
 qed
 thus v = COMP t u ⟹ composite-of t u v
  unfolding TU.map-def by blast
qed
thus composite-of t u v ⟹ seq t u ∧ v = COMP t u
 by (metis (no-types, lifting) arrE arr-composite-of comp-is-composite-of(2)
     composable-imp-seq comp-def)
qed

corollary is-rts-with-composites:
shows rts-with-composites resid
 using composite-of-iff composable-def
 by unfold-locales auto

sublocale rts-with-composites resid
 using is-rts-with-composites by blast
sublocale extensional-rts-with-composites resid ..

lemma naturality:
assumes arr τ and A.arr u
shows Dom τ u ·_B Map τ (A.trg u) = Map τ (A.src u) ·_B Cod τ u
proof −
 interpret τ: transformation A B ‹Dom τ› ‹Cod τ› ‹Map τ›
  using assms arr-char by blast
 show ?thesis
  using assms(2) τ.naturality1 τ.naturality2 τ.naturality3
  by (metis B.diamond-commutes)
qed

lemma Dom-comp [simp]:
assumes seq σ τ
shows Dom (comp σ τ) = Dom σ
 using assms
 by (metis Map-src composable-iff-arr-comp has-composites seqE_W_E src-comp)

lemma Cod-comp [simp]:
```

```
  assumes seq σ τ
  shows Cod (comp σ τ) = Cod τ
    using assms
    by (metis Map-trg composable-iff-arr-comp has-composites seqE_{W E} trg-comp)

  lemma Map-comp [simp]:
  assumes seq σ τ
  shows Map (comp σ τ) =
        (λx. Map σ (A.src x) ·_B Map τ (A.src x) ⊔_B Dom σ x)
    using assms composite-of-iff comp-is-composite-of has-composites
    by simp

  lemma Map-comp-ide:
  assumes seq σ τ and A.ide x
  shows Map (comp σ τ) x = Map σ x ·_B Map τ x
  proof −
    have 1: B.src (Map τ x) = Dom τ (A.src x)
      using assms
      by (metis A.src-ide seq-char transformation.preserves-src)
    have (Map σ (A.src x) ·_B Map τ (A.src x)) ⊔_B Dom σ x =
          Map σ (A.src x) ·_B Map τ (A.src x)
    proof −
      have B.seq (Map σ (A.src x)) (Map τ (A.src x))
        by (metis assms(1−2) 1 A.ideE A.ide-implies-arr A.src-ide
            B.seqI_{W E}(2) B.trg-def seq-char resid-Map-self seqE
            transformation.preserves-arr)
      hence B.src (Map σ (A.src x) ·_B Map τ (A.src x)) = Dom σ x
        using assms A.src-ide seq-char transformation.preserves-src B.src-comp
        by (metis B.composable-iff-seq)
      thus ?thesis
       using assms B.join-src [of Map σ (A.src x) ·_B Map τ (A.src x)] B.join-sym
             B.seq-implies-arr-comp ‹B.seq (Map σ (A.src x)) (Map τ (A.src x))›
        by presburger
    qed
    thus ?thesis
      using assms by force
  qed

end
```

3.10.3 Exponential by One

The isomorphism between an RTS A and the exponential $[\mathbf{1}, A]$ is important in various situations.

```
  locale exponential-by-One =
    One: one-arr-rts +
    A: extensional-rts A
  for A :: 'a resid      (infix \_A 70)
  begin
```

```
sublocale exponential-rts One.resid A ..
notation resid  (infix \ 70)
notation con    (infix ⌢ 50)

abbreviation Up :: 'a ⇒ ('b, 'a) arr
where Up t ≡
      if A.arr t
      then MkArr
           (constant-simulation.map One.resid A (A.src t))
           (constant-simulation.map One.resid A (A.trg t))
           (constant-transformation.map One.resid A t)
      else null

abbreviation Dn :: ('b, 'a) arr ⇒ 'a
where Dn t ≡ if arr t then Map t One.the-arr else A.null

sublocale Up: simulation A resid Up
proof
  show ⋀t. ¬ A.arr t ⟹ Up t = null
    by simp
  fix t u
  assume tu: t ⌢A u
  interpret T: constant-transformation One.resid A t
    using tu A.con-implies-arr
    by unfold-locales blast
  interpret U: constant-transformation One.resid A u
    using tu A.con-implies-arr
    by unfold-locales blast
  interpret TU: constant-transformation One.resid A ‹A t u›
    using tu
    by unfold-locales auto
  have 2: T.F = U.F
    using tu A.con-implies-arr A.con-imp-eq-src by auto
  have 3: TU.F = U.G
    using tu by auto
  show 1: Up t ⌢ Up u
  proof
    show coinitial (Up t) (Up u)
      using tu 2 A.con-implies-arr sources-char src-char
            T.transformation-axioms U.transformation-axioms
      by (intro coinitialI) auto
    show ⋀a. One.ide a ⟹ Map (Up t) a ⌢A Map (Up u) a
      by (simp add: T.arr-t U.arr-t tu)
  qed
  show Up (t \A u) = Up t \ Up u
  proof (intro arr-eqI)
    show arr (Up (t \A u))
      using tu TU.transformation-axioms by simp
```

```
    show arr (Up t \ Up u)
      using 1 by auto
    show Dom (Up (t \_A u)) = Dom (Up t \ Up u)
      using tu 2 3 A.con-implies-arr resid-def
            T.transformation-axioms U.transformation-axioms
      by simp
    show Cod (Up (t \_A u)) = Cod (Up t \ Up u)
    proof -
      have TU.G =
           (λt. if One.arr t
                then Cod (MkArr U.F U.G U.map) t \_A
                       (Map (MkArr U.F T.G T.map) (One.src t) \_A
                          Map (MkArr U.F U.G U.map) (One.src t))
                else A.null)
        using A.apex-sym A.cube A.trg-def by auto
      thus ?thesis
        using tu 2 3 A.con-implies-arr resid-def
              T.transformation-axioms U.transformation-axioms
        by simp
    qed
    show ⋀a. One.ide a ⟹
                Map (Up (t \_A u)) a = Map (Up t \ Up u) a
      using tu 2 A.con-implies-arr resid-def
            T.transformation-axioms U.transformation-axioms
      apply simp
      by (metis 3 A.join-src A.join-sym TU.arr-t One.arr-char)
  qed
qed

sublocale Dn: simulation resid A Dn
proof
  show ⋀t. ¬ arr t ⟹ Dn t = A.null
    by simp
  fix t u
  assume tu: t ⌢ u
  interpret T: transformation One.resid A ‹Dom t› ‹Cod t› ‹Map t›
    using tu con-implies-arr arr-char by blast
  interpret U: transformation One.resid A ‹Dom u› ‹Cod u› ‹Map u›
    using tu con-implies-arr arr-char by blast
  interpret TU: transformation One.resid A ‹Cod u› ‹Apex t u› ‹Resid t u›
    using tu transformation-Map-resid [of t u] resid-def [of t u] con-char
    by auto
  show 1: Dn t ⌢_A Dn u
    using tu con-implies-arr con-char One.ide-char_1RTS by auto
  have 2: Dom t = Dom u
    using tu con-char by auto
  show Dn (t \ u) = Dn t \_A Dn u
    using tu 1 2 con-implies-arr resid-def One.ide-char_1RTS One.arr-char
      null-char not-arr-null T.transformation-axioms U.transformation-axioms
```

```
        One.src-char A.null-is-zero A.not-arr-null TU.transformation-axioms
        Apex-def
    apply auto[1]
    by (metis (no-types, lifting) A.join-src A.join-sym A.residuation-axioms
      A.src-resid U.preserves-trg residuation.arr-resid)
qed

lemma inverse-simulations-Dn-Up:
shows inverse-simulations A resid Dn Up
proof
  show Dn ∘ Up = I A
    using One.arr-char by auto
  show Up ∘ Dn = I resid
  proof
    interpret UpoDown: composite-simulation resid A resid Dn Up ..
    fix t
    show (Up ∘ Dn) t = I resid t
    proof (cases arr t, intro arr-eqI)
      show ¬ arr t ⟹ (Up ∘ Dn) t = I resid t
        by auto
      show arr t ⟹ arr (UpoDown.map t) by blast
      show arr t ⟹ arr (I resid t) by simp
      fix t
      assume t: arr t
      interpret T: transformation One.resid A ‹Dom t› ‹Cod t› ‹Map t›
        using t arr-char [of t] by blast
      show Dom (UpoDown.map t) = Dom (I resid t)
       using t T.F.extensional T.preserves-arr One.arr-char One.ide-char1RTS
             T.preserves-src
        by auto
      show Cod (UpoDown.map t) = Cod (I resid t)
       using t T.G.extensional T.preserves-arr One.arr-char One.ide-char1RTS
             T.preserves-trg
        by auto
      show ⋀a. One.ide a ⟹ Map (UpoDown.map t) a = Map (I resid t) a
        using t T.preserves-arr One.arr-char by auto
    qed
  qed
qed

end
```

3.10.4 Evaluation Map

```
locale evaluation-map =
  A: weakly-extensional-rts A +
  B: extensional-rts B
for A :: 'a resid          (infix \A 55)
and B :: 'b resid          (infix \B 55)
```

begin

sublocale *AB*: *exponential-rts A B* **..**
sublocale *ABxA*: *product-rts AB.resid A* **..**

notation *AB.resid* (**infix** $\backslash_{[A,B]}$ *55*)
notation *ABxA.resid* (**infix** $\backslash_{[A,B]xA}$ *55*)
notation *AB.con* (**infix** $\frown_{[A,B]}$ *50*)
notation *ABxA.con* (**infix** $\frown_{[A,B]\times A}$ *50*)

definition *map* :: (*'a*, *'b*) *AB.arr* × *'a* ⇒ *'b*
where *map Fg* ≡ *if ABxA.arr Fg then AB.Map* (*fst Fg*) (*snd Fg*) *else B.null*

lemma *map-simp*:
assumes *ABxA.arr Fg*
shows *map Fg* = *AB.Map* (*fst Fg*) (*snd Fg*)
using *assms map-def* **by** *auto*

lemma *is-simulation*:
shows *simulation ABxA.resid B map*
proof
show ⋀*Fg*. ¬ *ABxA.arr Fg* ⟹ *map Fg* = *B.null*
using *map-def* **by** *auto*
fix *Fg Fg'*
assume *con*: *Fg* $\frown_{[A,B]\times A}$ *Fg'*
let *?F* = *fst Fg* **and** *?g* = *snd Fg*
let *?F'* = *fst Fg'* **and** *?g'* = *snd Fg'*
have *con-FF'*: *?F* $\frown_{[A,B]}$ *?F'*
using *con* **by** *blast*
have *con-gg'*: *?g* $\frown_A$ *?g'*
using *con* **by** *blast*
interpret *F*: *transformation A B* ‹*AB.Dom ?F*› ‹*AB.Cod ?F*› ‹*AB.Map ?F*›
using *AB.con-char con-FF'* **by** *auto*
interpret *F'*: *transformation A B*
‹*AB.Dom ?F*› ‹*AB.Cod ?F'*› ‹*AB.Map ?F'*›
using *AB.con-char con-FF'* **by** *metis*
interpret *F-F'*: *transformation A B* ‹*AB.Cod ?F'*› ‹*AB.Apex ?F ?F'*›
‹*AB.Map* (*AB.resid ?F ?F'*)›
using *AB.con-char AB.transformation-Map-resid con-FF'* **by** *auto*
show *map Fg* $\frown_B$ *map Fg'*
by (*metis A.arr-resid AB.resid-Map ABxA.con-implies-arr*(*2*) *B.conI*
B.not-arr-null F-F'.preserves-arr con con-FF' con-gg' map-def
ABxA.con-implies-arr(*1*))
show *map* (*Fg* $\backslash_{[A,B]xA}$ *Fg'*) = *map Fg* $\backslash_B$ *map Fg'*
by (*metis AB.resid-Map ABxA.arr-resid ABxA.con-implies-arr*(*2*)
ABxA.resid-def con con-FF' con-gg' fst-conv map-def
ABxA.con-implies-arr(*1*) *snd-conv*)
qed

sublocale *simulation ABxA.resid B map*
using *is-simulation* **by** *auto*
sublocale *binary-simulation AB.resid A B map* **..**

lemma *src-map*:
assumes *AB.arr Fg* **and** *A.arr f*
shows *B.src* (*map* (*Fg*, *f*)) = *AB.Dom Fg* (*A.src f*)
proof −
interpret *F*: *transformation A B* ‹*AB.Dom Fg*› ‹*AB.Cod Fg*› ‹*AB.Map Fg*›
using *assms AB.arr-char* **by** *auto*
show *?thesis*
by (*metis A.ide-src ABxA.arr-char B.src-join-of*(*1*) *F.naturality3*
F.preserves-src assms(*1*−*2*) *fst-conv map-def snd-conv*)
qed

lemma *trg-map*:
assumes *AB.arr Fg* **and** *A.arr f*
shows *B.trg* (*map* (*Fg*, *f*)) = *AB.Cod Fg* (*A.trg f*)
proof −
interpret *F*: *transformation A B* ‹*AB.Dom Fg*› ‹*AB.Cod Fg*› ‹*AB.Map Fg*›
using *assms AB.arr-char* **by** *auto*
show *?thesis*
by (*metis ABxA.arr-char B.apex-sym B.trg-join-of*(*1*) *F.G.preserves-trg*
F.naturality2-ax F.naturality3 assms(*1*−*2*) *fst-conv map-def snd-conv*)
qed

end

3.10.5 Currying

locale *Currying* =
A: *weakly-extensional-rts A* +
B: *weakly-extensional-rts B* +
C: *extensional-rts C*
for *A* :: *'a resid* (**infix** $\backslash_A$ *55*)
and *B* :: *'b resid* (**infix** $\backslash_B$ *55*)
and *C* :: *'c resid* (**infix** $\backslash_C$ *55*)
begin

sublocale *AxB*: *product-of-weakly-extensional-rts A B* **..**
sublocale *BC*: *exponential-rts B C* **..**
sublocale *BCxB*: *product-rts BC.resid B* **..**
sublocale *E*: *evaluation-map B C* **..**

notation *A.con* (**infix** $\frown_A$ *50*)
notation *B.con* (**infix** $\frown_B$ *50*)
notation *C.con* (**infix** $\frown_C$ *50*)
notation *C.prfx* (**infix** $\lesssim_C$ *50*)
notation *C.join* (**infixr** $\sqcup_C$ *52*)

notation *AxB.resid* (**infix** $\backslash_{AxB}$ *55*)
notation *AxB.con* (**infix** $\frown_{AxB}$ *52*)
notation *AxB.prfx* (**infix** $\lesssim_{AxB}$ *52*)
notation *BC.resid* (**infix** $\backslash_{[B,C]}$ *55*)
notation *BC.con* (**infix** $\frown_{[B,C]}$ *52*)
notation *BC.join* (**infixr** $\sqcup_{[B,C]}$ *52*)
notation *BCxB.resid* (**infix** $\backslash_{[B,C]xB}$ *55*)
notation *BCxB.con* (**infix** $\frown_{[B,C]xB}$ *52*)

definition *Curry* :: $('a \times 'b \Rightarrow 'c) \Rightarrow ('a \times 'b \Rightarrow 'c) \Rightarrow ('a \times 'b \Rightarrow 'c)$
$\Rightarrow 'a \Rightarrow ('b, 'c)$ *BC.arr*
where *Curry F G τ f* =
(*if A.arr f*
then BC.MkArr ($\lambda g.$ *F* (*A.src f*, *g*)) ($\lambda g.$ *G* (*A.trg f*, *g*)) ($\lambda g.$ *τ* (*f*, *g*))
else BC.null)

abbreviation *Curry3* :: $('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow ('b, 'c)$ *BC.arr*
where *Curry3 F* ≡ *Curry F F F*

definition *Uncurry* :: $('a \Rightarrow ('b, 'c)$ *BC.arr*$) \Rightarrow 'a \times 'b \Rightarrow 'c$
where *Uncurry τ f* ≡ *if AxB.arr f then E.map* (*τ* (*fst f*), *snd f*) *else C.null*

lemma *Curry-simp*:
assumes *A.arr f*
shows *Curry F G τ f* =
BC.MkArr ($\lambda g.$ *F* (*A.src f*, *g*)) ($\lambda g.$ *G* (*A.trg f*, *g*)) ($\lambda g.$ *τ* (*f*, *g*))
using *assms Curry-def* **by** *auto*

lemma *Uncurry-simp*:
assumes *AxB.arr f*
shows *Uncurry τ f* = *E.map* (*τ* (*fst f*), *snd f*)
using *assms Uncurry-def* **by** *auto*

lemma *Dom-Curry*:
assumes *A.arr f*
shows *BC.Dom* (*Curry F G τ f*) = ($\lambda g.$ *F* (*A.src f*, *g*))
using *assms Curry-simp* **by** *simp*

lemma *Cod-Curry*:
assumes *A.arr f*
shows *BC.Cod* (*Curry F G τ f*) = ($\lambda g.$ *G* (*A.trg f*, *g*))
using *assms Curry-simp* **by** *simp*

lemma *Map-Curry*:
assumes *A.arr f*
shows *BC.Map* (*Curry F G τ f*) = ($\lambda g.$ *τ* (*f*, *g*))
using *assms Curry-simp* **by** *simp*

lemma *Map-simulation-expansion*:

assumes *simulation A BC.resid G* **and** *AxB.arr f*
shows *BC.Map (G (fst f)) (snd f) =*
BC.Map (G (fst f)) (B.src (snd f)) ⊔C
BC.Map (G (A.src (fst f))) (snd f)
proof −
interpret *G*: *simulation A BC.resid G*
using *assms(1)* **by** *blast*
interpret *G*: *simulation-to-extensional-rts A BC.resid G* **..**
show *?thesis*
proof (*intro C.join-eqI′*)
show *BC.Map (G (fst f)) (B.src (snd f)) ≾C BC.Map (G (fst f)) (snd f)*
by (*metis B.con-arr-src(2) B.resid-src-arr B.trg-def*
BC.Map-preserves-prfx BC.cong-reflexive BC.resid-Map-self
G.preserves-reflects-arr assms(2) AxB.arr-char B.arrE)
show *BC.Map (G (A.src (fst f))) (snd f) ≾C BC.Map (G (fst f)) (snd f)*
by (*meson A.source-is-prfx A.src-in-sources BC.Map-preserves-prfx*
G.preserves-prfx assms(2) AxB.arr-char)
show *BC.Map (G (fst f)) (snd f) \C BC.Map (G (A.src (fst f))) (snd f) =*
BC.Map (G (fst f)) (B.src (snd f)) \C
BC.Map (G (A.src (fst f))) (snd f)
by (*metis A.con-arr-src(1) AxB.arr-char AxB.con-char B.con-arr-src(2)*
B.resid-src-arr B.trg-def BC.resid-Map G.preserves-con assms(2)
AxB.arrE)
show *BC.Map (G (fst f)) (snd f) \C BC.Map (G (fst f)) (B.src (snd f)) =*
BC.Map (G (A.src (fst f))) (snd f) \C
BC.Map (G (fst f)) (B.src (snd f))
by (*metis A.con-arr-src(2) A.resid-arr-self A.resid-src-arr*
B.con-arr-src(1) BC.resid-Map G.preserves-con
G.preserves-resid assms(2) AxB.arr-char A.arrE)
qed
qed

lemma *Map-simulation-monotone*:
assumes *simulation A BC.resid G* **and** *f ≾AxB g*
shows *BC.Map (G (fst f)) (snd f) ≾C BC.Map (G (fst g)) (snd g)*
proof −
interpret *G*: *simulation A BC.resid G*
using *assms(1)* **by** *blast*
interpret *G*: *simulation-to-extensional-rts A BC.resid G* **..**
show *?thesis*
using *assms(2)*
by (*metis AxB.con-char AxB.prfx-char AxB.prfx-implies-con*
BC.Map-resid-ide BC.arrE BC.con-def BC.ide-implies-arr
BC.prfx-char BC.resid-Map G.preserves-prfx)
qed

lemma *Curry-preserves-simulations* [*intro*]:
assumes *simulation AxB.resid C F*
shows *simulation A BC.resid (Curry3 F)*

```
proof −
  interpret F: simulation AxB.resid C F
    using assms by auto
  interpret F: binary-simulation-between-weakly-extensional-rts A B C F ..
  show ?thesis
  proof
    show ⋀t. ¬ A.arr t ⟹ Curry3 F t = BC.null
      using Curry-def by simp
    fix t u
    assume con: t ⌢A u
    interpret Ft: transformation B C ‹λg. F (A.src u, g)› ‹λg. F (A.trg t, g)›
                  ‹λg. F (t, g)›
      using con F.fixing-arr-gives-transformation-1 [of t] A.con-implies-arr(1)
            A.con-imp-eq-src
      by simp
    interpret Fu: transformation B C ‹λg. F (A.src u, g)› ‹λg. F (A.trg u, g)›
                  ‹λg. F (u, g)›
      using con F.fixing-arr-gives-transformation-1 [of u] A.con-implies-arr(2)
            A.con-imp-eq-src
      by simp
    show ∗: Curry3 F t ⌢[B,C] Curry3 F u
      using con Curry-def BC.con-char Curry-simp A.con-implies-arr
            A.con-imp-eq-src Ft.transformation-axioms Fu.transformation-axioms
      by simp
    show Curry3 F (t \A u) = Curry3 F t \[B,C] Curry3 F u
    proof −
      have BC.Dom (Curry3 F (t \A u)) =
           BC.Dom (Curry3 F t \[B,C] Curry3 F u)
        using ∗ A.arr-resid A.con-implies-arr(2) A.src-resid BC.Dom-resid
              Cod-Curry Dom-Curry con
        by presburger
      moreover have BC.Cod (Curry3 F (t \A u)) =
                    BC.Cod (Curry3 F t \[B,C] Curry3 F u)
      proof −
        have BC.Cod (Curry3 F (t \A u)) = (λg. F (A.trg (t \A u), g))
          using Cod-Curry con by force
        also have ... = BC.Apex (BC.MkArr (λg. F (A.src u, g))
                                  (λg. F (A.trg t, g)) (λg. F (t, g)))
                                (BC.MkArr (λg. F (A.src u, g))
                                  (λg. F (A.trg u, g)) (λg. F (u, g)))
        proof −
          have (λg. F (A.trg (t \A u), g)) =
               (λg. if B.arr g
                    then BC.Cod (BC.MkArr (λg. F (A.src u, g))
                                          (λg. F (A.trg u, g)) (λg. F (u, g))) g \C
                         (BC.Map (BC.MkArr
                                    (λg. F (A.src u, g)) (λg. F (A.trg t, g))
                                    (λg. F (t, g)))
                                 (B.src g) \C
```

```
                    BC.Map (BC.MkArr
                              (λg. F (A.src u, g)) (λg. F (A.trg u, g))
                              (λg. F (u, g)))
                      (B.src g))
          else C.null)
  proof
    fix g
    show F (A.trg (t \A u), g) =
         (if B.arr g
          then BC.Cod (BC.MkArr
                         (λg. F (A.src u, g)) (λg. F (A.trg u, g))
                         (λg. F (u, g))) g \C
               (BC.Map (BC.MkArr
                          (λg. F (A.src u, g)) (λg. F (A.trg t, g))
                          (λg. F (t, g)))
                 (B.src g) \C
                 BC.Map (BC.MkArr (λg. F (A.src u, g))
                                  (λg. F (A.trg u, g)) (λg. F (u, g)))
                   (B.src g))
          else C.null)
    proof (cases B.arr g)
      show ¬ B.arr g ⟹ ?thesis
        by (metis A.arr-resid A.ide-trg F.fixing-ide-gives-simulation-1 con
            simulation.extensional)
      assume g: B.arr g
      have F (t, B.src g) \C F (u, B.src g) = F (t \A u, B.src g)
        using g con F.preserves-resid AxB.resid-def AxB.con-char
        by (metis AxB.con-arr-self B.arr-src-iff-arr B.trg-def B.trg-src
            F.preserves-reflects-arr Ft.preserves-arr fst-conv snd-conv)
      moreover have F (A.trg (t \A u), g) =
                    F (A.trg u, g) \C F (t \A u, B.src g)
        using con g F.preserves-resid AxB.resid-def AxB.con-char
        by auto
          (metis (no-types, lifting) A.arr-resid A.con-imp-arr-resid
           A.resid-src-arr A.src-resid A.trg-def B.con-arr-src(1)
           B.resid-arr-src A.con-def)
      ultimately show ?thesis
        using con BC.Apex-def A.con-implies-arr F.extensional
              F.preserves-trg F.preserves-resid AxB.resid-def
        by auto
    qed
  qed
  thus ?thesis
  using con BC.Apex-def A.con-implies-arr F.extensional F.preserves-trg
    by auto
qed
also have ... = BC.Cod (Curry3 F t \[B,C] Curry3 F u)
  using con Curry-def A.con-implies-arr A.con-imp-eq-src BC.Apex-def
        Ft.transformation-axioms Fu.transformation-axioms BC.resid-def
```

```
    by simp
  finally show ?thesis by blast
qed
moreover have BC.Map (Curry3 F (t \<setminus>A u)) =
              BC.Map (Curry3 F t \<setminus>[B,C] Curry3 F u)
proof −
  have BC.Map (Curry3 F (t \<setminus>A u)) = (λg. F (t \<setminus>A u, g))
    using con Curry-def A.con-implies-arr A.con-imp-eq-src BC.Map-resid
    by simp
  also have ... =
            (λg. if B.arr g
                 then BC.Map (BC.MkArr (λg. F (A.src u, g))
                                       (λg. F (A.trg t, g))
                                       (λg. F (t, g)))
                             (B.src g) \<setminus>C
                      BC.Map (BC.MkArr (λg. F (A.src u, g))
                                       (λg. F (A.trg u, g))
                                       (λg. F (u, g)))
                             (B.src g) ⊔C
                      BC.Cod (BC.MkArr (λg. F (A.src u, g))
                                       (λg. F (A.trg u, g))
                                       (λg. F (u, g)))
                             g
                 else C.null)
  proof
    fix g
    show F (t \<setminus>A u, g) =
         (if B.arr g
          then BC.Map (BC.MkArr (λg. F (A.src u, g))
                                (λg. F (A.trg t, g))
                                (λg. F (t, g)))
                      (B.src g) \<setminus>C
               BC.Map (BC.MkArr (λg. F (A.src u, g))
                                (λg. F (A.trg u, g))
                                (λg. F (u, g)))
                      (B.src g) ⊔C
               BC.Cod (BC.MkArr (λg. F (A.src u, g))
                                (λg. F (A.trg u, g))
                                (λg. F (u, g)))
                      g
          else C.null)
    proof (cases B.arr g)
      show ¬ B.arr g ⟹ ?thesis
        using F.extensional by simp
      assume g: B.arr g
      have F (t \<setminus>A u, g) =
           F (t, B.src g) \<setminus>C F (u, B.src g) ⊔C F (A.trg u, g)
      proof −
        have F (t, B.src g) \<setminus>C F (u, B.src g) = F (t \<setminus>A u, B.src g)
```

```
            by (metis (no-types, lifting) AxB.con-char AxB.resid-def
                B.ide-def B.ide-src F.preserves-resid con g fst-eqD snd-eqD)
          moreover have F (t \<^sub>A u, B.src g) ⊔C F (A.trg u, g) =
                        F (t \<^sub>A u, g)
          proof -
            have C.join-of (F (t \<^sub>A u, B.src g)) (F (A.trg u, g))
                   (F (t \<^sub>A u, g))
              using F.preserves-joins BCxB.join-of-char
              by (metis A.arr-resid A.join-of-arr-src(2) A.src-in-sources
                  A.src-resid AxB.join-of-char(1) B.join-of-arr-src(1)
                  B.src-in-sources con g fst-eqD snd-conv)
            thus ?thesis
              by (meson C.join-is-join-of C.join-of-unique C.joinable-def)
          qed
          ultimately show ?thesis by auto
        qed
        thus ?thesis
          using g by simp
      qed
    qed
    finally show ?thesis
     using con Curry-def A.con-implies-arr A.con-imp-eq-src BC.Map-resid'
           Ft.transformation-axioms Fu.transformation-axioms
      by simp
  qed
  moreover have Curry3 F (t \<^sub>A u) ≠ BC.Null
    using BC.arr-char
    by (simp add: con Curry-def)
  moreover have Curry3 F t \<^sub>[B,C] Curry3 F u ≠ BC.Null
    using con BC.arr-char * BC.null-char BC.con-def by force
  moreover
  have ⋀x y. BC.Dom x = BC.Dom y ∧ BC.Cod x = BC.Cod y ∧
             BC.Map x = BC.Map y ∧ x ≠ BC.Null ∧ y ≠ BC.Null
               ⟹ x = y
    by (metis BC.Cod.simps(1) BC.Dom.simps(1) BC.Map.elims)
  ultimately show ?thesis by blast
 qed
qed
qed

lemma Uncurry-preserves-simulations [intro]:
assumes simulation A BC.resid F
shows simulation AxB.resid C (Uncurry F)
proof -
  interpret F: simulation A BC.resid F using assms by auto
  show ?thesis
  proof
    show ⋀t. ¬ AxB.arr t ⟹ Uncurry F t = C.null
      using Uncurry-def by presburger
```

```
    show ⋀t u. t ⌢_AxB u ⟹ Uncurry F t ⌢_C Uncurry F u
      using Uncurry-def AxB.con-implies-arr(1) AxB.con-implies-arr(2)
      by auto
    fix t u :: 'a * 'b
    assume con: t ⌢_AxB u
    have Uncurry F (t \_AxB u) = E.map (F (fst (t \_AxB u)), snd (t \_AxB u))
      using AxB.arr-resid Uncurry-def con by presburger
    also have ... = Uncurry F t \_C Uncurry F u
      using AxB.con-implies-arr(1) AxB.con-implies-arr(2) AxB.resid-def
        BCxB.resid-def E.preserves-resid Uncurry-simp con
      by auto
    finally show Uncurry F (t \_AxB u) = Uncurry F t \_C Uncurry F u
      by auto
  qed
qed

lemma Curry-preserves-transformations:
assumes transformation AxB.resid C F G τ
shows transformation A BC.resid (Curry3 F) (Curry3 G) (Curry F G τ)
proof −
  interpret τ: transformation AxB.resid C F G τ using assms by auto
  interpret τ: transformation-to-extensional-rts AxB.resid C F G τ ..
  interpret τ: transformation-of-binary-simulations A B C F G τ ..
  interpret Curry-F: simulation A BC.resid ‹Curry F F F›
    using Curry-preserves-simulations τ.F.simulation-axioms by simp
  interpret Curry-G: simulation A BC.resid ‹Curry G G G›
    using Curry-preserves-simulations τ.G.simulation-axioms by simp
  show ?thesis
  proof
    fix f
    show ¬ A.arr f ⟹ Curry F G τ f = BC.null
      using Curry-def by simp
    show A.ide f ⟹ BC.src (Curry F G τ f) = Curry3 F f
      unfolding Curry-def
     using BC.src-simp A.ide-implies-arr A.src-ide A.trg-ide BC.Dom.simps(1)
        BC.arr-MkArr τ.fixing-ide-gives-transformation-1
      by presburger
    show A.ide f ⟹ BC.trg (Curry F G τ f) = Curry3 G f
      unfolding Curry-def
     using BC.trg-simp A.ide-implies-arr A.src-ide A.trg-ide BC.Cod.simps(1)
        BC.arr-MkArr τ.fixing-ide-gives-transformation-1
      by presburger
    assume f: A.arr f
    interpret τ-src: transformation B C
                  ‹λg. F (A.src f, g)› ‹λg. G (A.src f, g)› ‹λg. τ (A.src f, g)›
      using f τ.fixing-ide-gives-transformation-1 by auto
    interpret τ-trg: transformation B C
                  ‹λg. F (A.trg f, g)› ‹λg. G (A.trg f, g)› ‹λg. τ (A.trg f, g)›
      using f τ.fixing-ide-gives-transformation-1 by auto
```

```
interpret τ.F: binary-simulation-between-weakly-extensional-rts A B C F ..
interpret F-f: transformation B C
                ‹λg. F (A.src f, g)› ‹λg. F (A.trg f, g)› ‹λg. F (f, g)›
  using f τ.F.fixing-arr-gives-transformation-1 by metis
show A: Curry F G τ (A.src f) \[B,C] Curry3 F f = Curry F G τ (A.trg f)
proof −
  have Curry F G τ (A.src f) \[B,C] Curry3 F f =
       BC.MkArr (λg. F (A.src f, g)) (λg. G (A.src f, g))
          (λg. τ (A.src f, g)) \[B,C]
        BC.MkArr (λg. F (A.src f, g)) (λg. F (A.trg f, g)) (λg. F (f, g))
    using Curry-def f by simp
  also have ... = BC.MkArr (λg. F (A.trg f, g)) (λg. G (A.trg f, g))
                     (λg. τ (A.trg f, g))
    (is ?LHS1 \[B,C] ?LHS2 = ?RHS)
  proof −
    have 1: ?LHS1 ⌢[B,C] ?LHS2
      using f BC.con-char τ-src.transformation-axioms
            F-f.transformation-axioms
      apply simp
      by (metis B.ide-iff-src-self B.ide-iff-trg-self C.conE C.conI
          C.con-arr-src(1) τ.fixing-ide-gives-transformation-0
          τ-trg.naturality1 τ-trg.preserves-arr τ-trg.preserves-src
          B.ide-implies-arr transformation.naturality1)
    have BC.Dom (?LHS1 \[B,C] ?LHS2) = BC.Dom ?RHS
      using 1 f BC.Dom-resid BC.con-char by simp
    moreover have BC.Cod (?LHS1 \[B,C] ?LHS2) = BC.Cod ?RHS
    proof −
      have BC.Cod (?LHS1 \[B,C] ?LHS2) = BC.Apex ?LHS1 ?LHS2
        using 1 f BC.Cod-resid BC.con-char by fastforce
      also have ... = (λg. G (A.trg f, g))
      proof
        fix g
        show BC.Apex ?LHS1 ?LHS2 g = G (A.trg f, g)
          unfolding BC.Apex-def
          using f
          by (metis B.ide-src BC.Cod.simps(1) BC.Map.simps(1)
              τ.fixing-ide-gives-transformation-0 τ-trg.G.extensional
              τ-trg.naturality2 transformation.naturality1-ax)
      qed
      also have ... = BC.Cod ?RHS
        using 1 f BC.Cod-resid BC.con-char by fastforce
      finally show ?thesis by blast
    qed
    moreover have BC.Map (?LHS1 \[B,C] ?LHS2) = BC.Map ?RHS
    proof
      fix g
      have τ (A.src f, B.src g) \C F (f, B.src g) ⊔C F (A.trg f, g) =
           τ (A.trg f, g)
      proof −
```

```
        have τ (A.src f, B.src g) \C F (f, B.src g) ⊔C F (A.trg f, g) =
             τ (A.trg f, B.src g) ⊔C F (A.trg f, g)
          by (metis AxB.src-char AxB.trg-char B.src-src B.trg-src
             C.null-is-zero(2) F-f.extensional F-f.preserves-arr
             τ.F.preserves-reflects-arr τ.naturality1 τ-trg.extensional
             fst-conv snd-conv)
        also have ... = τ (A.trg f, g)
          by (metis B.arr-src-iff-arr C.arr-prfx-join-self C.join-def
             C.join-is-join-of C.join-of-unique C.joinable-def
             C.not-ide-null C.null-is-zero(1) τ-trg.extensional
             τ-trg.naturality3)
        finally show ?thesis by blast
      qed
      moreover have ¬ B.arr g ⟹ C.null = τ (A.trg f, g)
        using τ-trg.extensional by presburger
      ultimately show BC.Map (?LHS1 \[B,C] ?LHS2) g = BC.Map ?RHS
g
        using 1
        by (cases B.arr g, simp-all)
      qed
      ultimately show ?thesis
        using 1 BC.con-char BC.resid-def by auto
    qed
    also have ... = Curry F G τ (A.trg f)
      using Curry-def f by simp
    finally show ?thesis by auto
  qed
  show Curry3 F f \[B,C] Curry F G τ (A.src f) = Curry3 G f
  proof −
    have Curry3 F f \[B,C] Curry F G τ (A.src f) =
         BC.MkArr (λg. F (A.src f, g)) (λg. F (A.trg f, g))
           (λg. F (f, g)) \[B,C]
          BC.MkArr (λg. F (A.src f, g)) (λg. G (A.src f, g))
            (λg. τ (A.src f, g))
      unfolding Curry-def
      using f by simp
    also have ... = BC.MkArr (λg. G (A.src f, g)) (λg. G (A.trg f, g))
                      (λg. G (f, g))
      (is ?LHS1 \[B,C] ?LHS2 = ?RHS)
    proof −
      have 1: BC.con ?LHS1 ?LHS2
        using A f BC.con-sym-ax BC.null-char Curry-simp by auto
      have BC.Dom (?LHS1 \[B,C] ?LHS2) = BC.Dom ?RHS
        using 1 f BC.Dom-resid BC.con-char by auto
      moreover have BC.Cod (?LHS1 \[B,C] ?LHS2) = BC.Cod ?RHS
      proof −
        have BC.Cod (?LHS1 \[B,C] ?LHS2) = BC.Apex ?LHS1 ?LHS2
          using 1 f BC.Dom-resid BC.con-char by fastforce
        also have ... = (λg. G (A.trg f, g))
```

```
  proof
    fix g
    have G (A.src f, g) \_C (F (f, B.src g) \_C τ (A.src f, B.src g)) =
         G (A.trg f, g)
    proof −
      have G (A.src f, g) \_C (F (f, B.src g) \_C τ (A.src f, B.src g)) =
           G (A.src f, g) \_C G (f, B.src g)
        by (metis AxB.src-char B.arr-src-iff-arr B.src-src C.con-sym-ax
            C.null-is-zero(2) F-f.preserves-arr τ.F.preserves-reflects-arr
            τ.naturality2 τ-src.G.extensional fst-conv snd-conv)
      also have ... = G ((A.src f, g) \_AxB (f, B.src g))
        by (metis A.con-arr-src(2) AxB.con-char B.arr-src-iff-arr
            B.con-arr-src(1) C.null-is-zero(2) τ.G.extensional
            τ.G.preserves-resid τ-src.extensional calculation f
            AxB.arr-resid-iff-con fst-conv snd-conv)
      also have ... = G (A.src f \_A f, g \_B B.src g)
        using AxB.resid-def B.arr-resid-iff-con τ.G.extensional f
        by auto
      also have ... = G (A.trg f, g)
        by (metis A.resid-src-arr B.resid-arr-src B.resid-reflects-con
            τ-trg.G.extensional f B.arr-def B.arr-resid-iff-con)
      finally show ?thesis by blast
    qed
    moreover have ¬ B.arr g ⟹ C.null = G (A.trg f, g)
      using τ-trg.G.extensional by presburger
    ultimately show BC.Apex ?LHS1 ?LHS2 g = G (A.trg f, g)
      using BC.Apex-def f
      by (cases B.arr g, simp-all)
  qed
  also have ... = BC.Cod ?RHS
    using 1 f by fastforce
  finally show ?thesis by blast
qed
moreover have BC.Map (?LHS1 \_[B,C] ?LHS2) = BC.Map ?RHS
proof
  fix g
  show BC.Map (?LHS1 \_[B,C] ?LHS2) g = BC.Map ?RHS g
  proof −
    have F (f, B.src g) \_C τ (A.src f, B.src g) ⊔_C G (A.src f, g) =
         G (f, g)
    proof (cases B.arr g)
      show ¬ B.arr g ⟹ ?thesis
        by (metis B.arr-src-iff-arr C.arr-prfx-join-self
            C.joinable-iff-join-not-null C.not-ide-null C.null-is-zero(1)
            F-f.extensional τ.naturality2)
      assume g: B.arr g
      have F (f, B.src g) \_C τ (A.src f, B.src g) ⊔_C G (A.src f, g) =
           G (f, B.src g) ⊔_C G (A.src f, g)
      by (metis AxB.src-char B.arr-src-iff-arr B.src-src F-f.preserves-arr
```

```
            τ.F.preserves-reflects-arr τ.naturality2 g fst-conv snd-conv)
        also have ... = G (f, g)
          using f g
                τ.G.preserves-joins
                  [of (f, B.src g) (A.src f, g) (f, g)]
        by (metis A.join-of-arr-src(2) A.src-in-sources AxB.join-of-char(1)
              B.join-of-arr-src(1) B.src-in-sources C.join-is-join-of
              C.join-of-unique C.joinable-def fst-conv snd-conv)
        finally show ?thesis by blast
      qed
      thus ?thesis
        using 1 f BC.Map-resid BC.con-char τ.G.extensional by auto
    qed
  qed
  ultimately show ?thesis
    using 1 BC.con-char BC.resid-def by auto
qed
also have ... = Curry3 G f
  using Curry-def f by simp
finally show ?thesis by blast
qed
show BC.join-of (Curry F G τ (A.src f)) (Curry3 F f) (Curry F G τ f)
proof −
  have ∗: ⋀t. B.arr t ⟹
              (τ (A.src f, B.src t) ⊔C F (f, B.src t)) ⊔C F (A.src f, t) =
              τ (A.src f, B.src t) ⊔C F (f, t)
  proof −
    fix t
    assume t: B.arr t
    have 1: C.joinable (τ (A.src f, B.src t)) (F (f, t))
   by (metis AxB.product-of-weakly-extensional-rts-axioms F-f.preserves-arr
        τ.F.preserves-reflects-arr τ.naturality3'E(2) fst-conv
        product-of-weakly-extensional-rts.src-char snd-conv t)
    show (τ (A.src f, B.src t) ⊔C F (f, B.src t)) ⊔C F (A.src f, t) =
        τ (A.src f, B.src t) ⊔C F (f, t)
    proof (intro C.join-eqI)
      show 2: τ (A.src f, B.src t) ⊔C F (f, B.src t) ≲C
              τ (A.src f, B.src t) ⊔C F (f, t)
        by (metis A.ide-trg A.trg-def AxB.prfx-char AxB.src-char
            B.arr-src-iff-arr B.ide-trg B.resid-src-arr B.src-src
            F-f.preserves-arr τ.F.preserves-reflects-arr τ.naturality3'E(1)
            τ.preserves-prfx f fst-conv snd-conv t)
      show 3: F (A.src f, t) ≲C τ (A.src f, B.src t) ⊔C F (f, t)
        by (metis (full-types) 1 t C.arr-prfx-join-self C.join-is-join-of
            C.join-of-symmetric C.join-of-unique C.joinable-def
            C.prfx-transitive F-f.naturality3)
      show 4: (τ (A.src f, B.src t) ⊔C F (f, t)) \C F (A.src f, t) =
              (τ (A.src f, B.src t) ⊔C F (f, B.src t)) \C F (A.src f, t)
      proof −
```

```
  have (τ (A.src f, B.src t) ⊔C F (f, t)) \C F (A.src f, t) =
       τ (A.src f, B.src t) \C F (A.src f, t) ⊔C
         F (f, t) \C F (A.src f, t)
by (meson 1 3 C.ide-implies-arr C.resid-joinE(3) C.residuation-axioms
       residuation.arr-resid-iff-con)
  also have ... = τ (A.src f, B.src t) \C F (A.src f, t) ⊔C
                    F (f, B.src t) \C F (A.src f, t)
    by (metis C.composite-ofE C.extensional F-f.naturality1
       F-f.naturality1' t)
  also have ... = (τ (A.src f, B.src t) ⊔C F (f, B.src t)) \C
                  F (A.src f, t)
   by (metis 2 3 C.con-prfx(1) C.joinable-iff-arr-join C.resid-joinE(3)
       C.con-implies-arr(1) C.prfx-implies-con)
  finally show ?thesis by blast
qed
show (τ (A.src f, B.src t) ⊔C F (f, t)) \C
       (τ (A.src f, B.src t) ⊔C F (f, B.src t)) =
     F (A.src f, t) \C (τ (A.src f, B.src t) ⊔C F (f, B.src t))
proof −
  have (τ (A.src f, B.src t) ⊔C F (f, t)) \C
         (τ (A.src f, B.src t) ⊔C F (f, B.src t)) =
    τ (A.src f, B.src t) \C (τ (A.src f, B.src t) ⊔C F (f, B.src t)) ⊔C
      F (f, t) \C (τ (A.src f, B.src t) ⊔C F (f, B.src t))
    using 1 2 C.prfx-implies-con C.resid-joinE(3) by blast
  also have ... = C.src (F (f, t) \C
                    (τ (A.src f, B.src t) ⊔C F (f, B.src t))) ⊔C
                  F (f, t) \C (τ (A.src f, B.src t) ⊔C F (f, B.src t))
  proof −
    have C.src (F (f, t) \C (τ (A.src f, B.src t) ⊔C F (f, B.src t))) =
         C.trg (τ (A.src f, B.src t) ⊔C F (f, B.src t))
      by (metis 2 C.arr-prfx-join-self C.conI C.con-sym-ax
         C.ide-implies-arr C.join-of-symmetric C.joinable-def
         C.joinable-iff-join-not-null C.not-arr-null C.null-is-zero(1)
         C.src-resid calculation)
    also have ... = τ (A.src f, B.src t) \C
                      (τ (A.src f, B.src t) ⊔C F (f, B.src t))
      by (metis C.arr-prfx-join-self C.join-def C.null-is-zero(2)
         C.prfx-implies-con C.resid-joinE(1) C.trg-def C.trg-join)
    finally show ?thesis by argo
  qed
  also have ... = F (f, t) \C
                    (τ (A.src f, B.src t) ⊔C F (f, B.src t))
    by (metis C.arr-prfx-join-self C.arr-resid-iff-con C.conI
       C.ide-implies-arr C.join-def C.join-src C.null-is-zero(2)
       C.resid-joinE(1))
  also have ... = (F (A.src f, t) \C F (f, B.src t)) \C
                  (τ (A.src f, B.src t) \C F (f, B.src t))
  proof −
    have F (f, t) \C (τ (A.src f, B.src t) ⊔C F (f, B.src t)) =
```

```
            (F (f, t) \_C F (f, B.src t)) \_C
              (τ (A.src f, B.src t) \_C F (f, B.src t))
          by (metis 2 C.con-sym-ax C.extensional-rts-axioms C.join-def
              C.not-ide-null C.null-is-zero(2) C.residuation-axioms
              extensional-rts.resid-join_E(1) residuation.conI)
        also have ... = (F (A.src f, t) \_C F (f, B.src t)) \_C
                        (τ (A.src f, B.src t) \_C F (f, B.src t))
          by (metis C.composite-ofE C.extensional C.join-ofE
              F-f.naturality3 t)
        finally show ?thesis by blast
      qed
      also have ... = F (A.src f, t) \_C
                      (τ (A.src f, B.src t) ⊔_C F (f, B.src t))
        by (metis 3 4 C.conI C.con-sym-ax C.join-def C.not-ide-null
            C.null-is-zero(2) C.resid-join_E(1))
      finally show ?thesis by blast
    qed
  qed
qed
have 1: BC.joinable
          (BC.MkArr (λg. F (A.src f, g)) (λg. G (A.src f, g))
                    (λg. τ (A.src f, g)))
          (BC.MkArr (λg. F (A.src f, g)) (λg. F (A.trg f, g))
                    (λg. F (f, g)))
  using ∗ BC.join-char(1) C.joinable-iff-join-not-null
        F-f.preserves-arr F-f.transformation-axioms AxB.src-char
        τ.naturality3'_E(2) τ-src.transformation-axioms
  by force
moreover have BC.MkArr (λg. F (A.src f, g)) (λg. G (A.src f, g))
                       (λg. τ (A.src f, g)) ⊔_[B,C]
                BC.MkArr (λg. F (A.src f, g)) (λg. F (A.trg f, g))
                         (λg. F (f, g)) =
              BC.MkArr (λg. F (A.src f, g)) (λg. G (A.trg f, g))
                       (λg. τ (f, g))
proof −
  have BC.Apex
         (BC.MkArr (λg. F (A.src f, g)) (λg. G (A.src f, g))
                   (λg. τ (A.src f, g)))
         (BC.MkArr (λg. F (A.src f, g)) (λg. F (A.trg f, g))
                   (λg. F (f, g))) =
       (λg. G (A.trg f, g))
  proof
    fix t
    show BC.Apex
           (BC.MkArr (λg. F (A.src f, g)) (λg. G (A.src f, g))
                     (λg. τ (A.src f, g)))
           (BC.MkArr (λg. F (A.src f, g)) (λg. F (A.trg f, g))
                     (λg. F (f, g))) t =
         G (A.trg f, t)
```

```
        using f τ.naturality1 [of (f, B.src t)]
              τ.naturality2 [of (A.trg f, t)] AxB.trg-char AxB.src-char
              τ.G.extensional BC.Apex-def
        by force
    qed
    moreover
    have (λt. (τ (A.src f, B.src t) ⊔C F (f, B.src t)) ⊔C F (A.src f, t)) =
         (λg. τ (f, g))
    proof
      fix t
      have B.arr t ⟹ AxB.src (f, t) = (A.src f, B.src t)
        by (simp add: AxB.src-char f)
      thus (τ (A.src f, B.src t) ⊔C F (f, B.src t)) ⊔C F (A.src f, t) =
           τ (f, t)
        by (metis ∗ B.arr-src-iff-arr C.arr-prfx-join-self C.con-sym-ax
            C.join-def C.join-sym C.not-ide-null C.null-is-zero(2)
            F-f.extensional τ.naturality3'E(1))
    qed
    ultimately show ?thesis
      using 1 BC.join-char(2) by auto
  qed
  ultimately show ?thesis
    using 1 BC.join-is-join-of Curry-def f by fastforce
 qed
qed

lemma Uncurry-preserves-transformations:
assumes transformation A BC.resid F G τ
shows transformation AxB.resid C (Uncurry F) (Uncurry G) (Uncurry τ)
proof −
  interpret τ: transformation A BC.resid F G τ using assms by auto
  interpret τ: transformation-to-extensional-rts A BC.resid F G τ ..
  interpret Uncurry-F: simulation AxB.resid C ‹Uncurry F›
    using τ.F.simulation-axioms Uncurry-preserves-simulations by blast
  interpret Uncurry-G: simulation AxB.resid C ‹Uncurry G›
    using τ.G.simulation-axioms Uncurry-preserves-simulations by blast
  interpret Uncurry-F: binary-simulation-between-weakly-extensional-rts
                       A B C ‹Uncurry F›
    ..
  interpret Uncurry-G: binary-simulation-between-weakly-extensional-rts
                       A B C ‹Uncurry G›
    ..
  show ?thesis
  proof
    show ⋀t. ¬ AxB.arr t ⟹ Uncurry τ t = C.null
      by (meson Uncurry-def)
    show ⋀a. AxB.ide a ⟹ C.src (Uncurry τ a) = Uncurry F a
      by (metis A.ide-implies-arr AxB.ide-char AxB.ide-implies-arr
```

```
    B.ide-implies-arr B.src-ide BC.Map-src Uncurry-simp
    E.map-def E.preserves-reflects-arr E.src-map
    Uncurry-F.preserves-reflects-arr τ.preserves-arr τ.preserves-src
    fst-conv snd-conv)
show ⋀a. AxB.ide a ⟹ C.trg (Uncurry τ a) = Uncurry G a
  unfolding Uncurry-def
  using E.trg-map E.map-def BC.ide-char_ERTS τ.G.preserves-ide
  apply auto[1]
   apply (metis BC.Dom-resid BC.con-arr-src(2) BC.resid-src-arr
     τ.preserves-trg)
  by (meson A.ide-implies-arr τ.preserves-arr)
fix f
assume f: AxB.arr f
interpret F-fst: transformation B C
                 ‹BC.Map (F (A.src (fst f)))›
                 ‹BC.Map (F (A.trg (fst f)))›
                 ‹BC.Map (F (fst f))›
proof −
  have (λt2. Uncurry F (A.src (fst f), t2)) = BC.Map (F (A.src (fst f)))
  proof
    fix t2
    show Uncurry F (A.src (fst f), t2) = BC.Map (F (A.src (fst f))) t2
      using Uncurry-def E.map-def f
      apply (cases B.arr t2)
       apply auto[2]
      by (metis A.ide-src BC.ide-char_ERTS τ.F.preserves-ide
          simulation.extensional)
  qed
  moreover have (λt2. Uncurry F (A.trg (fst f), t2)) =
                BC.Map (F (A.trg (fst f)))
  proof
    fix t2
    show Uncurry F (A.trg (fst f), t2) = BC.Map (F (A.trg (fst f))) t2
      using Uncurry-def E.map-def f
      apply (cases B.arr t2)
       apply auto[2]
      by (metis A.ide-trg BC.ide-char_ERTS τ.F.preserves-ide
          simulation.extensional)
  qed
  moreover have (λt2. Uncurry F (fst f, t2)) = BC.Map (F (fst f))
  proof
    fix t2
    show Uncurry F (fst f, t2) = BC.Map (F (fst f)) t2
      using Uncurry-def E.map-def f
      apply (cases B.arr t2)
       apply auto[2]
      by (metis BC.arrE τ.F.preserves-reflects-arr
          transformation.extensional)
  qed
```

```
  ultimately
  show transformation B C
        (BC.Map (F (A.src (fst f)))) (BC.Map (F (A.trg (fst f))))
        (BC.Map (F (fst f)))
    using f Uncurry-F.fixing-arr-gives-transformation-1 [of fst f]
    by fastforce
qed
interpret τ-fst: transformation B C
                 ‹BC.Map (F (A.src (fst f)))›
                 ‹BC.Map (G (A.trg (fst f)))›
                 ‹BC.Map (τ (fst f))›
  using f τ.preserves-arr BC.arr-char
  by (metis A.ide-src BC.Map.simps(1) BC.apex-sym BC.src-simp
      BC.src-join-of(1) BC.trg-simp BC.trg-join-of(1) τ.G.preserves-trg
      τ.naturality2 τ.naturality3 τ.preserves-src AxB.arr-char)
have 1: Uncurry τ (AxB.src f) \C Uncurry τ f = Uncurry G (AxB.trg f)
proof −
  have Uncurry τ (AxB.src f) \C Uncurry τ f =
       BC.Map (τ (A.src (fst f)))
              (B.src (snd f)) \C BC.Map (τ (fst f)) (snd f)
    using Uncurry-def E.map-def f AxB.src-char AxB.trg-char
          τ.preserves-arr
    by simp
  also have X: ... = BC.Map (τ (A.src (fst f))) (B.src (snd f)) \C
                     (BC.Map (τ (fst f)) (B.src (snd f)) ⊔C
                       BC.Map (F (A.src (fst f))) (snd f))
    using f τ-fst.naturality3 C.join-is-join-of C.join-of-unique
    by (metis AxB.arr-char C.joinable-def)
  also have ... = (BC.Map (τ (A.src (fst f))) (B.src (snd f)) \C
                     BC.Map (F (A.src (fst f))) (snd f)) \C
                  (BC.Map (τ (fst f)) (B.src (snd f)) \C
                     BC.Map (F (A.src (fst f))) (snd f))
    using C.resid-join_E(1)
    by (metis (no-types, lifting) X AxB.arr-char C.conE C.conI
        C.con-sym-ax C.con-with-join-of-iff(1) C.joinable-iff-join-not-null
        C.null-is-zero(2) τ-fst.naturality3 f)
  also have ... = BC.Map (τ (A.src (fst f))) (B.trg (snd f)) \C
                  (BC.Map (τ (fst f)) (B.src (snd f)) \C
                     BC.Map (F (A.src (fst f))) (snd f))
    by (metis (no-types, lifting) A.arr-src-iff-arr A.ide-src A.src-src
        A.trg-src AxB.arr-char BC.Dom-resid BC.arrE BC.con-arr-src(1)
        BC.ide-char_ERTS τ.F.preserves-ide τ.naturality1 τ.preserves-arr
        τ.preserves-src f transformation.naturality1)
  also have ... = BC.Map (τ (A.src (fst f))) (B.trg (snd f)) \C
                  BC.Map (τ (fst f)) (B.trg (snd f))
    using f τ-fst.naturality1 by fastforce
  also have ... = BC.Map (G (A.trg (fst f))) (B.trg (snd f))
    by (metis A.ide-trg A.resid-src-arr AxB.arr-char B.arr-trg-iff-arr
        BC.Map-preserves-prfx C.apex-sym C.arr-resid-iff-con
```

```
      C.ide-iff-src-self C.prfx-implies-con C.src-resid τ.preserves-prfx
      τ-fst.G.preserves-trg τ-fst.naturality1 τ-fst.naturality2 f)
  also have ... = Uncurry G (AxB.trg f)
  using Uncurry-def E.map-def f AxB.src-char AxB.trg-char τ.preserves-arr
    by simp
  finally show ?thesis by blast
qed
have 2: Uncurry F f \C Uncurry τ f = Uncurry G (AxB.trg f)
proof −
  have Uncurry F f \C Uncurry τ f =
       BC.Map (F (fst f)) (snd f) \C BC.Map (τ (fst f)) (snd f)
  using Uncurry-def E.map-def f AxB.src-char AxB.trg-char τ.preserves-arr
    by simp
  also have ... = (BC.Map (F (fst f)) (B.src (snd f)) ⨆C
                   BC.Map (F (A.src (fst f))) (snd f)) \C
                     BC.Map (τ (fst f)) (snd f)
    by (metis AxB.arr-char C.join-is-join-of C.join-of-unique
        C.joinable-def F-fst.naturality3 f)
  also have ... = (BC.Map (F (fst f)) (B.src (snd f)) \C
                     BC.Map (τ (fst f)) (snd f)) ⨆C
                  (BC.Map (F (A.src (fst f))) (snd f) \C
                     BC.Map (τ (fst f)) (snd f))
  proof −
    have 1: C.joinable (BC.Map (F (fst f)) (B.src (snd f)))
                       (BC.Map (F (A.src (fst f))) (snd f))
      by (meson AxB.arr-char C.joinable-def F-fst.naturality3 f)
    moreover have BC.Map (τ (fst f)) (snd f) ⌢C
                  (BC.Map (F (fst f)) (B.src (snd f)) ⨆C
                     BC.Map (F (A.src (fst f))) (snd f))
    proof −
      have 2: BC.Map (F (A.src (fst f))) (snd f) ⌢C
              BC.Map (τ (fst f)) (snd f)
        using f
        by (meson AxB.arr-char C.arrE C.con-with-join-of-iff(2)
            C.join-of-symmetric τ-fst.naturality3 τ-fst.preserves-arr)
      moreover have BC.Map (τ (fst f)) (snd f) \C
                      BC.Map (F (A.src (fst f))) (snd f) ⌢C
                    BC.Map (F (fst f)) (B.src (snd f)) \C
                      BC.Map (F (A.src (fst f))) (snd f)
      proof −
        have BC.Map (F (fst f)) (B.src (snd f)) \C
               BC.Map (F (A.src (fst f))) (snd f) =
             BC.Map (F (fst f)) (B.trg (snd f))
          using f F-fst.naturality1 by blast
        moreover
        have BC.Map (τ (fst f)) (snd f) \C
               BC.Map (F (A.src (fst f))) (snd f) =
             BC.Map (τ (fst f)) (B.trg (snd f))
          by (metis 2 C.composite-ofE C.cong-char C.join-ofE
```

```
        C.residuation-axioms F-fst.F.simulation-axioms
        τ-fst.naturality1 τ-fst.naturality3
        residuation.con-implies-arr(1)
        simulation.preserves-reflects-arr)
    moreover have BC.Map (F (fst f)) (B.trg (snd f)) ⌢C
                  BC.Map (τ (fst f)) (B.trg (snd f))
    proof −
      have BC.con (F (fst f)) (τ (fst f))
        by (meson AxB.arr-char BC.con-with-join-of-iff(2)
            BC.join-of-symmetric BC.residuation-axioms τ.preserves-arr
            assms f residuation.arr-def transformation.naturality3)
      thus ?thesis
        using f BC.con-char by simp
    qed
    ultimately show ?thesis
      using C.con-sym by presburger
  qed
  ultimately show ?thesis
  using 1 C.con-with-join-of-iff(1) C.con-sym C.join-is-join-of
    C.joinable-implies-con
    by blast
qed
ultimately show ?thesis
  using C.resid-join_E(3) by blast
qed
also have ... = BC.Map (G (A.trg (fst f))) (B.trg (snd f)) ⊔C
                (BC.Map (F (A.src (fst f))) (snd f) \C
                  BC.Map (τ (fst f)) (snd f))
proof −
  have BC.Map (F (fst f)) (B.src (snd f)) \C
         BC.Map (τ (fst f)) (snd f) =
       BC.Map (F (fst f)) (B.src (snd f)) \C
         (BC.Map (τ (fst f)) (B.src (snd f)) ⊔C
           BC.Map (F (A.src (fst f))) (snd f))
    using f τ-fst.naturality3 C.join-is-join-of C.join-of-unique
    by (metis AxB.arr-char C.joinable-def)
  also have ... = (BC.Map (F (fst f)) (B.src (snd f)) \C
                    BC.Map (τ (fst f)) (B.src (snd f))) \C
                  (BC.Map (F (A.src (fst f))) (snd f) \C
                    BC.Map (τ (fst f)) (B.src (snd f)))
    by (metis (no-types, lifting) AxB.arr-char C.conI C.con-sym-ax
        C.con-with-join-if(1) C.join-sym C.joinable-def
        C.joinable-iff-join-not-null C.null-is-zero(2)
        C.resid-join_E(1) τ-fst.naturality3 f C.conE)
  also have ... = BC.Map (G (A.trg (fst f))) (B.src (snd f)) \C
                  (BC.Map (F (A.src (fst f))) (snd f) \C
                    BC.Map (τ (fst f)) (B.src (snd f)))
  proof −
    have BC.resid (F (fst f)) (τ (fst f)) = G (A.trg (fst f))
```

```
    by (metis AxB.arr-char BC.arrE BC.arr-prfx-join-self BC.con-def
        BC.join-is-join-of BC.join-of-symmetric BC.join-of-unique
        BC.joinable-def BC.resid-join_E(1) BC.trg-def
        τ.G.preserves-trg τ.naturality2 τ.naturality3 f
        BC.ide-implies-arr)
  thus ?thesis
    using f BC.Map-resid-ide
    by (metis A.arr-trg-iff-arr AxB.arr-char B.ide-src
        τ.G.preserves-reflects-arr BC.arr-resid-iff-con)
qed
also have ... = BC.Map (G (A.trg (fst f))) (B.src (snd f)) \_C
                  BC.Map (G (A.trg (fst f))) (snd f)
  using AxB.arr-char τ-fst.naturality2 f by presburger
also have ... = BC.Map (G (A.trg (fst f))) (B.trg (snd f))
  by (metis AxB.arr-char B.con-arr-src(2) B.resid-src-arr
      τ-fst.G.preserves-resid f)
finally show ?thesis by simp
qed
also have ... = BC.Map (G (A.trg (fst f))) (B.trg (snd f)) ⊔_C
                BC.Map (G (A.trg (fst f))) (B.trg (snd f))
proof −
  have BC.Map (F (A.src (fst f))) (snd f) \_C
                       BC.Map (τ (fst f)) (snd f) =
       BC.Map (F (A.src (fst f))) (snd f) \_C
         (BC.Map (τ (fst f)) (B.src (snd f)) ⊔_C
           BC.Map (F (A.src (fst f))) (snd f))
    by (metis AxB.arr-char C.join-is-join-of C.join-of-unique
        C.joinable-def τ-fst.naturality3 f)
  also have ... = (BC.Map (F (A.src (fst f))) (snd f) \_C
                     BC.Map (F (A.src (fst f))) (snd f)) \_C
                   (BC.Map (τ (fst f)) (B.src (snd f)) \_C
                     BC.Map (F (A.src (fst f))) (snd f))
    by (metis (no-types, lifting) AxB.arr-char C.conE C.conI
        C.con-sym-ax C.con-with-join-of-iff(1) C.join-def
        C.null-is-zero(2) C.resid-join_E(1) τ-fst.naturality3
        calculation f)
  also have ... = BC.Map (F (A.src (fst f))) (B.trg (snd f)) \_C
                   (BC.Map (τ (fst f)) (B.src (snd f)) \_C
                     BC.Map (F (A.src (fst f))) (snd f))
    using AxB.arr-char C.trg-def F-fst.F.preserves-trg f
    by presburger
  also have ... = BC.Map (G (A.trg (fst f))) (B.trg (snd f))
    by (metis B.src-trg τ-fst.naturality1 τ-fst.naturality2)
  finally show ?thesis by simp
qed
also have ... = BC.Map (G (A.trg (fst f))) (B.trg (snd f))
  using C.join-arr-self f by auto
also have ... = Uncurry G (AxB.trg f)
  using Uncurry-def E.map-def f AxB.src-char AxB.trg-char
```

```
      τ.preserves-arr
  by simp
 finally show ?thesis by blast
qed
have 3: Uncurry τ (AxB.src f) =
       BC.Map (τ (A.src (fst f))) (B.src (snd f))
 using Uncurry-def E.map-def f AxB.src-char AxB.trg-char
      τ.preserves-arr
 by simp
have 4: Uncurry F f = BC.Map (F (fst f)) (snd f)
 using Uncurry-def E.map-def f by simp
have 5: Uncurry τ (AxB.trg f) =
       BC.Map (τ (A.trg (fst f))) (B.trg (snd f))
 using Uncurry-def E.map-def f AxB.src-char AxB.trg-char
      τ.preserves-arr
 by simp
show 6: Uncurry τ (AxB.src f) \C Uncurry F f = Uncurry τ (AxB.trg f)
 using 3 4 5 AxB.arr-char B.con-arr-src(2) B.resid-src-arr
      BC.joinable-implies-con BC.resid-Map τ.naturality1
      τ.naturality3'E(2) f
 by presburger
show 7: Uncurry F f \C Uncurry τ (AxB.src f) = Uncurry G f
 by (metis 3 4 AxB.arr-char B.con-arr-src(1) B.resid-arr-src BC.con-sym
     BC.joinable-implies-con BC.resid-Map Uncurry-simp E.map-def
     E.preserves-reflects-arr Uncurry-G.preserves-reflects-arr
     τ.naturality2 τ.naturality3'E(2) f fst-conv snd-conv)
show C.join-of (Uncurry τ (AxB.src f)) (Uncurry F f) (Uncurry τ f)
proof (intro C.join-ofI C.composite-ofI)
 have 8: Uncurry τ f \C Uncurry τ (AxB.src f) = Uncurry G f
 proof -
   have Uncurry τ f \C Uncurry τ (AxB.src f) =
        BC.Map (τ (fst f)) (snd f) \C BC.Map (τ (A.src (fst f)))
             (B.src (snd f))
     using Uncurry-def E.map-def f AxB.src-char AxB.trg-char
          τ.preserves-arr
     by simp
   also have ... = (BC.Map (τ (fst f)) (B.src (snd f)) ⨆C
                    BC.Map (F (A.src (fst f))) (snd f)) \C
                      BC.Map (τ (A.src (fst f))) (B.src (snd f))
     by (metis AxB.arr-char C.join-is-join-of C.join-of-unique
         C.joinable-def τ-fst.naturality3 f)
   also have ... = BC.Map (τ (fst f)) (B.src (snd f)) \C
                     BC.Map (τ (A.src (fst f))) (B.src (snd f)) ⨆C
                   BC.Map (F (A.src (fst f))) (snd f) \C
                     BC.Map (τ (A.src (fst f))) (B.src (snd f))
     by (metis 1 AxB.ide-trg C.conE C.conI C.join-def C.null-is-zero(2)
         C.prfx-implies-con C.resid-joinE(3) Uncurry-G.preserves-ide
         calculation f C.con-sym)
   also have ... = BC.Map (G (fst f)) (B.src (snd f)) ⨆C
```

```
                  BC.Map (F (A.src (fst f))) (snd f) \_C
                    BC.Map (τ (A.src (fst f))) (B.src (snd f))
  proof −
    have τ (fst f) \_[B,C] τ (A.src (fst f)) = G (fst f)
      by (metis (no-types, lifting) A.con-arr-src(2)
          BC.arr-resid-iff-con BC.join-src BC.resid-join_E(3)
          BC.src-resid BC.trg-def τ.G.preserves-reflects-arr
          τ.naturality2 τ.naturality3'_E(2) τ.naturality3'_E(1)
          τ.preserves-con(1) f AxB.arr-char)
    thus ?thesis
      using f BC.Map-resid-ide BC.con-char
      by (metis (no-types, lifting) AxB.arr-char B.ide-src BC.arr-char
          BC.resid-def τ.G.preserves-reflects-arr)
  qed
  also have ... = BC.Map (G (fst f)) (B.src (snd f)) ⊔_C
                  BC.Map (G (A.src (fst f))) (snd f)
    by (metis A.arr-src-iff-arr A.ide-src A.trg-src
        B.con-arr-src(1) B.resid-arr-src BC.joinable-implies-con
        BC.prfx-implies-con BC.resid-Map BC.resid-arr-ide
        τ.F.preserves-ide τ.G.preserves-ide τ.naturality1 τ.naturality2
        τ.naturality3'_E(2) f AxB.arr-char)
  also have ... = BC.Map (G (fst f)) (snd f)
    using Map-simulation-expansion τ.G.simulation-axioms f
    by presburger
  also have ... = Uncurry G f
    using Uncurry-def E.map-def f AxB.src-char AxB.trg-char
          τ.preserves-arr
    by simp
  finally show ?thesis by simp
qed
have 9: Uncurry τ f \_C Uncurry F f = Uncurry τ (AxB.trg f)
proof −
  have Uncurry τ f \_C Uncurry F f =
       BC.Map (τ (fst f)) (snd f) \_C BC.Map (F (fst f)) (snd f)
    using Uncurry-def E.map-def f AxB.src-char AxB.trg-char
          τ.preserves-arr
    by simp
  also have ... = (BC.Map (τ (fst f)) (B.src (snd f)) ⊔_C
                   BC.Map (F (A.src (fst f))) (snd f)) \_C
                  BC.Map (F (fst f)) (snd f)
    by (metis AxB.arr-char C.join-is-join-of C.join-of-unique
        C.joinable-def τ-fst.naturality3 f)
  also have ... = BC.Map (τ (fst f)) (B.src (snd f)) \_C
                    BC.Map (F (fst f)) (snd f) ⊔_C
                  BC.Map (F (A.src (fst f))) (snd f) \_C
                    BC.Map (F (fst f)) (snd f)
    by (metis 1 2 8 C.conI C.con-sym-ax C.join-def C.not-arr-null
        C.null-is-zero(2) C.resid-join_E(3)
        Uncurry-G.preserves-reflects-arr calculation f)
```

also have ... = $BC.Map\ (\tau\ (fst\ f))\ (B.src\ (snd\ f))\ \backslash_C$
$(BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f))\ \sqcup_C$
$BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f))\ \sqcup_C$
$BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)\ \backslash_C$
$(BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f))\ \sqcup_C$
$BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f))$
by (*metis AxB.arr-char C.join-is-join-of C.join-of-unique C.joinable-def F-fst.naturality3 f*)
also have ... = $(BC.Map\ (\tau\ (fst\ f))\ (B.src\ (snd\ f))\ \backslash_C$
$BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f)))\ \backslash_C$
$(BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)\ \backslash_C$
$BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f)))\ \sqcup_C$
$(BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)\ \backslash_C$
$(BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f))\ \sqcup_C$
$BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)))$
proof −
have $BC.Map\ (\tau\ (fst\ f))\ (B.src\ (snd\ f))\ \backslash_C$
$(BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f))\ \sqcup_C$
$BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)) =$
$(BC.Map\ (\tau\ (fst\ f))\ (B.src\ (snd\ f))\ \backslash_C$
$BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f))\ \backslash_C$
$(BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f))\ \backslash_C$
$BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f))$
using $C.resid\text{-}join_E(1)$ [*of* $BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f))$
$BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)$
$BC.Map\ (\tau\ (fst\ f))\ (B.src\ (snd\ f))$]
by (*metis AxB.arr-char C.conI C.con-sym-ax C.con-with-join-if*(2) *C.joinable-def C.null-is-zero*(2) *F-fst.naturality3 f*)
also have ... = $(BC.Map\ (\tau\ (fst\ f))\ (B.src\ (snd\ f))\ \backslash_C$
$BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f)))\ \backslash_C$
$(BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)\ \backslash_C$
$BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f)))$
using *C.cube* **by** *blast*
finally show *?thesis* **by** *presburger*
qed
also have ... = $(BC.Map\ (\tau\ (fst\ f))\ (B.src\ (snd\ f))\ \backslash_C$
$BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f)))\ \backslash_C$
$(BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)\ \backslash_C$
$BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f)))\ \sqcup_C$
$(BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)\ \backslash_C$
$BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f)))\ \backslash_C$
$(BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)\ \backslash_C$
$BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f)))$
using $C.resid\text{-}join_E(2)$ [*of* $BC.Map\ (F\ (fst\ f))\ (B.src\ (snd\ f))$
$BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)$
$BC.Map\ (F\ (A.src\ (fst\ f)))\ (snd\ f)$]
by (*metis* (*no-types*, *lifting*) *AxB.arr-char C.conI C.con-with-join-if*(2) *C.join-sym C.joinable-def C.joinable-iff-join-not-null C.joinable-implies-con*

```
      F-fst.naturality3 f)
  also have ... = (BC.Map (τ (fst f)) (B.src (snd f)) \_C
                   BC.Map (F (fst f)) (B.src (snd f))) \_C
                  (BC.Map (F (A.src (fst f))) (snd f) \_C
                    BC.Map (F (fst f)) (B.src (snd f))) ⊔_C
                   BC.Map (F (A.trg (fst f))) (B.trg (snd f))
    using C.apex-sym C.trg-def F-fst.naturality1 F-fst.preserves-trg f
    by auto
  also have ... = (BC.Map (τ (fst f)) (B.src (snd f)) \_C
                   BC.Map (F (fst f)) (B.src (snd f))) \_C
                  BC.Map (F (A.trg (fst f))) (snd f) ⊔_C
                 BC.Map (F (A.trg (fst f))) (B.trg (snd f))
    using AxB.arr-char F-fst.naturality2 f by presburger
  also have ... = BC.Map (τ (A.trg (fst f))) (B.src (snd f)) \_C
                   BC.Map (F (A.trg (fst f))) (snd f) ⊔_C
                  BC.Map (F (A.trg (fst f))) (B.trg (snd f))
  proof −
    have τ (fst f) \_[B,C] F (fst f) =
         (τ (A.src (fst f)) ⊔_[B,C] F (fst f)) \_[B,C] F (fst f)
      by (metis AxB.arr-char BC.join-is-join-of BC.join-of-unique
          BC.joinable-def τ.naturality3 f)
    also have ... = τ (A.src (fst f)) \_[B,C] F (fst f) ⊔_[B,C]
                    F (fst f) \_[B,C] F (fst f)
    by (metis AxB.arr-char BC.arr-prfx-join-self BC.con-def BC.join-sym
          BC.joinable-def BC.joinable-iff-join-not-null BC.not-ide-null
          BC.resid-join_E(3) τ.naturality3 f)
    also have ... = τ (A.src (fst f)) \_[B,C] F (fst f)
      by (metis A.arr-trg-iff-arr AxB.arr-char BC.join-src BC.join-sym
          BC.src-resid BC.trg-def τ.naturality1 τ.preserves-arr f
          BC.arr-resid-iff-con)
    also have ... = τ (A.trg (fst f))
      using τ.naturality1 f by blast
    finally have τ (fst f) \_[B,C] F (fst f) = τ (A.trg (fst f))
      by blast
    thus ?thesis
      using BC.Map-resid-ide
      by (metis A.arr-trg-iff-arr B.ide-src BC.arr-resid-iff-con
          τ.preserves-arr f AxB.arr-char)
  qed
  also have ... =  BC.Map (τ (A.trg (fst f))) (B.trg (snd f)) ⊔_C
                   BC.Map (F (A.trg (fst f))) (B.trg (snd f))
    by (metis (no-types, lifting) A.ide-trg A.src-trg AxB.arr-char BC.arrE
        BC.ide-char_ERTS BC.prfx-char τ.F.preserves-ide τ.G.preserves-ide
        τ.naturality2 f transformation.naturality1)
  also have ... = BC.Map (τ (A.trg (fst f))) (B.trg (snd f))
    by (metis (full-types) 1 2 5 6 7 8 AxB.cong-reflexive C.extensional
        Uncurry-G.preserves-prfx calculation f C.cube)
  also have ... = Uncurry τ (AxB.trg f)
using Uncurry-def E.map-def f AxB.src-char AxB.trg-char τ.preserves-arr
```

```
      by simp
    finally show ?thesis by blast
  qed
  show Uncurry τ (AxB.src f) ≲C Uncurry τ f
    using f 1 Uncurry-G.preserves-ide AxB.ide-trg by presburger
  show Uncurry F f ≲C Uncurry τ f
    using f 2 Uncurry-G.preserves-ide AxB.ide-trg by presburger
  show 10: C.cong (Uncurry τ f \C Uncurry τ (AxB.src f))
                  (Uncurry F f \C Uncurry τ (AxB.src f))
    using 7 8 C.ide-trg C.trg-def f by auto
  show C.cong (Uncurry τ f \C Uncurry F f)
              (Uncurry τ (AxB.src f) \C Uncurry F f)
    using 6 9 C.cube 10 by presburger
 qed
qed

lemma Uncurry-Curry:
assumes transformation AxB.resid C F G τ
shows Uncurry (Curry F G τ) = τ
proof
  interpret τ: transformation AxB.resid C F G τ using assms by auto
  interpret Curry-τ: transformation A BC.resid
                     ‹Curry3 F› ‹Curry3 G› ‹Curry F G τ›
    using assms Curry-preserves-transformations τ.transformation-axioms
    by simp
  fix f
  have ¬ AxB.arr f ⟹ Uncurry (Curry F G τ) f = τ f
    using Curry-def Uncurry-def τ.extensional by auto
  moreover have AxB.arr f ⟹ Uncurry (Curry F G τ) f = τ f
   by (simp add: Curry-τ.preserves-arr Currying.Uncurry-def Currying-axioms
       E.map-def Map-Curry)
  ultimately show Uncurry (Curry F G τ) f = τ f by blast
qed

lemma Curry-Uncurry:
assumes transformation A BC.resid F G τ
shows Curry (Uncurry F) (Uncurry G) (Uncurry τ) = τ
proof
  interpret τ: transformation A BC.resid F G τ
    using assms by blast
  interpret Uncurry-τ: transformation AxB.resid C
                       ‹Uncurry F› ‹Uncurry G› ‹Uncurry τ›
    using assms Uncurry-preserves-transformations by auto
  fix f
  show Curry (Uncurry F) (Uncurry G) (Uncurry τ) f = τ f
  proof (cases A.arr f)
    show ¬ A.arr f ⟹ ?thesis
      using Curry-def τ.extensional by auto
```

assume f: $A.arr\ f$
have $Curry\ (Uncurry\ F)\ (Uncurry\ G)\ (Uncurry\ \tau)\ f \neq BC.null$
by (*metis A.not-arr-null* τ.*F.extensional* τ.*F.preserves-reflects-arr Curry-preserves-transformations f transformation.preserves-arr Uncurry-*τ*.transformation-axioms*)
moreover
have $BC.Dom\ (Curry\ (Uncurry\ F)\ (Uncurry\ G)\ (Uncurry\ \tau)\ f) = BC.Dom\ (\tau\ f)$
proof
fix g
show $BC.Dom\ (Curry\ (Uncurry\ F)\ (Uncurry\ G)\ (Uncurry\ \tau)\ f)\ g = BC.Dom\ (\tau\ f)\ g$
using *Curry-def Uncurry-def E.map-def f Uncurry-*τ*.preserves-arr* τ*.preserves-arr*
apply *auto*[*1*]
apply (*metis A.ide-src BC.Map.simps*(*1*) *BC.src-simp BC.src-join-of*(*1*) τ*.naturality3* τ*.preserves-src*)
by (*metis BC.arrE simulation.extensional transformation-def*)
qed
moreover
have $BC.Cod\ (Curry\ (Uncurry\ F)\ (Uncurry\ G)\ (Uncurry\ \tau)\ f) = BC.Cod\ (\tau\ f)$
proof
fix g
have $BC.Cod\ (\tau\ f) = BC.Map\ (G\ (A.trg\ f))$
proof −
have $BC.trg\ (\tau\ f) = G\ (A.trg\ f)$
using f τ*.preserves-trg*
by (*metis BC.trg-join-of*(*2*) τ*.G.preserves-trg* τ*.naturality3* τ*.naturality2*)
thus *?thesis*
using f *BC.trg-simp* τ*.preserves-arr*
by (*metis BC.Map.simps*(*1*))
qed
thus $BC.Cod\ (Curry\ (Uncurry\ F)\ (Uncurry\ G)\ (Uncurry\ \tau)\ f)\ g = BC.Cod\ (\tau\ f)\ g$
using *Curry-def Uncurry-def E.map-def f Uncurry-*τ*.preserves-arr*
apply *auto*[*1*]
by (*metis A.ide-trg BC.ide-char*$_{ERTS}$ τ*.G.preserves-ide simulation.extensional*)
qed
moreover
have $BC.Map\ (Curry\ (Uncurry\ F)\ (Uncurry\ G)\ (Uncurry\ \tau)\ f) = BC.Map\ (\tau\ f)$
proof
fix g
show $BC.Map\ (Curry\ (Uncurry\ F)\ (Uncurry\ G)\ (Uncurry\ \tau)\ f)\ g = BC.Map\ (\tau\ f)\ g$
using *Curry-def Uncurry-def E.map-def f Uncurry-*τ*.preserves-arr*

```
      τ.preserves-arr
    apply auto[1]
    by (metis BC.arrE transformation.extensional)
  qed
  ultimately show ?thesis
    using BC.null-char
    by (metis BC.MkArr-Map BC.not-arr-null τ.preserves-arr f)
 qed
qed

lemma src-Curry:
assumes transformation AxB.resid C F G τ and A.arr f
shows BC.src (Curry F G τ f) = Curry3 F (A.src f)
proof −
 interpret τ: transformation A ‹(\⇩[B,C])› ‹Curry3 F› ‹Curry3 G›
              ‹Curry F G τ›
   using assms Curry-preserves-transformations by blast
 show ?thesis
   using assms(2)
   by (metis A.ide-src BC.src-join-of(1) τ.naturality3 τ.preserves-src)
qed

lemma trg-Curry:
assumes transformation AxB.resid C F G τ and A.arr f
shows BC.trg (Curry F G τ f) = Curry3 G (A.trg f)
proof −
 interpret τ: transformation A ‹(\⇩[B,C])› ‹Curry3 F› ‹Curry3 G›
              ‹Curry F G τ›
   using assms Curry-preserves-transformations by blast
 show ?thesis
   using assms(2)
   by (metis BC.trg-join-of(2) τ.G.preserves-trg τ.naturality3 τ.naturality2)
qed

lemma src-Uncurry:
assumes transformation A BC.resid F G τ and AxB.arr f
shows C.src (Uncurry τ f) = Uncurry F (AxB.src f)
proof −
 interpret τ: transformation AxB.resid C ‹Uncurry F› ‹Uncurry G›
              ‹Uncurry τ›
   using assms Uncurry-preserves-transformations by blast
 show ?thesis
   using assms(2)
   by (metis AxB.ide-src C.src-join-of(1) τ.naturality3 τ.preserves-src)
qed

lemma trg-Uncurry:
assumes transformation A BC.resid F G τ and AxB.arr f
shows C.trg (Uncurry τ f) = Uncurry G (AxB.trg f)
```

proof −
interpret τ: *transformation AxB.resid C* ‹*Uncurry F*› ‹*Uncurry G*›
‹*Uncurry* τ›
using *assms Uncurry-preserves-transformations* **by** *blast*
show *?thesis*
using *assms*(*2*)
by (*metis C.trg-join-of*(*2*) τ.*G.preserves-trg* τ.*naturality3* τ.*naturality2*)
qed

end

3.10.6 Currying and Uncurrying as Inverse Simulations

context *Currying*
begin

sublocale *AxB-C*: *exponential-rts AxB.resid C* ..
sublocale *A-BC*: *exponential-rts A BC.resid* ..

notation *AxB-C.resid* (**infix** $\backslash_{[AxB,C]}$ *55*)
notation *AxB-C.con* (**infix** $\frown_{[AxB,C]}$ *55*)
notation *A-BC.resid* (**infix** $\backslash_{[A,[B,C]]}$ *55*)
notation *A-BC.con* (**infix** $\frown_{[A,[B,C]]}$ *50*)

definition *CURRY* :: ('*a* × '*b*, '*c*) *AxB-C.arr* ⇒ ('*a*, ('*b*, '*c*) *BC.arr*) *A-BC.arr*
where *CURRY* ≡
(λ*X*. *if AxB-C.arr X*
then A-BC.MkArr
(*Curry* (*AxB-C.Dom X*) (*AxB-C.Dom X*) (*AxB-C.Dom X*))
(*Curry* (*AxB-C.Cod X*) (*AxB-C.Cod X*) (*AxB-C.Cod X*))
(*Curry* (*AxB-C.Dom X*) (*AxB-C.Cod X*) (*AxB-C.Map X*))
else A-BC.null)

lemma *Dom-CURRY* [*simp*]:
assumes *AxB-C.arr f*
shows *A-BC.Dom* (*CURRY f*) =
Curry (*AxB-C.Dom f*) (*AxB-C.Dom f*) (*AxB-C.Dom f*)
using *assms CURRY-def* **by** *auto*

lemma *Cod-CURRY* [*simp*]:
assumes *AxB-C.arr f*
shows *A-BC.Cod* (*CURRY f*) =
Curry (*AxB-C.Cod f*) (*AxB-C.Cod f*) (*AxB-C.Cod f*)
using *assms CURRY-def* **by** *auto*

lemma *Map-CURRY* [*simp*]:
assumes *AxB-C.arr f*
shows *A-BC.Map* (*CURRY f*) =
Curry (*AxB-C.Dom f*) (*AxB-C.Cod f*) (*AxB-C.Map f*)

```
  using assms CURRY-def by auto

lemma CURRY-preserves-con:
assumes con: AxB-C.con t u
shows A-BC.con (CURRY t) (CURRY u)
proof (unfold A-BC.con-char, intro conjI)
  have t: AxB-C.arr t
    using AxB-C.con-implies-arr(1) con by blast
  have u: AxB-C.arr u
    using AxB-C.con-implies-arr(2) con by blast
  show CURRY t ≠ AxB-C.Null
    unfolding CURRY-def
    using AxB-C.con-implies-arr(1) con by force
  show transformation A BC.resid
         (A-BC.Dom (CURRY t)) (A-BC.Cod (CURRY t))
         (A-BC.Map (CURRY t))
    using CURRY-def A-BC.null-char ‹CURRY t ≠ AxB-C.Null›
          Curry-preserves-transformations
    by fastforce
  show CURRY u ≠ AxB-C.Null
    unfolding CURRY-def
    using AxB-C.con-implies-arr(2) con by force
  show transformation A BC.resid
         (A-BC.Dom (CURRY u)) (A-BC.Cod (CURRY u))
         (A-BC.Map (CURRY u))
    using CURRY-def A-BC.null-char ‹CURRY u ≠ AxB-C.Null›
          Curry-preserves-transformations
    by auto
  show A-BC.Dom (CURRY t) = A-BC.Dom (CURRY u)
    using CURRY-def A-BC.Dom.simps(1) A-BC.null-char AxB-C.con-char
          ‹CURRY t ≠ AxB-C.Null› ‹CURRY u ≠ AxB-C.Null› con
    by presburger
  show ∀a. A.ide a ⟶
             BC.con (A-BC.Map (CURRY t) a) (A-BC.Map (CURRY u) a)
  proof (intro allI impI)
    fix a
    assume a: A.ide a
    have A-BC.Map (CURRY t) a =
         Curry (AxB-C.Dom t) (AxB-C.Cod t) (AxB-C.Map t) a
      using a t CURRY-def by simp
    moreover have A-BC.Map (CURRY u) a =
                  Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u) a
      using a u CURRY-def by simp
    moreover
    have BC.con (Curry (AxB-C.Dom t) (AxB-C.Cod t) (AxB-C.Map t) a)
                (Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u) a)
      unfolding BC.con-char
      using a t u con
      apply clarsimp
```

```
      apply (intro conjI)
      subgoal using AxB-C.arr-char AxB-C.null-char
        Curry-preserves-transformations
        by (metis A.ide-implies-arr BC.arr-char
          transformation.preserves-arr)
      subgoal using AxB-C.arr-char AxB-C.null-char
        Curry-preserves-transformations
        by (metis A.ide-implies-arr BC.arr-char
          transformation.preserves-arr)
      subgoal using AxB-C.arr-char AxB-C.null-char
        Curry-preserves-transformations
        by (metis A.ide-implies-arr BC.arr-char
          transformation.preserves-arr)
      subgoal by (meson A.ide-implies-arr AxB-C.arrE BC.arr-char
          Curry-preserves-transformations Currying-axioms
          transformation.preserves-arr)
       unfolding Curry-def
        using AxB-C.con-char
        by auto
    ultimately
    show BC.con (A-BC.Map (CURRY t) a) (A-BC.Map (CURRY u) a)
      by auto
  qed
qed

lemma CURRY-is-extensional:
assumes ¬ AxB-C.arr t
shows CURRY t = A-BC.null
  using assms CURRY-def by simp

lemma CURRY-preserves-arr:
assumes AxB-C.arr t
shows A-BC.arr (CURRY t)
  using assms CURRY-preserves-con by blast

lemma Dom-Map-CURRY-resid:
assumes con: AxB-C.con t u and f: A.arr f
shows BC.Dom (A-BC.Map (CURRY (t \[AxB,C] u)) f) =
      BC.Dom (A-BC.Map (CURRY t \[A,[B,C]] CURRY u) f)
proof −
  have t: AxB-C.arr t
    using assms(1) AxB-C.con-implies-arr(1) by blast
  have u: AxB-C.arr u
    using assms(1) AxB-C.con-implies-arr(2) by blast
  have A-BC.con (CURRY t) (CURRY u)
    using con CURRY-preserves-con by simp
  interpret Curry-trg-u: simulation A BC.resid ‹Curry3 (AxB-C.Cod t)›
    using t
    by (meson AxB-C.arrE Curry-preserves-simulations transformation-def)
```

interpret *Curry-trg-u*: *simulation-to-extensional-rts A BC.resid*
‹*Curry3* (*AxB-C.Cod t*)›
..
interpret *Curry-trg-u*: *transformation A BC.resid*
‹*Curry3* (*AxB-C.Cod u*)›
‹*Curry3* (*AxB-C.Cod u*)›
‹*Curry3* (*AxB-C.Cod u*)›
using *u*
by (*metis A-BC.arr-MkArr A-BC.arr-trg-iff-arr A-BC.trg-char Cod-CURRY CURRY-preserves-arr*)
interpret *Curry-u*: *transformation A BC.resid*
‹*Curry3* (*AxB-C.Dom u*)›
‹*Curry3* (*AxB-C.Cod u*)›
‹*Curry* (*AxB-C.Dom u*) (*AxB-C.Cod u*) (*AxB-C.Map u*)›
using *u Curry-preserves-transformations* **by** *blast*
interpret *Curry-u*: *transformation-to-extensional-rts A BC.resid*
‹*Curry3* (*AxB-C.Dom u*)›
‹*Curry3* (*AxB-C.Cod u*)›
‹*Curry* (*AxB-C.Dom u*) (*AxB-C.Cod u*) (*AxB-C.Map u*)›
..
have *BC.Dom* (*A-BC.Map* (*CURRY* (*t* $\backslash_{[AxB,C]}$ *u*)) *f*) =
(λ*g*. *AxB-C.Cod u* (*A.src f*, *g*))
using *f con Dom-Curry* **by** *simp*
also
have ... = *BC.Dom* (*Curry3* (*AxB-C.Cod u*) *f*)
using *f u Dom-Curry* **by** *auto*
also
have ... = *BC.Dom*
(*BC.join*
(*A-BC.Map* (*CURRY t*) (*A.src f*) $\backslash_{[B,C]}$
Curry (*AxB-C.Dom u*) (*AxB-C.Cod u*) (*AxB-C.Map u*)
(*A.src f*))
(*Curry3* (*AxB-C.Cod u*) *f*))
proof −
have *BC.joinable*
(*Curry3* (*AxB-C.Cod u*) *f*)
(*A-BC.Map* (*CURRY t*) (*A.src f*) $\backslash_{[B,C]}$
Curry (*AxB-C.Dom u*) (*AxB-C.Cod u*) (*AxB-C.Map u*) (*A.src f*))
proof −
have *BC.joinable*
(*Curry3* (*AxB-C.Dom u*) *f*)
(*A-BC.Map* (*CURRY t*) (*A.src f*))
using *u f*
by (*metis A-BC.con-char BC.join-of-symmetric BC.joinable-def Dom-CURRY* ‹*A-BC.con* (*CURRY t*) (*CURRY u*)› *transformation.naturality3*)
moreover
have *1*: *Curry* (*AxB-C.Dom u*) (*AxB-C.Cod u*) (*AxB-C.Map u*)
(*A.src f*) $\frown_{[B,C]}$

```
        (Curry3 (AxB-C.Dom u) f ⊔[B,C]
          A-BC.Map (CURRY t) (A.src f))
  proof (intro BC.con-with-join-if)
    show BC.joinable
          (Curry3 (AxB-C.Dom u) f)
          (A-BC.Map (CURRY t) (A.src f))
      using calculation by blast
    show A-BC.Map (CURRY t) (A.src f) ⌢[B,C]
         Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u) (A.src f)
      using A-BC.con-char ‹CURRY t ⌢[A,[B,C]] CURRY u› f u by auto
    show Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u)
          (A.src f) \[B,C]
           A-BC.Map (CURRY t) (A.src f) ⌢[B,C]
         Curry3 (AxB-C.Dom u) f \[B,C] A-BC.Map (CURRY t) (A.src f)
      using u A.ide-trg A-BC.con-char BC.conE BC.conI BC.con-sym-ax
            BC.cube Dom-CURRY Map-CURRY
            ‹CURRY t ⌢[A,[B,C]] CURRY u›
            f transformation.naturality1
      by (metis (full-types))
  qed
  ultimately
  have (Curry3 (AxB-C.Dom u) f ⊔[B,C]
          A-BC.Map (CURRY t) (A.src f)) \[B,C]
         Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u) (A.src f) =
         Curry3 (AxB-C.Dom u) f \[B,C]
           Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u)
            (A.src f) ⊔[B,C]
         A-BC.Map (CURRY t) (A.src f) \[B,C]
           Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u) (A.src f)
    using BC.resid-join_E(3)
          [of Curry3 (AxB-C.Dom u) f A-BC.Map (CURRY t) (A.src f)
              Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u)
                    (A.src f)]
    by blast
  hence BC.joinable
          (Curry3 (AxB-C.Dom u) f \[B,C]
             Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u)
               (A.src f))
          (A-BC.Map (CURRY t) (A.src f) \[B,C]
             Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u)
               (A.src f))
    by (metis 1 BC.arr-resid BC.con-sym BC.joinable-iff-arr-join)
  moreover
  have Curry3 (AxB-C.Cod u) f =
       Curry3 (AxB-C.Dom u) f \[B,C]
         Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u) (A.src f)
    using f u Curry-u.naturality2 by presburger
  ultimately show ?thesis by simp
qed
```

```
    thus ?thesis
      using f u BC.src-char BC.src-join BC.Dom-join BC.join-sym by auto
  qed
  also have ... = BC.Dom (A-BC.Map (CURRY t \[A,[B,C]] CURRY u) f)
    using f u con CURRY-preserves-con A-BC.Map-resid Dom-Curry
          Cod-Curry Cod-CURRY
    by simp
  finally show ?thesis by blast
qed

lemma Cod-Map-CURRY-resid:
assumes con: AxB-C.con t u and f: A.arr f
shows BC.Cod (A-BC.Map (CURRY (t \[AxB,C] u)) f) =
      BC.Cod (A-BC.Apex (CURRY t) (CURRY u) f)
proof -
  have t: AxB-C.arr t
    using assms(1) AxB-C.con-implies-arr(1) by blast
  have u: AxB-C.arr u
    using assms(1) AxB-C.con-implies-arr(2) by blast
  interpret Apex: simulation
                  A BC.resid ‹A-BC.Apex (CURRY t) (CURRY u)›
   using assms CURRY-preserves-con A-BC.Apex-is-simulation A-BC.con-char
    by blast
  interpret Cod-u: simulation AxB.resid C ‹AxB-C.Cod u›
    using AxB-C.ide-trg AxB-C.trg-char u by force
  interpret t: transformation
               AxB.resid C ‹AxB-C.Dom t› ‹AxB-C.Cod t› ‹AxB-C.Map t›
    using t AxB-C.arr-char by blast
  interpret u: transformation
               AxB.resid C ‹AxB-C.Dom u› ‹AxB-C.Cod u› ‹AxB-C.Map u›
    using u AxB-C.arr-char by blast

  have BC.Cod (A-BC.Map (CURRY (t \[AxB,C] u)) f) =
       BC.Cod (Curry3 (AxB-C.Apex t u) f)
    by (simp add: con Curry-def)
  also have ... = BC.Cod (A-BC.Apex (CURRY t) (CURRY u) f)
  proof
    fix g
    show BC.Cod (Curry3 (AxB-C.Apex t u) f) g =
         BC.Cod (A-BC.Apex (CURRY t) (CURRY u) f) g
    proof (cases B.arr g)
      assume g: ¬ B.arr g
      show ?thesis
        using f g Curry-def AxB-C.Apex-def A-BC.Apex-def
              CURRY-preserves-con
        apply simp
        using BC.Apex-def BC.Cod-resid BC.arr-resid-iff-con
              Apex.preserves-reflects-arr
        by presburger
```

```
next
assume g: B.arr g
show ?thesis
proof −
  have BC.Cod (Curry3 (AxB-C.Apex t u) f) g =
       AxB-C.Cod u (A.trg f, g) \_C
         (AxB-C.Map t (AxB.src (A.trg f, g)) \_C
           AxB-C.Map u (AxB.src (A.trg f, g)))
    using assms t u f g AxB.con-char Cod-Curry AxB-C.Apex-def
    by simp
  also have ... = BC.Cod
                    (A-BC.Cod (CURRY u) f \_[B,C]
                      (A-BC.Map (CURRY t) (A.src f) \_[B,C]
                        A-BC.Map (CURRY u) (A.src f))) g
  proof −
    have BC.con (A-BC.Map (CURRY t) (A.src f))
                (A-BC.Map (CURRY u) (A.src f))
      using assms f g t u AxB-C.con-char BC.con-char
            CURRY-preserves-con
      by (metis (no-types, lifting) A-BC.Apex-def BC.arr-char
          BC.conI BC.null-char Apex.simulation-axioms
          simulation.preserves-reflects-arr)
    moreover have BC.con (A-BC.Cod (CURRY u) f)
                         (A-BC.Map (CURRY t) (A.src f) \_[B,C]
                           A-BC.Map (CURRY u) (A.src f))
      using assms f g t u AxB-C.con-char BC.con-char
            CURRY-preserves-con
      by (metis A-BC.Apex-def BC.arr-char BC.conI BC.null-char
          Apex.simulation-axioms simulation.preserves-reflects-arr)
    moreover have AxB-C.Cod u (A.trg f, g) \_C
                    (AxB-C.Map t (AxB.src (A.trg f, g)) \_C
                      AxB-C.Map u (AxB.src (A.trg f, g))) =
                  (AxB-C.Cod u (A.src f, g) \_C
                    (AxB-C.Map t (A.src f, B.src g) \_C
                      AxB-C.Map u (A.src f, B.src g))) \_C
                  (AxB-C.Cod u (f, B.src g) \_C
                    (AxB-C.Map t (A.src f, B.src g) \_C
                      AxB-C.Map u (A.src f, B.src g) ⊔_C
                     AxB-C.Cod u (A.src f, B.src g)))
    proof −
      have (AxB-C.Cod u (A.src f, g) \_C
                    (AxB-C.Map t (A.src f, B.src g) \_C
                      AxB-C.Map u (A.src f, B.src g))) \_C
                  (AxB-C.Cod u (f, B.src g) \_C
                    (AxB-C.Map t (A.src f, B.src g) \_C
                      AxB-C.Map u (A.src f, B.src g) ⊔_C
                     AxB-C.Cod u (A.src f, B.src g))) =
           (AxB-C.Cod u (A.src f, g) \_C
             (AxB-C.Map t (A.src f, B.src g) \_C
```

```
          AxB-C.Map u (A.src f, B.src g))) \C
        (AxB-C.Cod u (f, B.src g) \C
          (AxB-C.Map t (A.src f, B.src g) \C
            AxB-C.Map u (A.src f, B.src g)))
proof −
  have C.src (AxB-C.Map t (A.src f, B.src g) \C
                AxB-C.Map u (A.src f, B.src g)) =
       AxB-C.Cod u (A.src f, B.src g)
  proof −
    have C.src (AxB-C.Map t (A.src f, B.src g) \C
                  AxB-C.Map u (A.src f, B.src g)) =
         C.trg (AxB-C.Map u (A.src f, B.src g))
      using AxB-C.con-char con f g by force
    also have ... = AxB-C.Cod u (A.src f, B.src g)
      by (metis A.ide-src AxB.product-rts-axioms
          AxB-C.Map.simps(1) AxB-C.Map-resid-ide
          AxB-C.con-arr-self AxB-C.trg-char
          AxB-C.trg-def B.ide-src C.trg-def f g
          fst-conv product-rts.ide-char snd-conv u)
    finally show ?thesis by blast
  qed
  thus ?thesis
    using C.join-src
    by (metis C.arr-resid C.conI C.join-sym
        C.null-is-zero(1−2))
qed
also have ... = (AxB-C.Cod u (A.src f, g) \C
                   AxB-C.Cod u (f, B.src g)) \C
                  ((AxB-C.Map t (A.src f, B.src g) \C
                     AxB-C.Map u (A.src f, B.src g)) \C
                   AxB-C.Cod u (f, B.src g))
  using C.cube by blast
also have ... = AxB-C.Cod u (A.trg f, g) \C
                  ((AxB-C.Map t (A.src f, B.src g) \C
                     AxB-C.Map u (A.src f, B.src g)) \C
                   AxB-C.Cod u (f, B.src g))
  by (metis A.con-arr-src(2) A.resid-src-arr AxB.con-char
      AxB.resid-def B.con-arr-src(1) B.resid-arr-src
      Cod-u.preserves-resid f g fst-conv snd-conv)
also have ... = AxB-C.Cod u (A.trg f, g) \C
                  ((AxB-C.Map t (A.src f, B.src g) \C
                     AxB-C.Map u (A.src f, B.src g)) \C
                   (AxB-C.Dom u (f, B.src g) \C
                     AxB-C.Map u (A.src f, B.src g)))
  by (metis AxB.arr-char AxB.src-char B.arr-src-iff-arr
      B.src-src f g fst-conv snd-conv u.naturality2)
also have ... = AxB-C.Cod u (A.trg f, g) \C
                  ((AxB-C.Map t (A.src f, B.src g) \C
                     AxB-C.Dom u (f, B.src g)) \C
```

```
                    (AxB-C.Map u (A.src f, B.src g) \_C
                      AxB-C.Dom u (f, B.src g)))
        using C.cube by presburger
      also have ... = AxB-C.Cod u (A.trg f, g) \_C
                    ((AxB-C.Map t (A.src f, B.src g) \_C
                      AxB-C.Dom t (f, B.src g)) \_C
                     (AxB-C.Map u (A.src f, B.src g) \_C
                      AxB-C.Dom u (f, B.src g)))
        using assms
        by (simp add: AxB-C.con-char)
      also have ... = AxB-C.Cod u (A.trg f, g) \_C
                    (AxB-C.Map t (A.trg f, B.src g) \_C
                      AxB-C.Map u (A.trg f, B.src g))
      proof −
        have AxB-C.Map t (A.src f, B.src g) \_C
               AxB-C.Dom t (f, B.src g) =
             AxB-C.Map t (A.trg f, B.src g)
          by (metis AxB.arr-char AxB.src-char AxB.trg-char
              B.arr-src-iff-arr B.src-src B.trg-src f g
              fst-conv snd-conv t.naturality1)
        moreover have AxB-C.Map u (A.src f, B.src g) \_C
                        AxB-C.Dom u (f, B.src g) =
                      AxB-C.Map u (A.trg f, B.src g)
          by (metis AxB.arr-char AxB.src-char AxB.trg-char
              B.arr-src-iff-arr B.src-src B.trg-src f g
              fst-conv snd-conv u.naturality1)
        ultimately show ?thesis by presburger
      qed
      also have ... = AxB-C.Cod u (A.trg f, g) \_C
                      (AxB-C.Map t (AxB.src (A.trg f, g)) \_C
                        AxB-C.Map u (AxB.src (A.trg f, g)))
        by (simp add: AxB.src-char f g)
      finally show ?thesis by argo
    qed
    ultimately show ?thesis
   using assms f g t u AxB-C.con-char CURRY-preserves-con BC.Apex-def
         Dom-Curry Cod-Curry Map-Curry BC.Cod-resid BC.Dom-resid
      by simp
  qed
  also have ... = BC.Cod (A-BC.Apex (CURRY t) (CURRY u) f) g
 using assms f g CURRY-preserves-con CURRY-preserves-arr A-BC.Apex-def
    by simp
  finally show ?thesis by auto
 qed
qed
qed
finally show ?thesis by blast
qed
```

lemma *Map-Map-CURRY-resid*:
assumes *con*: *AxB-C.con t u* **and** *f1*: *A.arr f1*
shows *BC.Map* (*A-BC.Map* (*CURRY* (*t* $\backslash_{[AxB,C]}$ *u*)) *f1*) =
BC.Map (*A-BC.Map* (*A-BC.resid* (*CURRY t*) (*CURRY u*)) *f1*)
proof −
have *t*: *AxB-C.arr t*
using *assms*(*1*) *AxB-C.con-implies-arr*(*1*) **by** *blast*
have *u*: *AxB-C.arr u*
using *assms*(*1*) *AxB-C.con-implies-arr*(*2*) **by** *blast*
interpret *t*: *transformation*
AxB.resid C ‹*AxB-C.Dom t*› ‹*AxB-C.Cod t*› ‹*AxB-C.Map t*›
using *t AxB-C.arr-char* [*of t*] **by** *blast*
interpret *t*: *transformation-of-binary-simulations*
A B C ‹*AxB-C.Dom t*› ‹*AxB-C.Cod t*› ‹*AxB-C.Map t*›
..
interpret *u*: *transformation*
AxB.resid C ‹*AxB-C.Dom u*› ‹*AxB-C.Cod u*› ‹*AxB-C.Map u*›
using *u AxB-C.arr-char* [*of u*] **by** *blast*
interpret *u*: *transformation-of-binary-simulations*
A B C ‹*AxB-C.Dom u*› ‹*AxB-C.Cod u*› ‹*AxB-C.Map u*›
..
interpret *tu*: *transformation*
AxB.resid C ‹*AxB-C.Cod u*› ‹*AxB-C.Apex t u*›
‹*AxB-C.Map* (*t* $\backslash_{[AxB,C]}$ *u*)›
using *con AxB-C.arr-char* [*of t* $\backslash_{[AxB,C]}$ *u*] **by** *simp*
interpret *tu*: *transformation-to-extensional-rts AxB.resid C*
‹*AxB-C.Cod u*› ‹*AxB-C.Apex t u*› ‹*AxB-C.Map* (*t* $\backslash_{[AxB,C]}$ *u*)›
..
interpret *Curry-t*: *transformation A BC.resid*
‹*Curry3* (*AxB-C.Dom t*)›
‹*Curry3* (*AxB-C.Cod t*)›
‹*Curry* (*AxB-C.Dom t*) (*AxB-C.Cod t*) (*AxB-C.Map t*)›
using *t Curry-preserves-transformations* **by** *blast*
interpret *Curry-u*: *transformation A BC.resid*
‹*Curry3* (*AxB-C.Dom u*)›
‹*Curry3* (*AxB-C.Cod u*)›
‹*Curry* (*AxB-C.Dom u*) (*AxB-C.Cod u*) (*AxB-C.Map u*)›
using *u Curry-preserves-transformations* **by** *blast*
interpret *Apex*: *simulation*
A BC.resid ‹*A-BC.Apex* (*CURRY t*) (*CURRY u*)›
using *assms CURRY-preserves-con A-BC.Apex-is-simulation A-BC.con-char*
by *blast*
interpret *Apex*: *simulation-to-extensional-rts*
A BC.resid ‹*A-BC.Apex* (*CURRY t*) (*CURRY u*)›
..
interpret *Cod-u*: *simulation AxB.resid C* ‹*AxB-C.Cod u*›
using *AxB-C.ide-trg AxB-C.trg-char u* **by** *force*
interpret *Cod-u*: *binary-simulation A B C* ‹*AxB-C.Cod u*› **..**
interpret *Cod-u*: *binary-simulation-between-weakly-extensional-rts*

```
                A B C ‹AxB-C.Cod u›
  ..

have ∗: ⋀f2. B.arr f2 ⟹
            ((AxB-C.Map t (A.src f1, B.src f2) \C
                AxB-C.Map u (A.src f1, B.src f2) ⊔C
              AxB-C.Cod u (A.src f1, B.src f2)) ⊔C
             AxB-C.Cod u (f1, B.src f2)) ⊔C
            AxB-C.Cod u (A.src f1, f2) =
            AxB-C.Map (t \[AxB,C] u) (f1, f2)
proof −
  fix f2
  assume f2: B.arr f2
  have 1:
    Curry (AxB-C.Dom t) (AxB-C.Cod t) (AxB-C.Map t) (A.src f1) ⌢[B,C]
     Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u) (A.src f1)
    by (metis A.ide-src A-BC.con-char con Map-CURRY f1
        CURRY-preserves-con t u)
  have ((AxB-C.Map t (A.src f1, B.src f2) \C
          AxB-C.Map u (A.src f1, B.src f2) ⊔C
        AxB-C.Cod u (A.src f1, B.src f2)) ⊔C
       AxB-C.Cod u (f1, B.src f2)) ⊔C
      AxB-C.Cod u (A.src f1, f2) =
      (AxB-C.Map t (A.src f1, B.src f2) \C
         AxB-C.Map u (A.src f1, B.src f2) ⊔C
       AxB-C.Cod u (f1, B.src f2)) ⊔C
      AxB-C.Cod u (A.src f1, f2)
    using AxB-C.Map-resid-ide con f1 f2 by force
  also have ... = (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) ⊔C
                   AxB-C.Cod u (f1, B.src f2)) ⊔C AxB-C.Cod u (A.src f1, f2)
    using f1 f2 con AxB-C.Map-resid-ide by simp
  also have ... = AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) ⊔C
                  AxB-C.Cod u (A.src f1, f2)
    using AxB.src-char AxB-C.Map-resid-ide con f1 f2 by force
  also have ... = AxB-C.Map (t \[AxB,C] u) (f1, f2)
  proof (intro C.join-eqI)
    show 2: AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) ≲C
            AxB-C.Map (t \[AxB,C] u) (f1, f2)
    proof −
      have AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) \C
             AxB-C.Map (t \[AxB,C] u) (f1, f2) =
           AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) \C
             (AxB-C.Map (t \[AxB,C] u) (AxB.src (f1, f2)) ⊔C
               AxB-C.Cod u (f1, f2))
        using f1 f2 AxB.prfx-char tu.naturality3
        by (metis AxB.arr-char C.join-is-join-of C.join-of-unique
            C.joinable-def fst-conv snd-conv)
      also have ... = (AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) \C
                       AxB-C.Map (t \[AxB,C] u) (AxB.src (f1, f2))) \C
```

```
                (AxB-C.Cod u (f1, f2) \C
                  AxB-C.Map (t \[AxB,C] u) (AxB.src (f1, f2)))
  using C.resid-join_E(2)
        [of AxB-C.Map (t \[AxB,C] u) (AxB.src (f1, f2))
            AxB-C.Cod u (f1, f2)
            AxB-C.Map (t \[AxB,C] u) (f1, B.src f2)]
  by (metis A.cong-reflexive AxB.prfx-char B.ide-trg
      B.resid-src-arr C.conI C.joinable-iff-join-not-null
      C.not-ide-null C.null-is-zero(2) calculation
      f1 f2 fst-conv snd-conv tu.preserves-prfx)
also have ... = AxB-C.Apex t u (f1, B.src f2) \C
                (AxB-C.Cod u (f1, f2) \C
                  AxB-C.Map (t \[AxB,C] u) (AxB.src (f1, f2)))
proof -
  have AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) \C
                   AxB-C.Map (t \[AxB,C] u) (AxB.src (f1, f2)) =
       (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) ⊔C
        AxB-C.Cod u (f1, B.src f2)) \C
         AxB-C.Map (t \[AxB,C] u) (AxB.src (f1, f2))
    using AxB.src-char AxB-C.Map-resid-ide con f1 f2 by fastforce
  also have ... = AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) \C
                    AxB-C.Map (t \[AxB,C] u) (AxB.src (f1, f2)) ⊔C
                  AxB-C.Cod u (f1, B.src f2) \C
                    AxB-C.Map (t \[AxB,C] u) (AxB.src (f1, f2))
    using C.resid-join_E(3)
    by (metis AxB.arr-char AxB.src-char B.arr-src-iff-arr B.src-src
        C.arr-prfx-join-self C.joinable-def C.prfx-implies-con
        f1 f2 fst-conv snd-conv tu.naturality3)
  also have ... = AxB-C.Apex t u (A.src f1, B.src f2) ⊔C
                  AxB-C.Cod u (f1, B.src f2) \C
                    AxB-C.Map (t \[AxB,C] u) (AxB.src (f1, f2))
    using AxB.src-char C.trg-def f1 f2 tu.preserves-trg by fastforce
  also have ... = AxB-C.Apex t u (A.src f1, B.src f2) ⊔C
                  AxB-C.Apex t u (f1, B.src f2)
    using AxB.src-char AxB-C.Map-resid-ide AxB-C.Apex-def con f2
    by force
  also have ... = AxB-C.Apex t u (f1, B.src f2)
  proof -
    have C.prfx (AxB-C.Apex t u (A.src f1, B.src f2))
                (AxB-C.Apex t u (f1, B.src f2))
      by (simp add: f1 f2)
    thus ?thesis
      by (metis AxB.arr-char AxB.ide-src AxB.src-char
          B.arr-src-iff-arr C.join-src C.prfx-implies-con
          C.src-ide f1 f2 fst-conv snd-conv transformation-def
          tu.G.preserves-ide tu.G.preserves-reflects-arr
          tu.transformation-axioms
          weakly-extensional-rts.con-imp-eq-src)
  qed
```

```
    finally show ?thesis by auto
  qed
  also
  have ... = AxB-C.Apex t u (f1, B.src f2) \C AxB-C.Apex t u (f1, f2)
    by (simp add: f1 f2 tu.naturality2)
  also have ... = AxB-C.Apex t u (A.trg f1, B.trg f2)
    by (metis A.residuation-axioms A.trg-def AxB.con-char AxB.resid-def
        B.con-arr-src(2) B.resid-src-arr f1 f2 fst-conv residuation.arrE
        snd-conv tu.G.preserves-resid)
  finally have AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) \C
                 AxB-C.Map (t \[AxB,C] u) (f1, f2) =
               AxB-C.Apex t u (A.trg f1, B.trg f2)
    by blast
  moreover have C.ide ...
    by (simp add: f1 f2)
  ultimately show ?thesis by simp
qed
show 3: AxB-C.Cod u (A.src f1, f2) ≾C
        AxB-C.Map (t \[AxB,C] u) (f1, f2)
proof −
  have AxB-C.Cod u (A.src f1, f2) \C
         AxB-C.Map (t \[AxB,C] u) (f1, f2) =
       AxB-C.Cod u (A.src f1, f2) \C
         (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) ⨆C
          AxB-C.Cod u (f1, f2))
    using AxB.src-char AxB-C.Map-resid-ide con f1 f2 by force
  also
  have ... = (AxB-C.Cod u (A.src f1, f2) \C AxB-C.Cod u (f1, f2)) \C
               (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) \C
                  AxB-C.Cod u (f1, f2))
  proof −
    have C.joinable (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2))
                    (AxB-C.Cod u (f1, f2))
      by (metis AxB.arr-char AxB.src-char C.joinable-def f1 f2
          fst-conv snd-conv tu.naturality3)
    moreover have AxB-C.Cod u (A.src f1, f2) ⌢C
                    AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) ⨆C
                      AxB-C.Cod u (f1, f2)
      by (metis A.con-arr-src(1) A.ide-trg A.resid-src-arr
          AxB.con-char AxB.prfx-char B.ide-trg B.residuation-axioms
          B.trg-def C.con-sym C.con-target C.con-with-join-if(2)
          C.joinable-implies-con calculation f1 f2 fst-conv
          residuation.arrE snd-conv u.G.preserves-con
          u.G.preserves-prfx)
    ultimately show ?thesis
      using C.resid-join_E(1) by fastforce
  qed
  also have ... = AxB-C.Cod u (A.trg f1, B.trg f2) \C
                    (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) \C
```

```
                    AxB-C.Cod u (f1, f2))
  proof −
    have AxB-C.Cod u (A.src f1, f2) \_C AxB-C.Cod u (f1, f2) =
         AxB-C.Cod u (A.trg f1, B.trg f2)

      using u.G.preserves-resid [of (A.src f1, f2) (f1, f2)]
      by (simp add: AxB.resid-def B.trg-def f1 f2)
    thus ?thesis
      by presburger
  qed
  also have ... = AxB-C.Cod u (A.trg f1, B.trg f2) \_C
                    AxB-C.Map (t \_[AxB,C] u) (A.trg f1, B.trg f2)
    by (metis (no-types, lifting) AxB.arr-char AxB.src-char AxB.trg-char
        f1 f2 fst-conv snd-conv tu.naturality1)
  also have ... = AxB-C.Apex t u (A.trg f1, B.trg f2)
    by (metis A.src-trg AxB.src-char AxB-C.Apex-def B.src-trg
        C.null-is-zero(2) fst-conv snd-conv tu.extensional
        tu.naturality2)
  finally have AxB-C.Cod u (A.src f1, f2) \_C
                 AxB-C.Map (t \_[AxB,C] u) (f1, f2) =
               AxB-C.Apex t u (A.trg f1, B.trg f2)
    by blast
  moreover have C.ide ...
    by (simp add: f1 f2)
  ultimately show ?thesis by simp
qed
show 4: AxB-C.Map (t \_[AxB,C] u) (f1, f2) \_C
          AxB-C.Cod u (A.src f1, f2) =
        AxB-C.Map (t \_[AxB,C] u) (f1, B.src f2) \_C
          AxB-C.Cod u (A.src f1, f2)
proof −
  have AxB-C.Map (t \_[AxB,C] u) (f1, f2) \_C
         AxB-C.Cod u (A.src f1, f2) =
       (AxB-C.Map (t \_[AxB,C] u) (A.src f1, B.src f2) ⊔_C
        AxB-C.Cod u (f1, f2)) \_C
         AxB-C.Cod u (A.src f1, f2)
    by (metis (no-types, lifting) AxB.ide-src AxB.src-char
        AxB-C.Map-resid AxB-C.Map-resid-ide C.arr-def
        C.ide-iff-src-self C.ide-implies-arr C.not-arr-null
        C.prfx-implies-con C.src-resid C.trg-def 2 con
        fst-conv snd-conv)
  also have ... = AxB-C.Map (t \_[AxB,C] u) (A.src f1, B.src f2) \_C
                    AxB-C.Cod u (A.src f1, f2) ⊔_C
                  AxB-C.Cod u (f1, f2) \_C AxB-C.Cod u (A.src f1, f2)
    by (metis C.conE C.conI C.con-sym C.joinable-iff-join-not-null
       C.null-is-zero(2) C.prfx-implies-con C.resid-join_E(3) 3 calculation)
  also have 4: ... = AxB-C.Map (t \_[AxB,C] u) (A.src f1, B.trg f2) ⊔_C
                       AxB-C.Cod u (f1, f2) \_C AxB-C.Cod u (A.src f1, f2)
    by (metis A.src-src A.trg-src AxB.src-char AxB.trg-char C.con-sym-ax
```

C.not-ide-null C.null-is-zero(2) 3 fst-conv snd-conv tu.naturality1 u.G.extensional)
also have ... = *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.trg f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.trg f2*)
using *f1 f2 u.G.preserves-con AxB.resid-def B.trg-def u.G.preserves-resid* [*of* (*f1*, *f2*) (*A.src f1*, *f2*)]

by *simp*
also have ... = *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*f1*, *B.trg f2*)
using *AxB.src-char AxB-C.Map-resid-ide con f1 f2* **by** *force*
finally have *L*: *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*f1*, *f2*) $\setminus_C$
AxB-C.Cod u (*A.src f1*, *f2*) =
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*f1*, *B.trg f2*)
by *blast*

have *5*: *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*f1*, *B.src f2*) $\setminus_C$
AxB-C.Cod u (*A.src f1*, *f2*) =
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*)) $\setminus_C$ *AxB-C.Cod u* (*A.src f1*, *f2*)
using *AxB.src-char AxB-C.Map-resid-ide con f1 f2* **by** *auto*
also have ... = *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Cod u (*A.src f1*, *f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*) $\setminus_C$ *AxB-C.Cod u* (*A.src f1*, *f2*)
proof −
have *C.joinable* (*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*))
(*AxB-C.Cod u* (*f1*, *B.src f2*))
by (*metis AxB.arr-char AxB.src-char B.arr-src-iff-arr B.src-src C.joinable-def f1 f2 fst-conv snd-conv tu.naturality3*)
moreover have *AxB-C.Cod u* (*A.src f1*, *f2*) $\frown_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*)
by (*metis 2 3 5 C.arr-resid-iff-con C.con-sym C.con-target C.prfx-implies-con C.resid-reflects-con*)
ultimately show *?thesis*
using *C.resid-join*$_E$(*3*) **by** *blast*
qed
also have ... = *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Cod u (*A.src f1*, *f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.trg f2*)
using *u.G.preserves-resid* [*of* (*f1*, *B.src f2*) (*A.src f1*, *f2*)]
by (*simp add*: *AxB.resid-def f1 f2*)
also have ... = *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.trg f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.trg f2*)
using *tu.naturality1* [*of* (*A.src f1*, *f2*)]
by (*simp add*: *AxB.src-char AxB.trg-char f1 f2*)
also have ... = *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*f1*, *B.trg f2*)
using *AxB.src-char AxB-C.Map-resid-ide con f1 f2* **by** *force*
finally have *R*: *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*f1*, *B.src f2*) $\setminus_C$
AxB-C.Cod u (*A.src f1*, *f2*) =

```
                AxB-C.Map (t \[AxB,C] u) (f1, B.trg f2)
    by blast

  show ?thesis
    using L R by auto
qed
show AxB-C.Map (t \[AxB,C] u) (f1, f2) \C
       AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) =
     AxB-C.Cod u (A.src f1, f2) \C
       AxB-C.Map (t \[AxB,C] u) (f1, B.src f2)
proof −
  have 5: AxB-C.Map (t \[AxB,C] u) (f1, f2) \C
            AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) =
          (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) ⊔C
            AxB-C.Cod u (f1, f2)) \C
            AxB-C.Map (t \[AxB,C] u) (f1, B.src f2)
    by (metis (no-types, lifting) AxB.ide-src AxB.src-char
        AxB-C.Map-resid AxB-C.Map-resid-ide C.arr-def
        C.ide-iff-src-self C.ide-implies-arr C.not-arr-null
        C.prfx-implies-con C.src-resid C.trg-def 3 con
        fst-conv snd-conv)
  also have 6: ... = AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) \C
                       AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) ⊔C
                     AxB-C.Cod u (f1, f2) \C
                       AxB-C.Map (t \[AxB,C] u) (f1, B.src f2)
 by (metis C.conI C.con-sym-ax C.joinable-iff-join-not-null C.not-ide-null
        C.null-is-zero(2) C.resid-join_E(3) 2 calculation)
  also have ... = (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) \C
                     AxB-C.Map (t \[AxB,C] u) (f1, B.src f2)) ⊔C
                   AxB-C.Apex t u (A.trg f1, f2)
  proof −
    have AxB-C.Cod u (f1, f2) \C
           AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) =
         (AxB-C.Cod u (f1, B.src f2) ⊔C AxB-C.Cod u (A.src f1, f2)) \C
           AxB-C.Map (t \[AxB,C] u) (f1, B.src f2)
    proof −
      have AxB.join-of (f1, B.src f2) (A.src f1, f2) (f1, f2)
        using f1 f2
        by (simp add: A.join-of-arr-src(2) AxB.join-of-char(1)
            B.join-of-arr-src(1))
      thus ?thesis
        using f1 f2 u.G.preserves-joins AxB.join-of-char
              C.join-is-join-of C.join-of-unique C.joinable-def
        by metis
    qed
    also have ... = (AxB-C.Cod u (f1, B.src f2) \C
                       AxB-C.Map (t \[AxB,C] u) (f1, B.src f2)) ⊔C
                     (AxB-C.Cod u (A.src f1, f2) \C
                       AxB-C.Map (t \[AxB,C] u) (f1, B.src f2))
```

using $C.resid\text{-}join_E(3)$ [*of AxB-C.Cod u* (*f1*, *B.src f2*)
AxB-C.Cod u (*A.src f1*, *f2*)
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*f1*, *B.src f2*)]
by (*metis 2 5 6 C.conE C.conI C.con-sym-ax*
C.join-def C.joinable-implies-con C.null-is-zero(*2*)
C.prfx-implies-con calculation)
also have ... = (*AxB-C.Cod u* (*f1*, *B.src f2*) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*))) $\sqcup_C$
(*AxB-C.Cod u* (*A.src f1*, *f2*) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*)))
using *f2 AxB.src-char AxB-C.Map-resid-ide 2 con*
by *force*
also have ... = (*AxB-C.Cod u* (*f1*, *B.src f2*) $\setminus_C$
AxB-C.Cod u (*f1*, *B.src f2*)) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Cod u (*f1*, *B.src f2*)) $\sqcup_C$
AxB-C.Cod u (*A.src f1*, *f2*) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*))
proof −
have *AxB-C.Cod u* (*f1*, *B.src f2*) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*)) =
(*AxB-C.Cod u* (*f1*, *B.src f2*) $\setminus_C$ *AxB-C.Cod u* (*f1*, *B.src f2*)) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Cod u (*f1*, *B.src f2*))
by (*metis* (*no-types*, *lifting*) *C.arr-prfx-join-self*
C.conI C.con-sym-ax C.joinable-iff-join-not-null
C.not-ide-null C.null-is-zero(*1*) $C.resid\text{-}join_E(1)$
5 2 calculation)
thus *?thesis* **by** *argo*
qed
also have ... = (*AxB-C.Cod u* (*A.trg f1*, *B.src f2*) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Cod u (*f1*, *B.src f2*))) $\sqcup_C$
(*AxB-C.Cod u* (*A.src f1*, *f2*) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*)))
using *u.G.preserves-trg* [*of* (*f1*, *B.src f2*)] *AxB.trg-char*
C.trg-def f1 f2
by *force*
also have ... = (*AxB-C.Cod u* (*A.trg f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.trg f1*, *B.src f2*)) $\sqcup_C$
(*AxB-C.Cod u* (*A.src f1*, *f2*) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*)))
using *tu.naturality1* [*of* (*f1*, *B.src f2*)] *AxB.src-char*

```
        AxB.trg-char f1 f2
    by force
  also have ... = AxB-C.Apex t u (A.trg f1, B.src f2) ⊔C
                  (AxB-C.Cod u (A.src f1, f2) \C
                    (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) ⊔C
                      AxB-C.Cod u (f1, B.src f2)))
    by (metis A.src-trg AxB.src-char AxB-C.Apex-def B.src-src
          C.ex-un-null C.null-is-zero(1) fst-conv snd-conv
          tu.extensional tu.naturality2)
  also have ... = AxB-C.Apex t u (A.trg f1, B.src f2) ⊔C
                  ((AxB-C.Cod u (A.src f1, f2) \C
                      AxB-C.Cod u (f1, B.src f2)) \C
                    (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) \C
                      AxB-C.Cod u (f1, B.src f2)))
    by (metis C.arr-prfx-join-self C.conI C.con-sym-ax
        C.joinable-iff-join-not-null C.not-ide-null C.null-is-zero(2)
        C.resid-joinE(1) 2 5 calculation)
  also have ... = AxB-C.Apex t u (A.trg f1, B.src f2) ⊔C
                  (AxB-C.Cod u (A.src f1, f2) \C
                     AxB-C.Cod u (f1, B.src f2)) \C
                     AxB-C.Map (t \[AxB,C] u) (A.trg f1, B.src f2)
    by (metis AxB.src-char AxB.trg-char B.src-src B.trg-src
          C.not-ide-null C.null-is-zero(1) 2 fst-conv
          snd-conv tu.extensional tu.naturality1)
  also have ... = AxB-C.Apex t u (A.trg f1, B.src f2) ⊔C
                  AxB-C.Cod u (A.trg f1, f2) \C
                    AxB-C.Map (t \[AxB,C] u) (A.trg f1, B.src f2)
    using u.G.preserves-resid [of (A.src f1, f2) (f1, B.src f2)]
          AxB.resid-def f1 f2
    by auto
  also have ... = AxB-C.Apex t u (A.trg f1, B.src f2) ⊔C
                  AxB-C.Apex t u (A.trg f1, f2)
    by (metis A.arr-trg-iff-arr A.src-trg AxB.arr-char AxB.src-char
          f1 f2 fst-conv snd-conv tu.naturality2)
  also have ... = AxB-C.Apex t u (A.trg f1, f2)
  proof −
    have AxB.join-of (A.trg f1, B.src f2) (A.trg f1, f2) (A.trg f1, f2)
      by (simp add: AxB.join-of-arr-src(1) f1 f2)
    thus ?thesis
      using tu.G.preserves-joins C.join-is-join-of
            C.join-of-unique C.joinable-def
      by meson
  qed
  finally have AxB-C.Cod u (f1, f2) \C
                 AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) =
               AxB-C.Apex t u (A.trg f1, f2)
    by blast
  thus ?thesis by auto
qed
```

also have ... = (*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*)) $\setminus_C$
(*AxB-C.Cod u* (*f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*)) $\sqcup_C$
AxB-C.Apex t u (*A.trg f1*, *f2*)
proof −
have *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*f1*, *B.src f2*) =
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*))
using *f1 f2 AxB.src-char AxB-C.Map-resid-ide con* **by** *auto*
also have ... = (*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*)) $\setminus_C$
(*AxB-C.Cod u* (*f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*))
proof −
have *C.joinable* (*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*))
(*AxB-C.Cod u* (*f1*, *B.src f2*))
by (*metis C.arr-prfx-join-self C.con-sym-ax*
C.joinable-iff-join-not-null C.not-ide-null C.null-is-zero(*1*)
2 5 6 calculation)
moreover have *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\frown_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*)
using *C.arr-prfx-join-self C.prfx-implies-con calculation*
by *presburger*
ultimately have *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\sqcup_C$
AxB-C.Cod u (*f1*, *B.src f2*)) =
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*)) $\setminus_C$
(*AxB-C.Cod u* (*f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*))
using *f1 f2 C.resid-join*$_E$(*2*) **by** *blast*
thus *?thesis* **by** *blast*
qed
finally have *AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*f1*, *B.src f2*) =
(*AxB-C.Map* (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*)) $\setminus_C$
(*AxB-C.Cod u* (*f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*))
by *blast*
thus *?thesis* **by** *argo*
qed
also have ... = *AxB-C.Apex t u* (*A.src f1*, *B.src f2*) $\setminus_C$
(*AxB-C.Cod u* (*f1*, *B.src f2*) $\setminus_C$
AxB-C.Map (*t* $\setminus_{[AxB,C]}$ *u*) (*A.src f1*, *B.src f2*)) $\sqcup_C$

```
                AxB-C.Apex t u (A.trg f1, f2)
  by (metis AxB.ide-src AxB.src-char C.not-ide-null
          C.null-is-zero(2) C.trg-def 2 fst-conv snd-conv tu.extensional
          tu.preserves-trg)
also have ... = AxB-C.Apex t u (A.src f1, B.src f2) \C
                  AxB-C.Apex t u (f1, B.src f2) ⊔C
                AxB-C.Apex t u (A.trg f1, f2)
  by (metis AxB.src-char B.src-trg B.trg-src C.con-sym-ax
      C.not-ide-null C.null-is-zero(2) 4 3 fst-conv snd-conv
      tu.extensional tu.naturality2)
also have ... = AxB-C.Apex t u (A.trg f1, B.src f2) ⊔C
                AxB-C.Apex t u (A.trg f1, f2)
  using tu.G.preserves-trg [of (f1, B.src f2)]
  by (simp add: AxB.trg-char C.con-imp-coinitial C.resid-ide(2) f1 f2)
also have ... = AxB-C.Apex t u (A.trg f1, f2)
proof −
  have AxB.join-of (A.trg f1, B.src f2) (A.trg f1, f2) (A.trg f1, f2)
    by (simp add: AxB.join-of-arr-src(1) f1 f2)
  thus ?thesis
    using tu.G.preserves-joins C.join-is-join-of C.join-of-unique
          C.joinable-def
    by meson
qed
finally have L: AxB-C.Map (t \[AxB,C] u) (f1, f2) \C
                  AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) =
                AxB-C.Apex t u (A.trg f1, f2)
  by blast

have R: AxB-C.Cod u (A.src f1, f2) \C
          AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) =
        AxB-C.Apex t u (A.trg f1, f2)
proof −
  have AxB-C.Cod u (A.src f1, f2) \C
         AxB-C.Map (t \[AxB,C] u) (f1, B.src f2) =
       AxB-C.Cod u (A.src f1, f2) \C
         (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) ⊔C
          AxB-C.Cod u (f1, B.src f2))
    using AxB.src-char AxB-C.Map-resid-ide con f1 f2 by force
  also have ... = (AxB-C.Cod u (A.src f1, f2) \C
                     AxB-C.Cod u (f1, B.src f2)) \C
                   (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) \C
                     AxB-C.Cod u (f1, B.src f2))
    by (metis C.conI C.con-sym-ax C.joinable-iff-join-not-null
        C.not-ide-null C.null-is-zero(2) C.resid-join_E(1) 3 4
        calculation)
  also have ... = AxB-C.Cod u (A.trg f1, f2) \C
                   (AxB-C.Map (t \[AxB,C] u) (A.src f1, B.src f2) \C
                     AxB-C.Cod u (f1, B.src f2))
    using u.G.preserves-resid [of (A.src f1, f2) (f1, B.src f2)]
```

```
        by (simp add: AxB.resid-def f1 f2)
      also have ... = AxB-C.Cod u (A.trg f1, f2) \C
                        AxB-C.Map (t \[AxB,C] u) (A.trg f1, B.src f2)
        by (metis AxB.src-char AxB.trg-char B.src-src B.trg-src
            C.ex-un-null C.not-ide-null C.null-is-zero(2) 2
            fst-conv snd-conv tu.extensional tu.naturality1)
      also have ... = AxB-C.Apex t u (A.trg f1, f2)
        by (metis A.src-trg AxB.src-char C.conI C.con-sym-ax
            C.prfx-implies-con C.residuation-axioms L 2 fst-conv
            residuation.arr-resid snd-conv tu.G.preserves-reflects-arr
            tu.naturality2)
      finally show ?thesis by blast
    qed
    show ?thesis
      using L R by simp
  qed
qed
finally show ((AxB-C.Map t (A.src f1, B.src f2) \C
                 AxB-C.Map u (A.src f1, B.src f2) ⨿C
               AxB-C.Cod u (A.src f1, B.src f2)) ⨿C
              AxB-C.Cod u (f1, B.src f2)) ⨿C
             AxB-C.Cod u (A.src f1, f2) =
             AxB-C.Map (t \[AxB,C] u) (f1, f2)
  by simp
qed
have BC.Map (A-BC.Map (CURRY (t \[AxB,C] u)) f1) =
     (λf2. if B.arr f2
           then AxB-C.Map t (AxB.src (f1, f2)) \C
                  AxB-C.Map u (AxB.src (f1, f2)) ⨿C
                AxB-C.Cod u (f1, f2)
           else C.null)
  unfolding curry-def
  using f1 con Curry-simp by simp
also have ... = (λf2. if B.arr f2
                      then ((AxB-C.Map t (A.src f1, B.src f2) \C
                               AxB-C.Map u (A.src f1, B.src f2) ⨿C
                             AxB-C.Cod u (A.src f1, B.src f2)) ⨿C
                            AxB-C.Cod u (f1, B.src f2)) ⨿C
                           AxB-C.Cod u (A.src f1, f2)
                      else C.null)
  using ∗ con f1 by auto
also
have ... = BC.Map (A-BC.Map (A-BC.resid (CURRY t) (CURRY u)) f1)
proof −
  have 1: Curry
            (AxB-C.Dom t) (AxB-C.Cod t) (AxB-C.Map t) (A.src f1) ⌢[B,C]
          Curry
            (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u) (A.src f1)
    by (metis A.ide-src A-BC.con-char Map-CURRY con
```

```
      CURRY-preserves-con f1 t u)
moreover
have BC.Dom
       (Curry (AxB-C.Dom t) (AxB-C.Cod t) (AxB-C.Map t)
          (A.src f1) \[B,C]
          Curry (AxB-C.Dom u) (AxB-C.Cod u) (AxB-C.Map u)
            (A.src f1)) =
     (λf2. AxB-C.Cod u (A.src f1, f2))
  using f1 t u 1 BC.Dom-resid Map-Curry Dom-Curry Cod-Curry by simp
moreover have transformation B C (λf2. AxB-C.Cod u (A.src f1, f2))
                                 (λf2. AxB-C.Cod u (A.trg f1, f2))
                                 (λf2. AxB-C.Cod u (f1, f2))
  using f1 Cod-u.fixing-arr-gives-transformation-1 by simp
ultimately
have BC.Map (A-BC.Map (A-BC.resid (CURRY t) (CURRY u)) f1) =
     (λf2. (BC.Resid
             (A-BC.MkArr
               (λg. AxB-C.Dom t (A.src f1, g))
               (λg. AxB-C.Cod t (A.src f1, g))
               (λg. AxB-C.Map t (A.src f1, g)))
             (A-BC.MkArr
               (λg. AxB-C.Dom u (A.src f1, g))
               (λg. AxB-C.Cod u (A.src f1, g))
               (λg. AxB-C.Map u (A.src f1, g)))
             (B.src f2) ⊔C
           AxB-C.Cod u (f1, B.src f2)) ⊔C
         AxB-C.Cod u (A.src f1, f2))
  using ∗ t u f1 C.joinable-iff-arr-join tu.preserves-arr con
        CURRY-preserves-con A-BC.Map-resid BC.join-char
        Map-CURRY BC.Map-resid-ide Map-Curry Dom-Curry Curry-def
  by auto
also have ... = (λf2. if B.arr f2
                    then ((AxB-C.Map t (A.src f1, B.src f2) \C
                           AxB-C.Map u (A.src f1, B.src f2) ⊔C
                          AxB-C.Cod u (A.src f1, B.src f2)) ⊔C
                         AxB-C.Cod u (f1, B.src f2)) ⊔C
                        AxB-C.Cod u (A.src f1, f2)
                    else C.null)
proof
  fix f2
  show (BC.Resid
          (A-BC.MkArr
            (λg. AxB-C.Dom t (A.src f1, g))
            (λg. AxB-C.Cod t (A.src f1, g))
            (λg. AxB-C.Map t (A.src f1, g)))
          (A-BC.MkArr
            (λg. AxB-C.Dom u (A.src f1, g))
            (λg. AxB-C.Cod u (A.src f1, g))
            (λg. AxB-C.Map u (A.src f1, g)))
```

```
            (B.src f2) ⨆C
           AxB-C.Cod u (f1, B.src f2)) ⨆C
          AxB-C.Cod u (A.src f1, f2) =
         (if B.arr f2
          then ((AxB-C.Map t (A.src f1, B.src f2) \C
                 AxB-C.Map u (A.src f1, B.src f2) ⨆C
                AxB-C.Cod u (A.src f1, B.src f2)) ⨆C
               AxB-C.Cod u (f1, B.src f2)) ⨆C
              AxB-C.Cod u (A.src f1, f2)
          else C.null)
      apply (cases B.arr f2)
       apply force
      using B.src-def C.arr-prfx-join-self C.join-def
      by fastforce
  qed
  finally show ?thesis by simp
 qed
 finally show ?thesis by blast
qed

lemma Cod-Curry-Apex:
assumes con: AxB-C.con t u and f1: A.arr f1
shows BC.Cod
        (Curry (AxB-C.Apex t u) (AxB-C.Apex t u) (AxB-C.Apex t u) f1) =
      BC.Cod (A-BC.Map (A-BC.resid (CURRY t) (CURRY u)) f1)
proof −
 have BC.Cod (Curry3 (AxB-C.Apex t u) f1) =
       BC.Cod (A-BC.Map (CURRY (t \[AxB,C] u)) f1)
   by (simp add: assms(1) Curry-def)
 also have ... = BC.Cod (A-BC.Apex (CURRY t) (CURRY u) f1)
   using assms Cod-Map-CURRY-resid by simp
 also have ... =
           BC.Cod (A-BC.Map (A-BC.resid (CURRY t) (CURRY u)) f1)
 proof −
   have BC.trg (A-BC.Map (CURRY t) (A.src f1) \[B,C]
                  A-BC.Map (CURRY u) (A.src f1) ⨆[B,C]
                 A-BC.Cod (CURRY u) f1) =
        BC.trg (A-BC.Cod (CURRY u) f1 \[B,C]
                  (A-BC.Map (CURRY t) (A.src f1) \[B,C]
                    A-BC.Map (CURRY u) (A.src f1)))
     by (metis A-BC.Map-preserves-prfx A-BC.Map-resid
        A-BC.cong-reflexive BC.apex-sym BC.joinable-iff-join-not-null
        BC.not-ide-null BC.null-is-zero(2) BC.trg-join
        CURRY-preserves-con con f1 A-BC.arr-resid)
   hence BC.Cod (A-BC.Map (CURRY t) (A.src f1) \[B,C]
                   A-BC.Map (CURRY u) (A.src f1) ⨆[B,C]
                  A-BC.Cod (CURRY u) f1) =
         BC.Cod (A-BC.Cod (CURRY u) f1 \[B,C]
                  (A-BC.Map (CURRY t) (A.src f1) \[B,C]
```

```
                    A-BC.Map (CURRY u) (A.src f1)))
      using BC.trg-char
      by (metis (no-types, lifting) BC.Map-trg BC.arr-trg-iff-arr
          BC.con-imp-arr-resid BC.join-char(2)
          BC.joinable-iff-arr-join BC.trg-def)
    thus ?thesis
      using assms CURRY-preserves-con A-BC.Map-resid A-BC.Apex-def
      by simp
  qed
  finally show ?thesis by blast
qed

lemma Curry-preserves-Apex:
assumes AxB-C.con t u
shows Curry3 (AxB-C.Apex t u) = A-BC.Apex (CURRY t) (CURRY u)
  (is ?LHS = ?RHS)
proof -
  have t: AxB-C.arr t
    using assms AxB-C.con-implies-arr(1) by blast
  have u: AxB-C.arr u
    using assms AxB-C.con-implies-arr(2) by blast
  interpret Apex: simulation
                A BC.resid ‹A-BC.Apex (CURRY t) (CURRY u)›
   using assms CURRY-preserves-con A-BC.Apex-is-simulation A-BC.con-char
    by blast
  interpret Cod-u: simulation AxB.resid C ‹AxB-C.Cod u›
    using AxB-C.ide-trg AxB-C.trg-char u by force
  interpret Curry-Apex: simulation A BC.resid
                        ‹Curry (AxB-C.Apex t u) (AxB-C.Apex t u)
                        (AxB-C.Apex t u)›
    using Curry-preserves-simulations AxB-C.Apex-is-simulation
          AxB-C.con-char assms
    by blast
  show ?thesis
  proof
    fix f1
    show ?LHS f1 = ?RHS f1
    proof (cases A.arr f1)
      show ¬ A.arr f1 ⟹ ?thesis
        using A-BC.Apex-def Curry-def by force
      assume f1: A.arr f1
      interpret Apex-curry: transformation B C
                            ‹BC.Dom (A-BC.Apex (CURRY t) (CURRY u) f1)›
                            ‹BC.Cod (A-BC.Apex (CURRY t) (CURRY u) f1)›
                            ‹BC.Map (A-BC.Apex (CURRY t) (CURRY u) f1)›
        using f1 BC.arr-char by blast
      interpret Map-Curry-Apex:
                transformation B C
                  ‹BC.Dom
```

```
            (Curry
              (AxB-C.Apex t u)
              (AxB-C.Apex t u)
              (AxB-C.Apex t u)
            f1)›
          ‹BC.Cod
            (Curry
              (AxB-C.Apex t u)
              (AxB-C.Apex t u)
              (AxB-C.Apex t u)
            f1)›
          ‹BC.Map
            (Curry
              (AxB-C.Apex t u)
              (AxB-C.Apex t u)
              (AxB-C.Apex t u)
            f1)›
  using f1 Curry-Apex.preserves-reflects-arr BC.arr-char
  by blast
show ?thesis
proof −
  have BC.Dom (?LHS f1) = BC.Dom (?RHS f1)
    by (metis A.con-arr-src(1) A.con-arr-src(2) A.con-implies-arr(1)
        A.ide-src A.resid-arr-src A-BC.Apex-def A-BC.Map-resid-ide
        BC.Dom-resid BC.arr-resid-iff-con Apex.preserves-reflects-arr
        Cod-Curry-Apex Curry-Apex.preserves-con
        Curry-Apex.preserves-resid assms CURRY-preserves-con f1)
  moreover
  have BC.Cod (?LHS f1) = BC.Cod (?RHS f1)
    using Cod-Curry-Apex Cod-Map-CURRY-resid assms f1 Curry-def
    by auto
  moreover
  have BC.Map (?LHS f1) = BC.Map (?RHS f1)
  proof
    fix f2
    show BC.Map (?LHS f1) f2 = BC.Map (?RHS f1) f2
    proof (cases B.arr f2)
      show ¬ B.arr f2 ⟹ ?thesis
        by (simp add: Apex-curry.extensional
            Map-Curry-Apex.extensional)
      assume f2: B.arr f2
      show ?thesis
      proof −
        have 0: BC.Map (?LHS f1) f2 =
                  AxB-C.Cod u (f1, f2) \C
                    (AxB-C.Map t (AxB.src (f1, f2)) \C
                       AxB-C.Map u (AxB.src (f1, f2)))
          using assms t u f1 f2 AxB.con-char Map-Curry AxB-C.Apex-def
          by simp
```

```
also have ... =
        BC.Map
          (A-BC.Cod (CURRY u) f1 \[B,C]
            (A-BC.Map (CURRY t) (A.src f1) \[B,C]
              A-BC.Map (CURRY u) (A.src f1))) f2
proof −
  have 1: AxB.join-of (f1, B.src f2) (A.src f1, f2) (f1, f2)
    by (simp add: A.join-of-arr-src(2) AxB.join-of-char(1)
        B.join-of-arr-src(1) f1 f2)
  have A-BC.Map (CURRY t) (A.src f1) ⌢[B,C]
       A-BC.Map (CURRY u) (A.src f1)
    using A.ide-src A-BC.con-char assms CURRY-preserves-con f1
    by presburger
  moreover have A-BC.Cod (CURRY u) f1 ⌢[B,C]
                A-BC.Map (CURRY t) (A.src f1) \[B,C]
                  A-BC.Map (CURRY u) (A.src f1)
    by (metis A-BC.Apex-def BC.arrE BC.conI
        Apex.preserves-reflects-arr f1)
  moreover have AxB-C.Cod u (f1, f2) \C
                  (AxB-C.Map t (AxB.src (f1, f2)) \C
                    AxB-C.Map u (AxB.src (f1, f2))) =
                AxB-C.Cod u (f1, B.src f2) \C
                  ((AxB-C.Map t (A.src f1, B.src f2) \C
                      AxB-C.Map u (A.src f1, B.src f2)) ⊔C
                    AxB-C.Cod u (A.src f1, B.src f2)) ⊔C
                  AxB-C.Cod u (A.src f1, f2) \C
                    (AxB-C.Map t (A.src f1, B.src f2) \C
                      AxB-C.Map u (A.src f1, B.src f2))
  proof −
    have AxB-C.Cod u (f1, B.src f2) \C
           ((AxB-C.Map t (A.src f1, B.src f2) \C
               AxB-C.Map u (A.src f1, B.src f2)) ⊔C
             AxB-C.Cod u (A.src f1, B.src f2)) ⊔C
         AxB-C.Cod u (A.src f1, f2) \C
           (AxB-C.Map t (A.src f1, B.src f2) \C
             AxB-C.Map u (A.src f1, B.src f2)) =
         AxB-C.Cod u (f1, B.src f2) \C
           (AxB-C.Map t (A.src f1, B.src f2) \C
             AxB-C.Map u (A.src f1, B.src f2)) ⊔C
         AxB-C.Cod u (A.src f1, f2) \C
           (AxB-C.Map t (A.src f1, B.src f2) \C
             AxB-C.Map u (A.src f1, B.src f2))
      using AxB-C.Map-resid-ide assms f1 f2 by fastforce
    also have ... = (AxB-C.Cod u (f1, B.src f2) ⊔C
                     AxB-C.Cod u (A.src f1, f2)) \C
                      (AxB-C.Map t (A.src f1, B.src f2) \C
                        AxB-C.Map u (A.src f1, B.src f2))
    proof −
      have C.joinable (AxB-C.Cod u (f1, B.src f2))
```

```
                    (AxB-C.Cod u (A.src f1, f2))
              using 1 Cod-u.preserves-joins C.joinable-def by blast
            moreover have AxB-C.Map t (A.src f1, B.src f2) \C
                            AxB-C.Map u (A.src f1, B.src f2) ⌢C
                          AxB-C.Cod u (f1, B.src f2) ⊔C
                          AxB-C.Cod u (A.src f1, f2)
              by (metis (no-types, lifting) 0 1
                  AxB.join-of-un-upto-cong AxB.prfx-implies-con
                  AxB.residuation-axioms AxB.src-char
                  C.join-is-join-of C.join-of-unique C.joinable-def
                  C.residuation-axioms Cod-u.preserves-joins
                  Map-Curry-Apex.preserves-arr f2 fst-conv
                  residuation.arrI residuation.arr-resid-iff-con
                  residuation.con-sym snd-conv)
              ultimately show ?thesis
                using C.resid-join_E(3) by simp
            qed
            also have ... = AxB-C.Cod u (f1, f2) \C
                              (AxB-C.Map t (A.src f1, B.src f2) \C
                                AxB-C.Map u (A.src f1, B.src f2))
              using 1 f1 f2 Cod-u.preserves-joins C.join-is-join-of
                    C.join-of-unique
              by (metis C.joinable-def)
            finally show ?thesis
              by (simp add: AxB.src-char f1 f2)
          qed
          ultimately show ?thesis
            using f1 f2 t u AxB-C.con-char BC.Map-resid BC.Apex-def
                  Cod-Curry Map-Curry
            by simp
        qed
        also have ... = BC.Map (?RHS f1) f2
          using assms f1 f2 CURRY-preserves-con CURRY-preserves-arr
                A-BC.Apex-def
          by simp
        finally show ?thesis by auto
      qed
    qed
  qed
  moreover have ?LHS f1 ≠ BC.Null
    by (simp add: f1 Curry-def)
  moreover have ?RHS f1 ≠ BC.Null
    using BC.arr-char f1 by blast
  ultimately show ?thesis
    by (metis BC.MkArr-Map)
qed
qed
qed
qed
```

sublocale *CURRY*: *simulation AxB-C.resid A-BC.resid CURRY*
proof
show $\bigwedge t.\ \neg$ *AxB-C.arr t* $\Longrightarrow$ *CURRY t* = *A-BC.null*
unfolding *CURRY-def*
by *auto*
fix *t u*
assume *con*: *AxB-C.con t u*
have *t*: *AxB-C.arr t*
using *AxB-C.con-implies-arr(1) con* **by** *blast*
have *u*: *AxB-C.arr u*
using *AxB-C.con-implies-arr(2) con* **by** *blast*
show *A-BC.con* (*CURRY t*) (*CURRY u*)
using *con CURRY-preserves-con* **by** *simp*
show *CURRY* (*t* $\setminus_{[AxB,C]}$ *u*) = *CURRY t* $\setminus_{[A,[B,C]]}$ *CURRY u*
proof −
have *1*: *AxB-C.Dom* (*t* $\setminus_{[AxB,C]}$ *u*) = *AxB-C.Cod u*
using *con AxB-C.con-char* **by** *auto*
have *2*: *AxB-C.Cod* (*t* $\setminus_{[AxB,C]}$ *u*) = *AxB-C.Apex t u*
using *con AxB-C.con-char* **by** *auto*
have *A-BC.Dom* (*CURRY* (*t* $\setminus_{[AxB,C]}$ *u*)) =
A-BC.Dom (*CURRY t* $\setminus_{[A,[B,C]]}$ *CURRY u*)
using *con 1 2 t u AxB-C.con-char*
apply *simp*
using ‹*CURRY t* $\frown_{[A,[B,C]]}$ *CURRY u*› **by** *force*
moreover
have *A-BC.Cod* (*CURRY* (*t* $\setminus_{[AxB,C]}$ *u*)) =
A-BC.Cod (*CURRY t* $\setminus_{[A,[B,C]]}$ *CURRY u*)
using *con 1 2 t u AxB-C.con-char Curry-preserves-Apex*
apply *simp*
using ‹*CURRY t* $\frown_{[A,[B,C]]}$ *CURRY u*› **by** *force*
moreover
have *A-BC.Map* (*CURRY* (*t* $\setminus_{[AxB,C]}$ *u*)) =
A-BC.Map (*CURRY t* $\setminus_{[A,[B,C]]}$ *CURRY u*)
(**is** *?LHS* = *?RHS*)
proof
fix *f1*
show *?LHS f1* = *?RHS f1*
proof (*cases A.arr f1*)
show ¬ *A.arr f1* $\Longrightarrow$ *?thesis*
by (*simp add*: ‹*A-BC.con* (*CURRY t*) (*CURRY u*)› *con Curry-def*)
assume *f1*: *A.arr f1*
have *BC.Dom* (*?LHS f1*) = *BC.Dom* (*?RHS f1*)
using *Dom-Map-CURRY-resid con f1* **by** *blast*
moreover
have *BC.Cod* (*?LHS f1*) = *BC.Cod* (*?RHS f1*)
using *Cod-Map-CURRY-resid Cod-Curry-Apex con*
Curry-preserves-Apex f1
by *force*

```
      moreover
      have BC.Map (?LHS f1) = BC.Map (?RHS f1)
        using f1 con Map-Map-CURRY-resid by blast
      moreover have ?LHS f1 ≠ A-BC.Null
        by (simp add: con f1 Curry-def)
      moreover have ?RHS f1 ≠ A-BC.Null
        by (metis A.arrE A-BC.arr-def A-BC.arr-resid A-BC.conE_ERTS
            BC.con-char ‹CURRY t ⌢[A,[B,C]] CURRY u› f1)
      ultimately show ?thesis
        by (metis BC.MkArr-Map)
    qed
  qed
  moreover have CURRY t \[A,[B,C]] CURRY u ≠ AxB-C.Null
    using A-BC.null-char ‹A-BC.con (CURRY t) (CURRY u)› by auto
  moreover have CURRY (t \[AxB,C] u) ≠ A-BC.Null
    using con A-BC.arr-char AxB-C.arr-resid CURRY-preserves-arr
    by presburger
  ultimately show ?thesis
    by (metis A-BC.MkArr-Map)
 qed
qed

lemma CURRY-is-simulation:
shows simulation AxB-C.resid A-BC.resid CURRY
 ..

definition UNCURRY
          :: ('a, ('b, 'c) BC.arr) A-BC.arr ⇒ ('a × 'b, 'c) AxB-C.arr
where UNCURRY f ≡ if A-BC.arr f
                  then AxB-C.MkArr (Uncurry (A-BC.Dom f))
                                   (Uncurry (A-BC.Cod f))
                                   (Uncurry (A-BC.Map f))
                  else AxB-C.null

lemma Dom-UNCURRY [simp]:
assumes A-BC.arr f
shows AxB-C.Dom (UNCURRY f) = Uncurry (A-BC.Dom f)
 using assms UNCURRY-def by auto

lemma Cod-UNCURRY [simp]:
assumes A-BC.arr f
shows AxB-C.Cod (UNCURRY f) = Uncurry (A-BC.Cod f)
 using assms UNCURRY-def by auto

lemma Map-UNCURRY [simp]:
assumes A-BC.arr f
shows AxB-C.Map (UNCURRY f) = Uncurry (A-BC.Map f)
 using assms UNCURRY-def by auto
```

```
lemma UNCURRY-CURRY [simp]:
assumes AxB-C.arr t
shows UNCURRY (CURRY t) = t
proof −
  interpret CURRY: simulation AxB-C.resid A-BC.resid CURRY
    using CURRY-is-simulation by simp
  have UNCURRY (CURRY t) =
       AxB-C.MkArr
         (Uncurry (A-BC.Dom (CURRY t)))
         (Uncurry (A-BC.Cod (CURRY t)))
         (Uncurry (A-BC.Map (CURRY t)))
    using assms UNCURRY-def by auto
  also have ... = AxB-C.MkArr
                    (Uncurry (Curry3 (AxB-C.Dom t)))
                    (Uncurry (Curry3 (AxB-C.Cod t)))
                    (Uncurry
                       (Curry (AxB-C.Dom t) (AxB-C.Cod t) (AxB-C.Map t)))
    using assms by simp
  also have ... =
            AxB-C.MkArr (AxB-C.Dom t) (AxB-C.Cod t) (AxB-C.Map t)
    by (metis AxB-C.arr-MkArr AxB-C.arr-char AxB-C.ide-MkIde
        AxB-C.ide-implies-arr Uncurry-Curry assms transformation-def)
  also have ... = t
    using assms AxB-C.MkArr-Map AxB-C.arr-char by auto
  finally show ?thesis by blast
qed

lemma CURRY-UNCURRY [simp]:
assumes A-BC.arr t
shows CURRY (UNCURRY t) = t
proof −
  have 1: AxB-C.arr (UNCURRY t)
  using assms AxB-C.arr-char UNCURRY-def Uncurry-preserves-transformations
    by fastforce
  have CURRY (UNCURRY t) =
       A-BC.MkArr
         (Curry3 (Uncurry (A-BC.Dom t)))
         (Curry3 (Uncurry (A-BC.Cod t)))
         (Curry (Uncurry (A-BC.Dom t)) (Uncurry (A-BC.Cod t))
               (Uncurry (A-BC.Map t)))
    using assms 1 CURRY-def by simp
  also have ... = A-BC.MkArr (A-BC.Dom t) (A-BC.Cod t) (A-BC.Map t)
    by (metis A-BC.arrE A-BC.arr-MkArr A-BC.arr-src-iff-arr
      A-BC.arr-trg-iff-arr A-BC.src-char A-BC.trg-char Curry-Uncurry assms)
  also have ... = t
    using assms A-BC.MkArr-Map A-BC.arr-char by force
  finally show ?thesis by blast
qed
```

```
lemma UNCURRY-is-simulation:
shows simulation A-BC.resid AxB-C.resid UNCURRY
proof
  show ⋀t. ¬ A-BC.arr t ⟹ UNCURRY t = AxB-C.null
    using UNCURRY-def by auto
  show ∗: ⋀t u. A-BC.con t u ⟹ AxB-C.con (UNCURRY t) (UNCURRY u)
  proof −
    fix t u
    assume con: A-BC.con t u
    have t: A-BC.arr t
      using con
      by (simp add: A-BC.con-implies-arr(1))
    have u: A-BC.arr u
      using con
      by (simp add: A-BC.con-implies-arr(2))
    show AxB-C.con (UNCURRY t) (UNCURRY u)
    proof (unfold AxB-C.con-char, intro conjI)
      show UNCURRY t ≠ AxB-C.Null
        using t UNCURRY-def by simp
      show UNCURRY u ≠ AxB-C.Null
        using u UNCURRY-def by simp
      show transformation AxB.resid C
            (AxB-C.Dom (UNCURRY t)) (AxB-C.Cod (UNCURRY t))
            (AxB-C.Map (UNCURRY t))
        using t A-BC.arr-char Uncurry-preserves-transformations by simp
      show transformation AxB.resid C
            (AxB-C.Dom (UNCURRY u)) (AxB-C.Cod (UNCURRY u))
            (AxB-C.Map (UNCURRY u))
        using u A-BC.arr-char Uncurry-preserves-transformations by simp
      show AxB-C.Dom (UNCURRY t) = AxB-C.Dom (UNCURRY u)
        using t u con A-BC.con-char by simp
      show ∀ ab. AxB.ide ab ⟶
                 AxB-C.Map (UNCURRY t) ab ⌢C
                 AxB-C.Map (UNCURRY u) ab
        using A-BC.con-char Uncurry-simp con t u by auto
    qed
  qed
  show ⋀t u. t ⌢[A,[B,C]] u ⟹
             UNCURRY (t \[A,[B,C]] u) =
             UNCURRY t \[AxB,C] UNCURRY u
  proof −
    fix t u
    assume con: A-BC.con t u
    have con': UNCURRY t ⌢[AxB,C] UNCURRY u
      using con ∗ by simp
    hence 1: CURRY (UNCURRY t \[AxB,C] UNCURRY u) =
          CURRY (UNCURRY t) \[A,[B,C]] CURRY (UNCURRY u)
      by auto
    also have 2: ... = t \[A,[B,C]] u
```

```
      using A-BC.con-implies-arr(1) A-BC.con-implies-arr(2) con by force
    also have ... = CURRY (UNCURRY (t \[A,[B,C]] u))
      by (simp add: con)
    finally have CURRY (UNCURRY t \[AxB,C] UNCURRY u) =
                 CURRY (UNCURRY (t \[A,[B,C]] u))
      by blast
    thus UNCURRY (t \[A,[B,C]] u) = UNCURRY t \[AxB,C] UNCURRY u
      using con
      by (metis AxB-C.residuation-axioms UNCURRY-CURRY 1 2 con'
          residuation.arr-resid)
  qed
qed

sublocale UNCURRY: simulation A-BC.resid AxB-C.resid UNCURRY
  using UNCURRY-is-simulation by blast

interpretation inverse-simulations
                 A-BC.resid AxB-C.resid CURRY UNCURRY
  using CURRY-UNCURRY UNCURRY-CURRY CURRY.extensional
        UNCURRY.extensional
  by unfold-locales auto

sublocale CURRY: invertible-simulation AxB-C.resid A-BC.resid CURRY
  ..

lemma invertible-simulation-CURRY:
shows invertible-simulation AxB-C.resid A-BC.resid CURRY
  ..

sublocale UNCURRY: invertible-simulation
                     A-BC.resid AxB-C.resid UNCURRY
  ..

lemma invertible-simulation-UNCURRY:
shows invertible-simulation A-BC.resid AxB-C.resid UNCURRY
  ..

sublocale inverse-simulations A-BC.resid AxB-C.resid CURRY UNCURRY
  ..

lemma inverse-simulations-CURRY-UNCURRY:
shows inverse-simulations A-BC.resid AxB-C.resid CURRY UNCURRY
  ..

end
```

3.10.7 Coextension of a Simulation

Here we define the coextension, of a simulation G from $X \times A$ to B, to a simulation F from X to $[A, B]$, and we prove that it is universal for the property $eval \circ (F \times A) = G$.

context *evaluation-map*
begin

abbreviation (*input*) *coext*
:: *'c resid* ⇒ (*'c* × *'a* ⇒ *'b*) ⇒ *'c* ⇒ (*'a*, *'b*) *AB.arr*
where *coext X G* ≡ *Currying.Curry3 X A B G*

lemma *Uncurry-simulation-expansion*:
assumes *weakly-extensional-rts X*
and *simulation X AB.resid F*
shows *Currying.Uncurry X A B F* =
map ∘ (*product-simulation.map X A F* (*I A*))
proof −
interpret *X*: *weakly-extensional-rts X*
using *assms*(*1*) **by** *blast*
interpret *XxA*: *product-rts X A* **..**
interpret *F*: *simulation X AB.resid F*
using *assms* **by** *blast*
interpret *Currying X A B* **..**
interpret *A*: *identity-simulation A* **..**
interpret *FxA*: *product-simulation X A AB.resid A F A.map* **..**
show *Uncurry F* = *map* ∘ *FxA.map*
using *Uncurry-def map-def FxA.map-def* **by** *auto*
qed

lemma *universality*:
assumes *weakly-extensional-rts X*
and *simulation* (*product-rts.resid X A*) *B G*
shows *simulation X AB.resid* (*Currying.Curry3 X A B G*)
and *Currying.Uncurry X A B* (*coext X G*) = *G*
and ∃!*F*. *simulation X AB.resid F* ∧ *Currying.Uncurry X A B F* = *G*
proof −
interpret *X*: *weakly-extensional-rts X*
using *assms*(*1*) **by** *blast*
interpret *XxA*: *product-rts X A* **..**
interpret *simulation XxA.resid B G*
using *assms* **by** *blast*
interpret *Currying X A B* **..**
interpret *A*: *identity-simulation A* **..**
let *?F* = *Curry G G G*
interpret *F*: *simulation X AB.resid ?F*
using *assms Curry-preserves-simulations* **by** *blast*
interpret *FxA*: *product-simulation X A AB.resid A ?F A.map* **..**
show *simulation X* ($\backslash_{[A,B]}$) *?F*

```
    using F.simulation-axioms by blast
  show Uncurry (coext X G) = G
    using Uncurry-simulation-expansion Uncurry-Curry Uncurry-def
          FxA.map-def Map-Curry map-def XxA.arr-char extensional
    by auto
  moreover
  have ⋀F'. simulation X AB.resid F' ∧ Uncurry F' = G ⟹ F' = ?F
  proof −
    fix F'
    assume F': simulation X AB.resid F' ∧ Uncurry F' = G
    interpret F': simulation X AB.resid F'
      using F' by blast
    interpret F': simulation-as-transformation X AB.resid F' ..
    interpret F'xA: product-simulation X A AB.resid A F' A.map ..
    have Uncurry F' = G
      using F' F'xA.map-def Uncurry-def Map-Curry map-def XxA.arr-char
        extensional
      by auto
    hence Curry (Uncurry F') (Uncurry F') (Uncurry F') = ?F
      by simp
    thus F' = ?F
      using Curry-Uncurry F'.transformation-axioms by simp
  qed
  ultimately show ∃!F. simulation X AB.resid F ∧ Uncurry F = G
    using F.simulation-axioms
    by (metis (no-types, lifting))
qed

lemma comp-coext-simulation:
assumes weakly-extensional-rts X and weakly-extensional-rts X'
and simulation X X' G
and simulation (product-rts.resid X' A) B H
shows coext X' H ∘ G = coext X (H ∘ product-simulation.map X A G (I A))
proof −
  interpret X: weakly-extensional-rts X
    using assms(1) by blast
  interpret X': weakly-extensional-rts X'
    using assms(2) by blast
  interpret XxA: product-rts X A ..
  interpret X'xA: product-rts X' A ..
  interpret G: simulation X X' G
    using assms(3) by blast
  interpret H: simulation X'xA.resid B H
    using assms(4) by blast
  interpret A: identity-simulation A ..
  interpret GxA: product-simulation X A X' A G A.map ..
  interpret coext-H: simulation X' AB.resid ‹coext X' H›
    using universality H.simulation-axioms X'.weakly-extensional-rts-axioms
    by auto
```

```
    interpret coext-H-o-G: simulation X AB.resid ‹coext X' H ∘ G›
      using universality(1) [of X' H] simulation-comp G.simulation-axioms
        H.simulation-axioms X'.weakly-extensional-rts-axioms
      by auto
    interpret coext-H-o-G-x-A: product-simulation X A AB.resid A
                               ‹coext X' H ∘ G› A.map ..

    have coext X' H ∘ G = coext X (map ∘ coext-H-o-G-x-A.map)
      using universality(1−3) X.weakly-extensional-rts-axioms
        coext-H-o-G.simulation-axioms coext-H-o-G-x-A.simulation-axioms
        simulation-axioms simulation-comp [of - - coext-H-o-G-x-A.map - map]
        Uncurry-simulation-expansion
      by (metis (no-types, lifting))
    also have ... = coext X
                    (map ∘ (product-simulation.map X' A (coext X' H) A.map
                            ∘ GxA.map))
      using simulation-interchange
            [of X X' G A A A.map AB.resid coext X' H A A.map]
          G.simulation-axioms A.simulation-axioms coext-H.simulation-axioms
          comp-simulation-identity [of A A A.map] A.simulation-axioms
        by simp
    also have ... = coext X
                    (map ∘ product-simulation.map X' A (coext X' H) A.map
                           ∘ GxA.map)
      using fun.map-comp by metis
    also have ... = coext X (H ∘ GxA.map)
     using H.simulation-axioms X'.weakly-extensional-rts-axioms universality(2)
           Uncurry-simulation-expansion coext-H.simulation-axioms
      by fastforce
    finally have coext X' H ∘ G = coext X (H ∘ GxA.map) by blast
    thus ?thesis by fastforce
  qed

end


locale evaluation-map-between-extensional-rts =
  evaluation-map +
A: extensional-rts A
begin

  lemma Uncurry-transformation-expansion:
  assumes weakly-extensional-rts X
  and transformation X AB.resid F G T
  shows Currying.Uncurry X A B T =
        map ∘ product-transformation.map X A AB.resid A F (I A) T (I A)
  proof −
    interpret X: weakly-extensional-rts X
      using assms(1) by blast
```

```
  interpret XxA: product-rts X A ..
  interpret F: simulation X AB.resid F
    using assms transformation-def by blast
  interpret G: simulation X AB.resid G
    using assms transformation-def by blast
  interpret T: transformation X AB.resid F G T
    using assms by blast
  interpret Currying X A B ..
  interpret Uncurry-F: simulation XxA.resid B ‹Uncurry F›
    using F.simulation-axioms Uncurry-preserves-simulations by blast
  interpret Uncurry-G: simulation XxA.resid B ‹Uncurry G›
    using G.simulation-axioms Uncurry-preserves-simulations by blast
  interpret Uncurry-T: transformation
                       XxA.resid B ‹Uncurry F› ‹Uncurry G› ‹Uncurry T›
    using T.transformation-axioms Uncurry-preserves-transformations by blast
  interpret A: identity-simulation A ..
  interpret A: simulation-as-transformation A A A.map ..
  interpret TxA: product-transformation
                 X A AB.resid A F A.map G A.map T A.map ..
  show Uncurry T = map ∘ TxA.map
  proof (intro transformation-eqI
                 [of XxA.resid B Uncurry F Uncurry G Uncurry T
                     map ∘ TxA.map])
    show transformation XxA.resid B (Uncurry F) (Uncurry G) (Uncurry T)
      using Uncurry-T.transformation-axioms by blast
    show transformation
           XxA.resid B (Uncurry F) (Uncurry G) (map ∘ TxA.map)
      using simulation-axioms F.simulation-axioms G.simulation-axioms
            TxA.transformation-axioms
            transformation-whisker-left
              [of XxA.resid ABxA.resid TxA.F1xF0.map TxA.G1xG0.map
                  TxA.map B map]
            X.weakly-extensional-rts-axioms B.weakly-extensional-rts-axioms
            Uncurry-simulation-expansion [of X F]
            Uncurry-simulation-expansion [of X G]
      by presburger
    show extensional-rts B
      using B.extensional-rts-axioms by blast
    fix xa
    assume xa: XxA.ide xa
    show Uncurry T xa = (map ∘ TxA.map) xa
      using xa Uncurry-def map-def TxA.map-def by auto
  qed
qed

lemma universality2:
assumes weakly-extensional-rts X
and transformation (product-rts.resid X A) B F G T
shows transformation X AB.resid
```

```
        (Currying.Curry3 X A B F) (Currying.Curry3 X A B G)
        (Currying.Curry X A B F G T)
and Currying.Uncurry X A B (Currying.Curry X A B F G T) = T
and ∃!T'. transformation X AB.resid
           (Currying.Curry3 X A B F) (Currying.Curry3 X A B G)
           T' ∧
         Currying.Uncurry X A B T' = T
proof −
  interpret X: weakly-extensional-rts X
    using assms(1) by blast
  interpret XxA: product-rts X A ..
  interpret XxA: product-of-weakly-extensional-rts X A ..
  interpret F: simulation XxA.resid B F
    using assms transformation-def by blast
  interpret F: simulation-as-transformation XxA.resid B F ..
  interpret G: simulation XxA.resid B G
    using assms transformation-def by blast
  interpret G: simulation-as-transformation XxA.resid B G ..
  interpret T: transformation XxA.resid B F G T
    using assms transformation-def by blast
  interpret T: transformation-to-extensional-rts XxA.resid B F G T ..
  interpret Currying X A B ..
  interpret A: identity-simulation A ..

  let ?F' = Curry3 F
  interpret F': simulation X AB.resid ?F'
    using F.simulation-axioms Curry-preserves-simulations by blast
  let ?G' = Curry3 G
  interpret G': simulation X AB.resid ?G'
    using G.simulation-axioms Curry-preserves-simulations by blast
  let ?T' = Curry F G T
  interpret T': transformation X AB.resid ?F' ?G' ?T'
    using T.transformation-axioms Curry-preserves-transformations
    by blast

  interpret ABxA: product-of-extensional-rts AB.resid A ..
  interpret BxA: product-of-extensional-rts B A ..

  interpret IA: identity-simulation A ..
  interpret IA: simulation-as-transformation A A A.map ..
  interpret T'xA: product-transformation
                  X A AB.resid A ?F' ‹I A› ?G' ‹I A› ?T' ‹I A› ..

  show transformation X AB.resid ?F' ?G' ?T' ..
  moreover show Uncurry (Curry F G T) = T
  proof −
    have map ∘ T'xA.map = T
    proof
      fix t
```

```
show (map ∘ T'xA.map) t = T t
proof (cases XxA.arr t)
  show ¬ XxA.arr t ⟹ ?thesis
    using T.extensional T'xA.extensional extensional by simp
  assume t: XxA.arr t
  have 0: X.arr (fst (XxA.src t))
    using t XxA.arr-src-iff-arr by blast
  have 1: A.arr (snd (XxA.src t))
    using t XxA.arr-src-iff-arr by blast
  have 2: AB.arr (fst (ABxA.join
                       (Curry F G T (fst (XxA.src t)),
                                     snd (XxA.src t))
                       (T'xA.F1xF0.map t)))
    using t 0 1 ABxA.arr-char BxA.joinable-iff-arr-join
          T'xA.TC.joinable T'xA.map-simp T'xA.preserves-arr
    by presburger
  have 3: A.arr (snd (ABxA.join
                       (Curry F G T (fst (XxA.src t)),
                                     snd (XxA.src t))
                       (T'xA.F1xF0.map t)))
    using t 0 1 ABxA.arr-char BxA.joinable-iff-arr-join
          T'xA.TC.joinable T'xA.map-simp T'xA.preserves-arr
    by presburger

  have ∗: (map ∘ T'xA.map) t =
          AB.Map
            (fst (ABxA.join
                    (Curry F G T (X.src (fst t)), A.src (snd t))
                    (Curry F F F (fst t), snd t)))
            (snd (ABxA.join
                    (Curry F G T (X.src (fst t)), A.src (snd t))
                    (Curry F F F (fst t), snd t)))
    unfolding map-def T'xA.map-simp
    using t 0 1 2 3 XxA.src-char trg-Curry
          T'xA.F1xF0.map-simp [of fst t snd t]
    by simp

  have 6: ABxA.joinable
            (Curry F G T (X.src (fst t)), A.src (snd t))
            (Curry F F F (fst t), snd t)
  proof −
    have AB.joinable (Curry F G T (X.src (fst t))) (Curry3 F (fst t))
    proof −
      have AB.arr (Curry F G T (X.src (fst t)))
        using t T'.preserves-arr X.arr-src-if-arr by blast
      moreover have AB.arr (Curry F F F (fst t))
        using t AB.joinable-def T'.naturality3 by simp
      moreover have AB.Dom (Curry F G T (X.src (fst t))) =
                    AB.Dom (Curry3 F (fst t))
```

```
  using t Dom-Curry by auto
moreover
have ⋀u. A.arr u
           ⟹ B.joinable
               (AB.Map (Curry F G T (X.src (fst t))) (A.src u) ⊔B
                  AB.Map (Curry3 F (fst t)) (A.src u))
                (AB.Dom (Curry F G T (X.src (fst t))) u)
proof −
  fix u
  assume u: A.arr u
  have AB.Map (Curry F G T (X.src (fst t))) (A.src u) ≲B
       AB.Map (Curry F G T (fst t)) u
    using t u Map-Curry B.composite-of-def T.naturality2'
          XxA.src-char
    by auto
  moreover have AB.Map (Curry3 F (fst t)) (A.src u) ≲B
                AB.Map (Curry F G T (fst t)) u
  proof −
    have F (fst t, A.src u) ≲B T (fst t, u)
    proof −
      have 1: F (fst t, A.src u) \B T (fst t, u) =
              F (fst t, A.src u) \B
                (T (X.src (fst t), A.src u) ⊔B F (fst t, u))
        using t u T.naturality3 [of (fst t, u)] XxA.arr-char
              XxA.src-char B.join-is-join-of B.join-of-unique
              B.joinable-def
        apply auto[1]
        by metis
      also have ... = (F (fst t, A.src u) \B F (fst t, u)) \B
                      (T (X.src (fst t), A.src u) \B F (fst t, u))
      proof −
        have B.joinable
               (F (fst t, u))
               (T (X.src (fst t), A.src u))
          using t u XxA.arr-char
          by (metis B.join-sym B.joinable-iff-join-not-null
              B.not-arr-null T.naturality3'E(1) T.preserves-arr
              XxA.src-char fst-conv snd-conv)
        moreover have F (fst t, A.src u) ⌢B
                      F (fst t, u) ⊔B T (X.src (fst t), A.src u)
          using t u 1
          by (metis A.con-arr-src(1) B.conE B.conI B.con-sym
              B.join-sym T.preserves-con(2) XxA.con-arr-self
              XxA.con-char fst-conv snd-conv)
        ultimately show ?thesis
          using t u B.resid-joinE(2) B.join-sym by simp
      qed
      also have ... = F (X.trg (fst t), A.trg u) \B
                      (T (X.src (fst t), A.src u) \B F (fst t, u))
```

```
        using t u XxA.con-char XxA.resid-def X.trg-def
              F.preserves-resid [of (fst t, A.src u) (fst t, u)]

        by simp
      moreover have B.ide ...
      proof −
        have B.ide (F (X.trg (fst t), A.trg u))
          using t u F.preserves-ide XxA.ide-char by simp
        moreover have B.coinitial
                        (F (X.trg (fst t), A.trg u))
                        (T (X.src (fst t), A.src u) \_B F (fst t, u))
        proof
          show B.arr (F (X.trg (fst t), A.trg u))
            using t u F.preserves-reflects-arr by simp
          show B.src (F (X.trg (fst t), A.trg u)) =
               B.src (T (X.src (fst t), A.src u) \_B F (fst t, u))
            using t u XxA.arr-char [of (fst t, u)]
            by (metis B.src-ide T.naturality1 T.preserves-src
                XxA.arr-char XxA.ide-trg XxA.src-char XxA.trg-char
                calculation fst-conv snd-conv)
        qed
        ultimately show ?thesis
          using t u B.resid-ide(2) by auto
      qed
      ultimately show ?thesis by simp
    qed
    thus ?thesis
      using t u Map-Curry by auto
  qed
  moreover have AB.Dom (Curry F G T (X.src (fst t))) u ≲B
                 AB.Map (Curry F G T (fst t)) u
    using t u Dom-Curry Map-Curry
    apply auto[1]
    using A.trg-def T.general-naturality(2) XxA.ide-trg
          XxA.resid-def XxA.trg-char
    by force
  ultimately show B.joinable
                    (AB.Map (Curry F G T (X.src (fst t))) (A.src u) ⊔B
                       AB.Map (Curry F F F (fst t)) (A.src u))
                    (AB.Dom (Curry F G T (X.src (fst t))) u)
    by (meson A.weakly-extensional-rts-axioms AB.joinable-def
        B.extensional-rts-axioms T'.naturality3 XxA.arr-char
        exponential-rts.intro exponential-rts.join-char(1) t u)
qed
ultimately show ?thesis
  unfolding AB.join-char(1)
  using t Map-Curry Dom-Curry
  by (intro allI impI conjI) auto
qed
```

```
  moreover have A.joinable (A.src (snd t)) (snd t)
    using t A.join-src A.join-sym A.joinable-iff-join-not-null by force
  ultimately show ?thesis
    using ABxA.join-of-char(2) by auto
qed
hence ABxA.join
        (Curry F G T (X.src (fst t)), A.src (snd t))
        (Curry3 F (fst t), snd t) =
      (AB.join (Curry F G T (X.src (fst t))) (Curry3 F (fst t)),
       A.join (A.src (snd t)) (snd t))
  using t 0 1 2 3 ABxA.join-simp by auto
also have ... = (AB.join
                   (Curry F G T (X.src (fst t)))
                   (Curry3 F (fst t)), snd t)
  using t A.join-src A.join-sym XxA.src-char by simp
also have ... = (Curry F G T (fst t), snd t)
proof −
  have AB.join
         (Curry F G T (X.src (fst t)))
         (Curry3 F (fst t)) =
       Curry F G T (fst t)
  proof (intro AB.arr-eqI)
    show 4: AB.arr
             (AB.join
                (Curry F G T (X.src (fst t)))
                (Curry3 F (fst t)))
      using t 2 T'xA.F1xF0.map-def XxA.src-char calculation
      by auto
    show 5: AB.arr (Curry F G T (fst t))
      using t T'.preserves-arr by blast
    show AB.Dom
           (AB.join
              (Curry F G T (X.src (fst t)))
              (Curry3 F (fst t))) =
         AB.Dom (Curry F G T (fst t))
      using t Dom-Curry
      by (metis 6 AB.join-is-join-of AB.join-of-unique
          ABxA.join-of-char(2) T'.naturality3 XxA.arr-char fst-conv)
    show AB.Cod
           (AB.join
              (Curry F G T (X.src (fst t))) (Curry3 F (fst t))) =
         AB.Cod (Curry F G T (fst t))
      using t Cod-Curry
      by (metis 6 AB.join-is-join-of AB.join-of-unique
          ABxA.join-of-char(2) T'.naturality3 XxA.arr-char fst-conv)
    show ⋀a. A.ide a ⟹
                AB.Map
                  (AB.join
                     (Curry F G T (X.src (fst t))))
```

```
                    (Curry3 F (fst t)))
                  a =
                 AB.Map (Curry F G T (fst t)) a
          using t Map-Curry AB.Map-join
          apply auto[1]
          by (metis (no-types, lifting) 4 AB.join-is-join-of
              AB.join-of-unique AB.joinable-iff-arr-join T'.naturality3)
        qed
        thus ?thesis by simp
      qed
      finally have ABxA.join
                   (Curry F G T (X.src (fst t)), A.src (snd t))
                   (Curry3 F (fst t), snd t) =
                  (Curry F G T (fst t), snd t)
        by blast
      hence (map ∘ T'xA.map) t = AB.Map (Curry F G T (fst t)) (snd t)
        using ∗ by auto
      also have ... = T t
        using t Map-Curry by simp
      finally show (map ∘ T'xA.map) t = T t by blast
    qed
  qed
  thus ?thesis
    using Uncurry-transformation-expansion [of X ?F' ?G' ?T']
          X.weakly-extensional-rts-axioms T.transformation-axioms
          Uncurry-Curry
    by blast
qed
moreover
have ⋀T''. ⟦transformation X (\[A,B]) (coext X F) (coext X G) T'';
             Uncurry T'' = T⟧
               ⟹ T'' = ?T'
proof −
  fix T''
  assume T'': transformation X (\[A,B]) (coext X F) (coext X G) T''
  assume 1: Uncurry T'' = T
  interpret T'': transformation X AB.resid ‹coext X F› ‹coext X G› T''
    using T'' by blast
  interpret T''xA: product-transformation X A AB.resid A
                   ‹coext X F› ‹I A› ‹coext X G› ‹I A› T'' ‹I A› ..
  show T'' = ?T'
  proof (intro transformation-eqI)
    show transformation X (\[A,B]) (coext X F) (coext X G) T''
      by fact
    show transformation X (\[A,B]) (coext X F) (coext X G) ?T'
      ..
    show extensional-rts AB.resid
      ..
    fix x
```

```
    assume x: X.ide x
    show T'' x = Curry F G T x
    proof (intro AB.arr-eqI)
      show AB.arr (T'' x)
        using x T''.preserves-arr by blast
      show AB.arr (Curry F G T x)
        using x T'.preserves-arr by blast
      show AB.Dom (T'' x) = AB.Dom (Curry F G T x)
        unfolding Curry-def Uncurry-def
        using x map-def
        apply auto[1]
        by (metis (no-types, opaque-lifting) AB.Map-src Map-Curry
            T''.preserves-src X.ide-implies-arr ‹AB.arr (T'' x)›)
      show AB.Cod (T'' x) = AB.Cod (Curry F G T x)
        unfolding Curry-def Uncurry-def
        using x map-def
        apply auto[1]
        by (metis (no-types, opaque-lifting) AB.Map-trg Map-Curry
            T''.preserves-trg X.ide-implies-arr ‹AB.arr (T'' x)›)
      fix a
      assume a: A.ide a
      have AB.Map (T'' x) a = T (x, a)
      proof -
        have AB.Map (T'' x) a = AB.Map (Curry F G (Uncurry T'') x) a
          using T'' Curry-Uncurry [of coext X F coext X G T'']
                Uncurry-Curry F.transformation-axioms
                G.transformation-axioms
          by simp
        also have ... = AB.Map (Curry F G T x) a
          unfolding Curry-def Uncurry-def
          using a x 1 map-def T''xA.map-simp-ide
                T''.transformation-axioms X.weakly-extensional-rts-axioms
                Uncurry-transformation-expansion
                  [of X Curry3 F Curry3 G T'']
          by auto
        also have ... = T (x, a)
          unfolding Curry-def
          using a x by simp
        finally show ?thesis by simp
      qed
      thus AB.Map (T'' x) a = AB.Map (Curry F G T x) a
        unfolding Curry-def Uncurry-def
        using x a map-def by auto
    qed
  qed
qed
ultimately
show ∃!T'. transformation X (\[A,B]) (Curry3 F) (Curry3 G) T' ∧
           Uncurry T' = T
```

by *blast*
qed

end

3.10.8 Compositors

For any RTS's A, B, and C, there exists a "compositor" simulation from the product of exponential RTS's $[B,\ C] \times [A,\ B]$ to the exponential RTS $[A,\ C]$.

locale *COMP* =
A: *extensional-rts A* +
B: *extensional-rts B* +
C: *extensional-rts C*
for *A* :: *'a resid*
and *B* :: *'b resid*
and *C* :: *'c resid*
begin

sublocale *AC*: *exponential-rts A C* ..
sublocale *AB*: *exponential-rts A B* ..
sublocale *BC*: *exponential-rts B C* ..

sublocale *BCxAB*: *product-rts BC.resid AB.resid* ..
sublocale *BCxAB*: *product-of-extensional-rts BC.resid AB.resid* ..

interpretation *AB*: *identity-simulation AB.resid* ..
interpretation *BC*: *identity-simulation BC.resid* ..

interpretation *ABxA*: *product-rts AB.resid A* ..
interpretation *BCxB*: *product-rts BC.resid B* ..
interpretation *BCxAB*: *identity-simulation BCxAB.resid* ..
interpretation *BCxAB-x-A*: *product-rts BCxAB.resid A* ..
interpretation *BC-x-ABxA*: *product-rts BC.resid ABxA.resid* ..

interpretation *E-AB*: *RTSConstructions.evaluation-map A B* ..
interpretation *E-BC*: *RTSConstructions.evaluation-map B C* ..
interpretation *BCxE-AB*: *product-simulation*
BC.resid ABxA.resid BC.resid B
BC.map E-AB.map ..

interpretation *ASSOC-BC-AB-A*: *ASSOC BC.resid AB.resid A* ..
sublocale *Currying*: *Currying BCxAB.resid A C* ..

The following definition is expressed in a form that makes it evident that it defines a simulation.

definition *map* :: *('b, 'c) BC.arr* × *('a, 'b) AB.arr* ⇒ *('a, 'c) AC.arr*
where *map* = (λ*F*. *Currying.Curry3 F*)
(*E-BC.map* ∘ *BCxE-AB.map* ∘ *ASSOC-BC-AB-A.map*)

```
sublocale simulation BCxAB.resid AC.resid map
  unfolding map-def
  using E-BC.simulation-axioms BCxE-AB.simulation-axioms
        ASSOC-BC-AB-A.simulation-axioms simulation-comp
  by auto

lemma is-simulation:
shows simulation BCxAB.resid AC.resid map
  ..

sublocale binary-simulation BC.resid AB.resid AC.resid map ..

lemma is-binary-simulation:
shows binary-simulation BC.resid AB.resid AC.resid map
  ..

sublocale E-BC-o-BCxE-AB: composite-simulation
                          BC-x-ABxA.resid BCxB.resid C
                          BCxE-AB.map E-BC.map
  ..
```

The following explicit formula is more useful for calculations. There is a bit of work involved to show that the two versions are equal, but notice that as a consequence we obtain a proof that the explicit formula actually defines a simulation. A similar amount of work would be required to show this directly.

```
lemma map-eq:
shows map = (λgf. if BCxAB.arr gf
                  then AC.MkArr
                         (BC.Dom (fst gf) ∘ AB.Dom (snd gf))
                         (BC.Cod (fst gf) ∘ AB.Cod (snd gf))
                         (BC.Map (fst gf) ∘ AB.Map (snd gf))
                  else AC.Null)
proof
  fix gf
  show map gf = (if BCxAB.arr gf
                 then AC.MkArr
                        (BC.Dom (fst gf) ∘ AB.Dom (snd gf))
                        (BC.Cod (fst gf) ∘ AB.Cod (snd gf))
                        (BC.Map (fst gf) ∘ AB.Map (snd gf))
                 else AC.Null)
  proof (cases BCxAB.arr gf)
    show ¬ BCxAB.arr gf ⟹ ?thesis
      using map-def Currying.Curry-def AC.null-char by presburger
    assume gf: BCxAB.arr gf
    have AC.Dom (map gf) = BC.Dom (fst gf) ∘ AB.Dom (snd gf)
    proof −
      have AC.Dom (map gf) =
```

```
    (λt. E-BC.map
          (BCxE-AB.map (ASSOC-BC-AB-A.map (BCxAB.src gf, t))))
  using gf Currying.Dom-Curry map-def by simp
also have ... = BC.Dom (fst gf) ∘ AB.Dom (snd gf)
proof
  fix t
  have E-BC.map
         (BCxE-AB.map (ASSOC-BC-AB-A.map (BCxAB.src gf, t))) =
       (if A.arr t
        then E-BC.map
             (BCxE-AB.map (fst (BCxAB.src gf), snd (BCxAB.src gf), t))
        else C.null)
    using gf ASSOC-BC-AB-A.map-def BCxAB.arr-src-iff-arr
      BCxAB.extensional
    by (metis (no-types, lifting) ASSOC-BC-AB-A.map-eq
      ASSOC-BC-AB-A.preserves-reflects-arr BCxAB-x-A.arr-char
      BCxE-AB.preserves-reflects-arr E-BC.extensional
      fst-conv snd-conv)
  also have ... = E-BC.map
                    (BCxE-AB.map
                      (fst (BCxAB.src gf),
                       snd (BCxAB.src gf), t))
    using BCxE-AB.map-def E-BC.extensional by fastforce
  also have ... = E-BC.map
                    (BCxE-AB.map
                      (BC.src (fst gf), AB.src (snd gf), t))
    by (metis (no-types, lifting) AB.src-eqI BC.src-eqI
      BCxAB.con-arr-src(2) BCxAB.con-char BCxAB.ide-char
      BCxAB.ide-src gf)
  also have ... = E-BC.map
                    (BC.src (fst gf),
                     E-AB.map (AB.src (snd gf), t))
    using gf AB.src-simp BCxE-AB.map-simp BCxE-AB.map-def
      E-BC.extensional
    by simp
  also have ... = E-BC.map (BC.src (fst gf), AB.Dom (snd gf) t)
    using gf E-AB.extensional
          E-AB.map-simp [of (AB.src (snd gf), t)]
          AB.src-simp [of snd gf]
    apply auto[1]
    by (metis (no-types, lifting) AB.arr-MkArr AB.arr-src-iff-arr
      transformation.extensional)
  also have ... = BC.Dom (fst gf) (AB.Dom (snd gf) t)
    using gf BC.src-simp E-BC.map-simp E-BC.extensional
    apply auto[1]
    by (metis (mono-tags, lifting) BC.Map.simps(1) BC.arr-MkArr
      BC.arr-src-iff-arr transformation.extensional)
  also have ... = (BC.Dom (fst gf) ∘ AB.Dom (snd gf)) t
    by simp
```

```
    finally show E-BC.map
                  (BCxE-AB.map
                    (ASSOC-BC-AB-A.map (BCxAB.src gf, t))) =
                 (BC.Dom (fst gf) ∘ AB.Dom (snd gf)) t
      by blast
  qed
  finally show ?thesis by blast
qed
moreover have AC.Cod (map gf) = BC.Cod (fst gf) ∘ AB.Cod (snd gf)
proof −
  have AC.Cod (map gf) =
       (λt. E-BC.map
              (BCxE-AB.map (ASSOC-BC-AB-A.map (BCxAB.trg gf, t))))
    using gf Currying.Cod-Curry map-def by simp
  also have ... = BC.Cod (fst gf) ∘ AB.Cod (snd gf)
  proof
    fix t
    have E-BC.map
           (BCxE-AB.map (ASSOC-BC-AB-A.map (BCxAB.trg gf, t))) =
         (if A.arr t
          then E-BC.map
                 (BCxE-AB.map
                   (fst (BCxAB.trg gf), snd (BCxAB.trg gf), t))
          else C.null)
      using gf ASSOC-BC-AB-A.map-def BCxAB.arr-trg-iff-arr
        BCxAB.extensional
      by (metis (no-types, lifting) ASSOC-BC-AB-A.map-eq
          ASSOC-BC-AB-A.preserves-reflects-arr BCxAB-x-A.arr-char
          BCxE-AB.preserves-reflects-arr E-BC.extensional
          fst-conv snd-conv)
    also have ... = E-BC.map
                      (BCxE-AB.map
                        (fst (BCxAB.trg gf), snd (BCxAB.trg gf), t))
      using BCxE-AB.map-def E-BC.extensional by fastforce
    also have ... = E-BC.map
                      (BCxE-AB.map
                        (BC.trg (fst gf), AB.trg (snd gf), t))
      using BCxAB.trg-char gf by auto
    also have ... = E-BC.map
                      (BC.trg (fst gf), E-AB.map (AB.trg (snd gf), t))
      using gf AB.trg-simp BCxE-AB.map-simp
      by (simp add: BCxE-AB.map-def E-BC.extensional)
    also have ... = E-BC.map (BC.trg (fst gf), AB.Cod (snd gf) t)
      using gf E-AB.extensional E-AB.map-simp [of (AB.trg (snd gf), t)]
            AB.trg-simp [of snd gf]
      apply auto[1]
      by (metis (no-types, lifting) AB.arr-MkArr AB.arr-trg-iff-arr
          transformation.extensional)
    also have ... = BC.Cod (fst gf) (AB.Cod (snd gf) t)
```

```
  using gf BC.trg-simp BCxE-AB.map-simp E-BC.map-simp E-BC.extensional
    apply auto[1]
    by (metis (mono-tags, lifting) BC.Map.simps(1) BC.arr-MkArr
        BC.arr-trg-iff-arr transformation.extensional)
  also have ... = (BC.Cod (fst gf) ∘ AB.Cod (snd gf)) t
    by simp
  finally show E-BC.map
                 (BCxE-AB.map
                    (ASSOC-BC-AB-A.map (BCxAB.trg gf, t))) =
               (BC.Cod (fst gf) ∘ AB.Cod (snd gf)) t
    by blast
  qed
  finally show ?thesis by blast
qed
moreover have AC.Map (map gf) = BC.Map (fst gf) ∘ AB.Map (snd gf)
proof −
  have AC.Map (map gf) =
       (λt. E-BC.map (BCxE-AB.map (ASSOC-BC-AB-A.map (gf, t))))
    using gf Currying.Map-Curry map-def by simp
  also have ... = BC.Map (fst gf) ∘ AB.Map (snd gf)
  proof
    fix t
    have E-BC.map (BCxE-AB.map (ASSOC-BC-AB-A.map (gf, t))) =
         (if A.arr t
          then E-BC.map (BCxE-AB.map (fst gf, snd gf, t))
          else C.null)
      using gf ASSOC-BC-AB-A.map-def
      by (metis (no-types, lifting) ASSOC-BC-AB-A.map-eq
          ASSOC-BC-AB-A.preserves-reflects-arr BCxAB-x-A.arr-char
         BCxE-AB.preserves-reflects-arr E-BC.extensional fst-conv snd-conv)
    also have ... = E-BC.map (BCxE-AB.map (fst gf, snd gf, t))
      using BCxE-AB.map-def E-BC-o-BCxE-AB.extensional by force
    also have ... = BC.Map (fst gf) (AB.Map (snd gf) t)
     using gf E-AB.map-def E-BC.map-def BCxE-AB.map-def AB.arr-char
       BC.arr-char
      apply (auto simp add: transformation.preserves-arr)[1]
      by (simp add: transformation-axioms-def transformation-def)
    also have ... = (BC.Map (fst gf) ∘ AB.Map (snd gf)) t
      by simp
    finally show E-BC.map
                   (BCxE-AB.map (ASSOC-BC-AB-A.map (gf, t))) =
                 (BC.Map (fst gf) ∘ AB.Map (snd gf)) t
      by blast
  qed
  finally show ?thesis by blast
qed
ultimately show ?thesis
  using gf Currying.Curry-simp map-def
  by (simp add: Currying.Curry-def AC.null-char)
```

qed
qed

end

3.10.9 Functoriality of Exponential

Here we show that the covariant and contravariant exponential RTS constructions are "meta-functorial": they preserve identity simulations and compositions of simulations. We say "meta-functorial", rather than "functorial", because we do not have formal categories to serve as the domain and codomain for these constructions.

abbreviation *cov-Exp*
:: *'c resid* ⇒ *'a resid* ⇒ *'b resid* ⇒ (*'a* ⇒ *'b*)
⇒ (*'c*, *'a*) *exponential-rts.arr* ⇒ (*'c*, *'b*) *exponential-rts.arr* (*Exp*→)
where *Exp*→ *X B C* ≡
λ*G t2*. *COMP.map X B C* (*exponential-rts.MkIde G*, *t2*)

abbreviation *cnt-Exp*
:: *'a resid* ⇒ *'b resid* ⇒ *'c resid* ⇒ (*'a* ⇒ *'b*)
⇒ (*'b*, *'c*) *exponential-rts.arr* ⇒ (*'a*, *'c*) *exponential-rts.arr* (*Exp*←)
where *Exp*← *A B X* ≡
λ*F t1*. *COMP.map A B X* (*t1*, *exponential-rts.MkIde F*)

lemma *cov-Exp-eq*:
assumes *extensional-rts X* **and** *extensional-rts B* **and** *extensional-rts C*
shows *Exp*→ *X B C G* =
(λ*t2*. *if simulation B C G* ∧ *residuation.arr* (*exponential-rts.resid X B*) *t2*
then exponential-rts.MkArr
(*G* ∘ *exponential-rts.Dom t2*)
(*G* ∘ *exponential-rts.Cod t2*)
(*G* ∘ *exponential-rts.Map t2*)
else exponential-rts.Null)
proof
interpret *X*: *extensional-rts X*
using *assms*(*1*) **by** *blast*
interpret *B*: *extensional-rts B*
using *assms*(*2*) **by** *blast*
interpret *C*: *extensional-rts C*
using *assms*(*3*) **by** *blast*
interpret *XB*: *exponential-rts X B* **..**
interpret *XC*: *exponential-rts X C* **..**
interpret *COMP*: *COMP X B C* **..**
fix *t2*
show *Exp*→ *X B C G t2* =
(*if simulation B C G* ∧ *XB.arr t2*
then XC.MkArr (*G* ∘ *XB.Dom t2*) (*G* ∘ *XB.Cod t2*) (*G* ∘ *XB.Map t2*)
else XC.Null)

```
  using COMP.map-eq apply simp
  using COMP.BC.ide-implies-arr transformation.axioms(4) by blast
qed

lemma cnt-Exp-eq:
assumes extensional-rts X and extensional-rts A and extensional-rts B
shows Exp⁻ A B X F =
      (λt1. if residuation.arr (exponential-rts.resid B X) t1 ∧ simulation A B F
            then exponential-rts.MkArr
                 (exponential-rts.Dom t1 ∘ F)
                 (exponential-rts.Cod t1 ∘ F)
                 (exponential-rts.Map t1 ∘ F)
            else exponential-rts.Null)
proof
  interpret X: extensional-rts X
    using assms(1) by blast
  interpret A: extensional-rts A
    using assms(2) by blast
  interpret B: extensional-rts B
    using assms(3) by blast
  interpret AX: exponential-rts A X ..
  interpret BX: exponential-rts B X ..
  interpret COMP: COMP A B X ..
  fix t1
  show Exp⁻ A B X F t1 =
       (if BX.arr t1 ∧ simulation A B F
        then AX.MkArr (BX.Dom t1 ∘ F) (BX.Cod t1 ∘ F) (BX.Map t1 ∘ F)
        else AX.Null)
    using COMP.map-eq apply simp
    using COMP.AB.ide-implies-arr transformation-def by blast
qed

lemma simulation-cov-Exp:
assumes extensional-rts X and extensional-rts B and extensional-rts C
and simulation B C G
shows simulation (exponential-rts.resid X B) (exponential-rts.resid X C)
        (Exp⁺ X B C G)
proof −
  interpret COMP X B C
    using assms COMP-def by blast
  show ?thesis
    using assms(4) fixing-ide-gives-simulation-1 by blast
qed

lemma simulation-cnt-Exp:
assumes extensional-rts X and extensional-rts A and extensional-rts B
and simulation A B F
shows simulation (exponential-rts.resid B X) (exponential-rts.resid A X)
        (Exp⁻ A B X F)
```

```
proof −
  interpret COMP A B X
    using assms COMP-def by blast
  show ?thesis
    using assms(4) fixing-ide-gives-simulation-0 by blast
qed

lemma cov-Exp-ide:
assumes extensional-rts X and extensional-rts B
shows Exp→ X B B (I B) = I (exponential-rts.resid X B)
proof −
  interpret X: extensional-rts X
    using assms(1) by blast
  interpret B: extensional-rts B
    using assms(2) by blast
  interpret XB: exponential-rts X B ..
  interpret BB: exponential-rts B B ..
  interpret B: identity-simulation B
    using assms
    by (simp add: extensional-rts.axioms(1) identity-simulation.intro)
  interpret I-XB: identity-simulation XB.resid ..
  interpret COMP X B B
    using assms COMP-def by blast
  interpret XB: simulation XB.resid XB.resid ‹Exp→ X B B B.map›
    using simulation-cov-Exp X.extensional-rts-axioms B.extensional-rts-axioms
          B.simulation-axioms
    by blast
  show ?thesis
  proof
    fix t2
    show COMP.map X B B (BB.MkIde B.map, t2) = I-XB.map t2
    proof (cases XB.arr t2)
      show ¬ XB.arr t2 ⟹ ?thesis
        using XB.extensional XB.extensional by presburger
      assume t2: XB.arr t2
      interpret t2: transformation X B ‹XB.Dom t2› ‹XB.Cod t2› ‹XB.Map t2›
        using t2 by blast
      have B.map ∘ XB.Dom t2 = XB.Dom t2
        using comp-identity-simulation t2.F.simulation-axioms by blast
      moreover have B.map ∘ XB.Cod t2 = XB.Cod t2
        using comp-identity-simulation t2.G.simulation-axioms by blast
      moreover have B.map ∘ XB.Map t2 = XB.Map t2
      proof
        fix x
        show (B.map ∘ XB.Map t2) x = XB.Map t2 x
          apply simp
          by (metis t2.extensional t2.preserves-arr)
      qed
      ultimately show ?thesis
```

```
      using t2 map-eq comp-identity-simulation XB.MkArr-Map XB.arr-char
            XB.preserves-reflects-arr
      by force
    qed
  qed
qed

lemma cnt-Exp-ide:
assumes extensional-rts X and extensional-rts B
shows Exp⇐ B B X (I B) = I (exponential-rts.resid B X)
proof −
  interpret X: extensional-rts X
    using assms(1) by blast
  interpret B: extensional-rts B
    using assms(2) by blast
  interpret BX: exponential-rts B X ..
  interpret BB: exponential-rts B B ..
  interpret B: identity-simulation B
    using assms
    by (simp add: extensional-rts.axioms(1) identity-simulation.intro)
  interpret I-BX: identity-simulation BX.resid ..
  interpret COMP B B X
    using assms COMP-def by blast
  interpret BX: simulation BX.resid BX.resid ‹Exp⇐ B B X B.map›
    using X.extensional-rts-axioms B.extensional-rts-axioms simulation-cnt-Exp
          B.simulation-axioms
    by blast
  show ?thesis
  proof
    fix t1
    show COMP.map B B X (t1, BB.MkIde (I B)) = I-BX.map t1
    proof (cases BX.arr t1)
      show ¬ BX.arr t1 ⟹ ?thesis
        using BX.extensional [of t1] BX.extensional [of t1] by presburger
      assume t1: BX.arr t1
     interpret t1: transformation B X ‹BX.Dom t1› ‹BX.Cod t1› ‹BX.Map t1›
        using t1 by blast
      have BX.Dom t1 ∘ B.map = BX.Dom t1
        using comp-simulation-identity t1.F.simulation-axioms by blast
      moreover have BX.Cod t1 ∘ B.map = BX.Cod t1
        using comp-simulation-identity t1.G.simulation-axioms by blast
      moreover have BX.Map t1 ∘ B.map = BX.Map t1
      proof
        fix x
        show (BX.Map t1 ∘ B.map) x = BX.Map t1 x
          using t1.extensional by fastforce
      qed
      ultimately show ?thesis
        using t1 map-eq comp-identity-simulation BX.MkArr-Map BX.arr-char
```

```
            BX.preserves-reflects-arr
        by force
    qed
  qed
qed

lemma cov-Exp-comp:
assumes extensional-rts X and extensional-rts B and extensional-rts C
and extensional-rts D and simulation B C F and simulation C D G
shows Exp→ X B D (G ∘ F) = Exp→ X C D G ∘ Exp→ X B C F
proof −
  interpret X: extensional-rts X
    using assms(1) by blast
  interpret B: extensional-rts B
    using assms(2) by blast
  interpret C: extensional-rts C
    using assms(3) by blast
  interpret D: extensional-rts D
    using assms(4) by blast
  interpret XB: exponential-rts X B ..
  interpret BC: exponential-rts B C ..
  interpret CD: exponential-rts C D ..
  interpret BD: exponential-rts B D ..
  interpret XC: exponential-rts X C ..
  interpret XD: exponential-rts X D ..
  interpret F: simulation B C F
    using assms(5) by blast
  interpret G: simulation C D G
    using assms(6) by blast
  interpret GoF: composite-simulation B C D F G ..
  interpret XBC: COMP X B C
    using assms COMP-def by blast
  interpret XCD: COMP X C D
    using assms COMP-def by blast
  interpret XBD: COMP X B D
    using assms COMP-def by blast
  interpret EXP-F: simulation XB.resid XC.resid ‹Exp→ X B C F›
    using assms simulation-cov-Exp by blast
  interpret EXP-G: simulation XC.resid XD.resid ‹Exp→ X C D G›
    using assms simulation-cov-Exp by blast
  interpret EXP-GoF: simulation XB.resid XD.resid ‹Exp→ X B D (G ∘ F)›
    using assms GoF.simulation-axioms simulation-cov-Exp
    by blast
  interpret F: simulation-as-transformation B C F ..
  interpret G: simulation-as-transformation C D G ..
  show Exp→ X B D (G ∘ F) = Exp→ X C D G ∘ Exp→ X B C F
  proof
    fix t2
    show Exp→ X B D (G ∘ F) t2 = (Exp→ X C D G ∘ Exp→ X B C F) t2
```

```
    proof (cases XB.arr t2)
      show ¬ XB.arr t2 ⟹ ?thesis
        using EXP-GoF.extensional EXP-F.extensional EXP-G.extensional
        by simp
      assume t2: XB.arr t2
      interpret t2: transformation X B ‹XB.Dom t2› ‹XB.Cod t2› ‹XB.Map t2›
        using t2 by blast
      show ?thesis
      using t2 XBD.map-eq XCD.map-eq XBC.map-eq transformation-whisker-left
            F.transformation-axioms G.transformation-axioms
            t2.transformation-axioms C.weakly-extensional-rts-axioms
            D.weakly-extensional-rts-axioms F.simulation-axioms
            G.simulation-axioms
            transformation-whisker-left
              [of X B XB.Dom t2 XB.Cod t2 XB.Map t2 C F]
        by auto
    qed
  qed
qed

lemma cnt-Exp-comp:
assumes extensional-rts X and extensional-rts B and extensional-rts C
and extensional-rts D and simulation B C F and simulation C D G
shows Exp← B D X (G ∘ F) = Exp← B C X F ∘ Exp← C D X G
proof −
  interpret X: extensional-rts X
    using assms(1) by blast
  interpret B: extensional-rts B
    using assms(2) by blast
  interpret C: extensional-rts C
    using assms(3) by blast
  interpret D: extensional-rts D
    using assms(4) by blast
  interpret BC: exponential-rts B C ..
  interpret CD: exponential-rts C D ..
  interpret BD: exponential-rts B D ..
  interpret BX: exponential-rts B X ..
  interpret CX: exponential-rts C X ..
  interpret DX: exponential-rts D X ..
  interpret F: simulation B C F
    using assms(5) by blast
  interpret G: simulation C D G
    using assms(6) by blast
  interpret GoF: composite-simulation B C D F G ..
  interpret BCX: COMP B C X
    using assms COMP-def by blast
  interpret CDX: COMP C D X
    using assms COMP-def by blast
  interpret BDX: COMP B D X
```

```
    using assms COMP-def by blast
  interpret EXP-F: simulation CX.resid BX.resid ‹Exp← B C X F›
    using assms simulation-cnt-Exp by blast
  interpret EXP-G: simulation DX.resid CX.resid ‹Exp← C D X G›
    using assms simulation-cnt-Exp by blast
  interpret EXP-GoF: simulation DX.resid BX.resid ‹Exp← B D X (G ∘ F)›
    using assms GoF.simulation-axioms simulation-cnt-Exp
    by blast
  interpret F: simulation-as-transformation B C F ..
  interpret G: simulation-as-transformation C D G ..
  show Exp← B D X (G ∘ F) = Exp← B C X F ∘ Exp← C D X G
  proof
    fix t1
    show Exp← B D X (G ∘ F) t1 = (Exp← B C X F ∘ Exp← C D X G) t1
    proof (cases DX.arr t1)
      show ¬ DX.arr t1 ⟹ ?thesis
        using EXP-GoF.extensional EXP-F.extensional EXP-G.extensional
        by simp
      assume t1: DX.arr t1
     interpret t1: transformation D X ‹DX.Dom t1› ‹DX.Cod t1› ‹DX.Map t1›
        using t1 by blast
      show ?thesis
        using t1 BDX.map-eq CDX.map-eq BCX.map-eq
              F.transformation-axioms G.transformation-axioms
              t1.transformation-axioms B.weakly-extensional-rts-axioms
              C.weakly-extensional-rts-axioms F.simulation-axioms
              G.simulation-axioms transformation-whisker-right
              transformation-whisker-right
                [of D X DX.Dom t1 DX.Cod t1 DX.Map t1 C G]
        by auto
    qed
  qed
qed

lemma cov-Exp-preserves-inverse-simulations:
assumes extensional-rts X and extensional-rts B and extensional-rts C
and inverse-simulations B C G F
shows inverse-simulations
        (exponential-rts.resid X B) (exponential-rts.resid X C)
        (Exp→ X C B G) (Exp→ X B C F)
proof −
  interpret X: extensional-rts X
    using assms(1) by blast
  interpret B: extensional-rts B
    using assms(2) by blast
  interpret C: extensional-rts C
    using assms(3) by blast
  interpret XB: exponential-rts X B ..
  interpret XC: exponential-rts X C ..
```

interpret *B*: *identity-simulation B*
using *assms*
by (*simp add*: *extensional-rts.axioms*(*1*) *identity-simulation.intro*)
interpret *XB*: *identity-simulation XB.resid* **..**
interpret *C*: *identity-simulation C*
using *assms*
by (*simp add*: *extensional-rts.axioms*(*1*) *identity-simulation.intro*)
interpret *XC*: *identity-simulation XC.resid* **..**
interpret *F*: *simulation B C F*
using *assms*(*3*−*4*) *inverse-simulations-def* **by** *auto*
interpret *G*: *simulation C B G*
using *assms*
by (*simp add*: *inverse-simulations-def*)
interpret *XBC*: *COMP X B C*
using *assms COMP-def* **by** *blast*
interpret *XCB*: *COMP X C B*
using *assms COMP-def* **by** *blast*
interpret *XBB*: *COMP X B B*
using *assms COMP-def* **by** *blast*
interpret *XCC*: *COMP X C C*
using *assms COMP-def* **by** *blast*
interpret *HOM-F*: *simulation XB.resid XC.resid* ‹*Exp*→ *X B C F*›
using *assms simulation-cov-Exp F.simulation-axioms* **by** *blast*
interpret *HOM-G*: *simulation XC.resid XB.resid* ‹*Exp*→ *X C B G*›
using *assms simulation-cov-Exp G.simulation-axioms* **by** *blast*
interpret *FG*: *inverse-simulations B C G F*
using *assms* **by** *blast*
show *?thesis*
proof
show *Exp*→ *X B C F* ∘ *Exp*→ *X C B G* = *XC.map*
proof −
have *Exp*→ *X B C F* ∘ *Exp*→ *X C B G* = *Exp*→ *X C C* (*F* ∘ *G*)
using *assms cov-Exp-comp* [*of X C B C G F*] *F.simulation-axioms*
G.simulation-axioms
by *presburger*
also have ... = *Exp*→ *X C C C.map*
using *FG.inv* **by** *simp*
also have ... = *XC.map*
using *assms cov-Exp-ide* **by** *blast*
finally show *?thesis* **by** *blast*
qed
show *Exp*→ *X C B G* ∘ *Exp*→ *X B C F* = *XB.map*
proof −
have *Exp*→ *X C B G* ∘ *Exp*→ *X B C F* = *Exp*→ *X B B* (*G* ∘ *F*)
using *assms cov-Exp-comp* [*of X B C B F G*] *F.simulation-axioms*
G.simulation-axioms
by *presburger*
also have ... = *Exp*→ *X B B B.map*
using *FG.inv*′ **by** *simp*

```
    also have ... = XB.map
      using assms cov-Exp-ide by blast
    finally show ?thesis by blast
  qed
 qed
qed

lemma cov-Exp-preserves-invertible-simulations:
assumes extensional-rts X and extensional-rts B and extensional-rts C
and invertible-simulation B C F
shows invertible-simulation
        (exponential-rts.resid X B) (exponential-rts.resid X C)
        (Exp→ X B C F)
proof −
 obtain G where G: inverse-simulations B C G F
   using assms inverse-simulations-def invertible-simulation-def′ by blast
 have inverse-simulations
        (exponential-rts.resid X B) (exponential-rts.resid X C)
        (Exp→ X C B G) (Exp→ X B C F)
   using assms G cov-Exp-preserves-inverse-simulations by blast
 thus ?thesis
   using inverse-simulations-def invertible-simulation-def′ by blast
qed

lemma cnt-Exp-preserves-inverse-simulations:
assumes extensional-rts X and extensional-rts B and extensional-rts C
and inverse-simulations B C G F
shows inverse-simulations
        (exponential-rts.resid B X) (exponential-rts.resid C X)
        (Exp← B C X F) (Exp← C B X G)
proof −
 interpret X: extensional-rts X
   using assms(1) by blast
 interpret B: extensional-rts B
   using assms(2) by blast
 interpret C: extensional-rts C
   using assms(3) by blast
 interpret BX: exponential-rts B X ..
 interpret CX: exponential-rts C X ..
 interpret B: identity-simulation B
   using assms
   by (simp add: extensional-rts.axioms(1) identity-simulation.intro)
 interpret BX: identity-simulation BX.resid ..
 interpret C: identity-simulation C
   using assms
   by (simp add: extensional-rts.axioms(1) identity-simulation.intro)
 interpret CX: identity-simulation CX.resid ..
 interpret F: simulation B C F
   using assms(3−4) inverse-simulations-def by auto
```

```
  interpret G: simulation C B G
    using assms by (simp add: inverse-simulations-def)
  interpret HOM-F: simulation CX.resid BX.resid ‹Exp⇐ B C X F›
    using assms simulation-cnt-Exp [of X B C F] F.simulation-axioms by blast
  interpret HOM-G: simulation BX.resid CX.resid ‹Exp⇐ C B X G›
    using assms simulation-cnt-Exp [of X C B G] G.simulation-axioms by blast
  interpret FG: inverse-simulations B C G F
    using assms by blast
  show ?thesis
  proof
    show Exp⇐ C B X G ∘ Exp⇐ B C X F = CX.map
      using assms cnt-Exp-comp [of X C B C G F] F.simulation-axioms
            G.simulation-axioms FG.inv cnt-Exp-ide
      by force
    show Exp⇐ B C X F ∘ Exp⇐ C B X G = BX.map
      using assms cnt-Exp-comp [of X B C B F G] F.simulation-axioms
            G.simulation-axioms FG.inv' cnt-Exp-ide
      by force
  qed
qed

lemma cnt-Exp-preserves-invertible-simulations:
assumes extensional-rts X and extensional-rts B and extensional-rts C
and invertible-simulation B C F
shows invertible-simulation
        (exponential-rts.resid C X) (exponential-rts.resid B X)
        (Exp⇐ B C X F)
proof −
  obtain G where G: inverse-simulations B C G F
    using assms inverse-simulations-def invertible-simulation-def' by blast
  show ?thesis
    using assms G cnt-Exp-preserves-inverse-simulations
          inverse-simulations-sym inverse-simulations-def invertible-simulation-def'
    by fast
qed

end
```

Chapter 4

RTS's in Categories

4.1 RTS-Categories

In this section, we develop the notion of an *RTS-category*, which is analogous to a 2-category, except that the "vertical" structure is that of an RTS, rather than a category. So an RTS-category is a category with respect to a "horizontal" composition, which also has a vertical structure as an RTS.

theory *RTSCategory*
imports *Main RTSConstructions Category3.ConcreteCategory*
Category3.CartesianClosedCategory Category3.EquivalenceOfCategories
begin

4.1.1 Definition and Basic Properties

locale *rts-category* =
V: *extensional-rts resid* +
H: *category hcomp* +
VV: *fibered-product-rts resid resid resid H.dom H.cod* +
H: *simulation VV.resid resid*
‹λ*t*. *if VV.arr t then hcomp* (*fst t*) (*snd t*) *else V.null*›
for *resid* :: ′*a resid* (**infix** \ 70)
and *hcomp* :: ′*a comp* (**infixr** ⋆ 53) +
assumes *null-coincidence* [*simp*]: *H.null* = *V.null*
and *arr-coincidence* [*simp*]: *H.arr* = *V.arr*
and *src-dom* [*simp*]: *V.src* (*H.dom t*) = *H.dom t*
begin

notation *H.in-hom* («- : - → -»)

abbreviation *null*
where *null* ≡ *V.null*

abbreviation *arr*
where *arr* ≡ *V.arr*

abbreviation *src*
where $src \equiv V.src$

abbreviation *trg*
where $trg \equiv V.trg$

abbreviation *dom*
where $dom \equiv H.dom$

abbreviation *cod*
where $cod \equiv H.cod$

We refer to the identities for the horizontal composition as *objects*.

abbreviation *obj*
where $obj \equiv H.ide$

We refer to the identities for the vertical residuation as *states*.

abbreviation *sta*
where $sta \equiv V.ide$

interpretation VV: *fibered-product-of-extensional-rts resid resid resid dom cod*
..
interpretation H: *simulation-between-extensional-rts* $VV.resid$ *resid*
$\langle \lambda t.\ if\ VV.arr\ t\ then\ fst\ t \star snd\ t\ else\ null \rangle$
..

lemma *obj-is-isolated*:
assumes *obj* a **and** *obj* a' **and** $a \star a' \neq null$
shows $a = a'$
using *assms H.ide-def* **by** *fastforce*

lemma *obj-implies-sta*:
assumes *obj* a
shows *sta* a
using *assms H.ide-char arr-coincidence V.ide-src src-dom* **by** *metis*

lemma *trg-dom* [*simp*]:
shows *trg* (*dom* t) = *dom* t
by (*metis src-dom V.trg-src*)

lemma *src-cod* [*simp*]:
shows *src* (*cod* t) = *cod* t
by (*metis H.dom-cod src-dom*)

lemma *trg-cod* [*simp*]:
shows *trg* (*cod* t) = *cod* t
by (*metis H.dom-cod src-dom V.trg-src*)

lemma *dom-src* [*simp*]:
shows *dom* (*src t*) = *dom t*
using *H.preserves-src*
by (*metis VV.F.preserves-con VV.F.preserves-reflects-arr V.con-arr-src*(*1*)
V.con-imp-eq-src V.src-def src-dom)

lemma *dom-trg* [*simp*]:
shows *dom* (*trg t*) = *dom t*
using *H.preserves-trg*
by (*metis VV.F.extensional VV.F.preserves-trg V.arr-trg-iff-arr trg-dom*)

lemma *cod-src* [*simp*]:
shows *cod* (*src t*) = *cod t*
using *H.preserves-src*
by (*metis VV.G.preserves-con VV.G.preserves-reflects-arr V.con-arr-src*(*1*)
V.con-imp-eq-src src-cod V.src-def)

lemma *cod-trg* [*simp*]:
shows *cod* (*trg t*) = *cod t*
using *H.preserves-trg*
by (*metis VV.G.extensional VV.G.preserves-trg V.arr-trg-iff-arr trg-cod*)

lemma *arr-hcomp* [*intro*]:
assumes *H.seq t u*
shows *arr* (*t* ⋆ *u*)
using *assms VV.arr-char H.preserves-reflects-arr* **by** *auto*

lemma *sta-hcomp* [*intro*]:
assumes *H.seq t u* **and** *sta t* **and** *sta u*
shows *sta* (*t* ⋆ *u*)
using *assms VV.ide-char*$_{FP}$ *H.preserves-ide*
by (*elim H.seqE*) *auto*

lemma *src-hcomp* [*simp*]:
assumes *H.seq t u*
shows *src* (*t* ⋆ *u*) = *src t* ⋆ *src u*
using *assms VV.arr-char H.preserves-src* [*of* (*t*, *u*)] *VV.src-char VV.arr-src-iff-arr*
by (*elim H.seqE*) *auto*

lemma *trg-hcomp* [*simp*]:
assumes *H.seq t u*
shows *trg* (*t* ⋆ *u*) = *trg t* ⋆ *trg u*
using *assms VV.arr-char H.preserves-trg* [*of* (*t*, *u*)] *VV.trg-char VV.arr-trg-iff-arr*
by (*elim H.seqE*) *auto*

lemma *con-implies-hpar*:
assumes *t* ⌢ *u*
shows *H.par t u*

using *assms V.con-implies-arr arr-coincidence cod-src V.con-imp-eq-src dom-src*
by *metis*

lemma *hpar-arr-resid*:
assumes $t \frown u$
shows *H.par t* $(t \setminus u)$
using *assms con-implies-hpar V.con-implies-arr arr-coincidence V.arr-resid cod-src dom-src V.resid-arr-self*
by *auto*

lemma *dom-resid* [*simp*]:
assumes $t \frown u$
shows *dom* $(t \setminus u) =$ *dom t*
using *assms hpar-arr-resid* **by** *simp*

lemma *cod-resid* [*simp*]:
assumes $t \frown u$
shows *cod* $(t \setminus u) =$ *cod t*
using *assms hpar-arr-resid* **by** *simp*

RTS-categories enjoy an "interchange law" between residuation and composition.

lemma *resid-hcomp*:
assumes $r \frown t$ **and** $s \frown u$ **and** *H.seq r s*
shows $r \star s \frown t \star u$
and $(r \star s) \setminus (t \star u) = r \setminus t \star s \setminus u$
proof −
have *tu*: *H.seq t u*
using *assms con-implies-hpar*
by (*elim H.seqE*, *intro H.seqI*) *auto*
have *1*: *VV.con* (*r*, *s*) (*t*, *u*)
using *assms tu VV.con-char*
by (*elim H.seqE*) *auto*
have *2*: *H.dom r* = *H.cod s* ∧ *H.dom t* = *H.cod u*
using *assms tu* **by** *blast*
show $r \star s \frown t \star u$
using *assms 1 2 VV.con-char VV.arr-char V.con-implies-arr H.preserves-con* [*of* (*r*, *s*) (*t*, *u*)]
by *simp*
show $(r \star s) \setminus (t \star u) = r \setminus t \star s \setminus u$
using *assms 1 2 VV.resid-def VV.arr-char V.con-implies-arr H.preserves-resid* [*of* (*r*, *s*) (*t*, *u*)]
by *auto*
qed

lemma *dom-vcomp* [*simp*]:
assumes *V.composable t u*
shows *dom* $(t \cdot u) =$ *dom t*

using *assms arr-coincidence*
by (*metis dom-src V.src-comp*)

lemma *cod-vcomp* [*simp*]:
assumes *V.composable t u*
shows *cod* (*t* · *u*) = *cod t*
using *assms arr-coincidence*
by (*metis cod-src V.src-comp*)

If the vertical structure is that of an RTS with composites, then the usual middle-four interchange law holds, as for 2-categories, between the horizontal and vertical compositions.

lemma *interchange*:
assumes *V.composable t r* **and** *V.composable u s* **and** *H.seq t u*
shows *V.composable* (*t* ⋆ *u*) (*r* ⋆ *s*)
and (*t* ⋆ *u*) · (*r* ⋆ *s*) = (*t* · *r*) ⋆ (*u* · *s*)
proof −
have *r*: *arr r* **and** *s*: *arr s* **and** *rs*: *dom r* = *cod s*
using *assms H.seqE*
apply *auto*[*3*]
by (*metis cod-src cod-trg V.composable-imp-seq dom-src dom-trg V.seqE*$_{W\,E}$)
have *1*: *V.composite-of* (*t* ⋆ *u*) (*r* ⋆ *s*) ((*t* · *r*) ⋆ (*u* · *s*))
proof
have *2*: *t* · *r* ⌢ *t*
using *assms*(*1*) *V.con-comp-iff* [*of t t r*] *V.con-sym* **by** *force*
have *3*: *u* · *s* ⌢ *u*
using *assms*(*2*) *V.con-comp-iff* [*of u u s*] *V.con-sym* **by** *force*
show *t* ⋆ *u* ≲ *t* · *r* ⋆ *u* · *s*
by (*metis 2 3 V.arr-comp V.arr-resid V.con-sym V.prfx-comp*
arr-coincidence assms(*1*−*3*) *resid-hcomp*(*1*) *resid-hcomp*(*2*)
sta-hcomp)
show (*t* · *r* ⋆ *u* · *s*) \ (*t* ⋆ *u*) ∼ *r* ⋆ *s*
by (*metis 2 3 H.seqI V.comp-resid-prfx V.prfx-reflexive*
arr-coincidence hpar-arr-resid resid-hcomp(*2*) *rs*)
qed
show *V.composable* (*t* ⋆ *u*) (*r* ⋆ *s*)
using *assms 1 V.composable-def* **by** *auto*
show (*t* ⋆ *u*) · (*r* ⋆ *s*) = *t* · *r* ⋆ *u* · *s*
using *1 V.comp-is-composite-of* **by** *blast*
qed

lemma *hcomp-monotone*:
assumes *r* ≲ *t* **and** *s* ≲ *u* **and** *H.seq r s*
shows *r* ⋆ *s* ≲ *t* ⋆ *u*
by (*metis V.prfx-implies-con assms*(*1*−*3*) *hpar-arr-resid*
resid-hcomp(*1*−*2*) *sta-hcomp*)

lemma *dom-join* [*simp*]:
assumes *V.joinable t u*

shows $H.dom\ (t \sqcup u) = H.dom\ t$
using *assms con-implies-hpar V.joinable-implies-con*
by (*metis dom-trg hpar-arr-resid V.trg-join*)

lemma *cod-join* [*simp*]:
assumes $V.joinable\ t\ u$
shows $H.cod\ (t \sqcup u) = H.cod\ t$
using *assms con-implies-hpar V.joinable-implies-con*
by (*metis cod-trg hpar-arr-resid V.trg-join*)

lemma *join-hcomp*:
assumes $V.joinable\ r\ t$ **and** $V.joinable\ s\ u$ **and** $H.seq\ r\ s$
shows $(r \star s) \sqcup (t \star u) = (r \sqcup t) \star (s \sqcup u)$
proof (*intro V.join-eqI*)
show $r \star s \lesssim (r \sqcup t) \star (s \sqcup u)$
using *assms hcomp-monotone V.arr-prfx-join-self* **by** *presburger*
show *0*: $t \star u \lesssim (r \sqcup t) \star (s \sqcup u)$
using *assms hcomp-monotone V.arr-prfx-join-self V.join-sym H.seqE H.seqI*
con-implies-hpar V.joinable-iff-join-not-null V.joinable-implies-con
by *metis*
have *1*: $H.seq\ (r \sqcup t)\ (s \sqcup u)$
using *assms H.seqE H.seqI arr-coincidence cod-join dom-join V.joinable-iff-arr-join*
by *metis*
have *2*: $r \sqcup t \frown t \wedge r \sqcup t \frown r$
using *assms V.arr-prfx-join-self V.con-sym V.join-sym V.joinable-iff-arr-join*
V.prfx-implies-con
by *metis*
have *3*: $s \sqcup u \frown u \wedge s \sqcup u \frown s$
using *assms V.arr-prfx-join-self V.con-sym V.join-sym V.joinable-iff-arr-join*
V.prfx-implies-con
by *metis*
have *4*: $(r \sqcup t) \setminus t = r \setminus t \wedge (r \sqcup t) \setminus r = t \setminus r$
by (*metis* (*no-types*, *lifting*) *2 V.arr-resid-iff-con V.con-sym*
V.join-src V.join-sym V.joinable-implies-con $V.resid\text{-}join_E(3)$
V.src-resid V.trg-def assms(*1*))
have *5*: $(s \sqcup u) \setminus u = s \setminus u \wedge (s \sqcup u) \setminus s = u \setminus s$
by (*metis* (*no-types*, *lifting*) *3 V.arr-resid-iff-con V.con-sym*
V.join-src V.join-sym V.joinable-implies-con $V.resid\text{-}join_E(3)$
V.src-resid V.trg-def assms(*2*))
show $((r \sqcup t) \star (s \sqcup u)) \setminus (t \star u) = (r \star s) \setminus (t \star u)$
using *assms 1 2 3 4 5 V.joinable-implies-con resid-hcomp* **by** *auto*
show $((r \sqcup t) \star (s \sqcup u)) \setminus (r \star s) = (t \star u) \setminus (r \star s)$
using *assms 1 2 3 4 5 V.joinable-implies-con resid-hcomp*
apply *auto*[*1*]
by (*metis 0 V.arr-resid-iff-con V.prfx-implies-con hpar-arr-resid resid-hcomp*(*2*))
qed

The source and target maps given by the vertical structure are functorial with respect to the horizontal structure.

```
sublocale src: functor hcomp hcomp src
  apply unfold-locales
  subgoal using V.src-def by auto
  by auto

sublocale trg: functor hcomp hcomp trg
  apply unfold-locales
  subgoal using V.trg-def by auto
  by auto
```

An isomorphism with respect to the horizontal composition is an identity with respect to the vertical residuation.

```
lemma iso-implies-sta:
assumes H.iso f
shows sta f
proof −
  obtain g where inv-fg: H.inverse-arrows f g
    using assms by blast
  have f: arr f and g: arr g and fg: H.dom f = H.cod g
    using inv-fg arr-coincidence
    by auto fastforce+

  have sta (cod f ⋆ f)
  proof −
    have sta ((trg f ⋆ trg g) ⋆ f)
    proof −
      have 1: sta (trg g ⋆ f)
      proof −
        have 2: sta ((g ⋆ src f) · (trg g ⋆ f))
        proof −
          have sta (g ⋆ f)
            using inv-fg obj-implies-sta by blast
          moreover have V.composable g (trg g)
            using V.composable-iff-comp-not-null g by auto
          moreover have V.composable (src f) f
            using V.composable-iff-comp-not-null f by auto
          moreover have H.seq g (src f)
            by (metis H.dom-null H.ext H.ide-compE
                H.inverse-arrowsE H.seqI cod-src dom-trg
                inv-fg src.preserves-arr trg.is-extensional)
          ultimately show ?thesis
            using g interchange by auto
        qed
        moreover have V.composable (g ⋆ src f) (trg g ⋆ f)
          using 2 V.composable-iff-arr-comp by blast
        ultimately show sta (trg g ⋆ f)
          using g V.comp-is-composite-of
                V.divisors-of-ide
                  [of g ⋆ src f trg g ⋆ f (g ⋆ src f) · (trg g ⋆ f)]
```

```
      by blast
    qed
    moreover have H.seq (trg f) (trg g ⋆ f)
      using f g inv-fg 1
      by (intro H.seqI) auto
    moreover have sta (trg f)
      using assms arr-coincidence by fastforce
    ultimately show ?thesis
      using g H.comp-assoc by auto
  qed
  moreover have H.inverse-arrows (trg f) (trg g)
    using inv-fg trg.preserves-inverse-arrows by auto
  ultimately show ?thesis
    by fastforce
 qed
 thus sta f
   using assms H.comp-cod-arr by force
qed
```

4.1.2 Hom-RTS's

We have defined the vertical structure of an RTS-category as a "global" residuation, but in fact a pair of arrows can only be consistent if they have the same domain and codomain with respect to the horizontal composition. If we restrict the global residuation to sets of arrows having the same domain and codomain, then we obtain hom-RTS's, analogous to the hom-categories in the case of a 2-category.

```
abbreviation HOM
where HOM a b ≡ sub-rts.resid resid (λt. «t : a → b»)

lemma sub-rts-HOM:
shows sub-rts resid (λt. «t : a → b»)
proof
  show ⋀t. «t : a → b» ⟹ arr t
    by auto
  show ⋀t u. ⟦«t : a → b»; «u : a → b»; t ⌣ u⟧ ⟹ «t \ u : a → b»
    using H.in-homE H.in-homI hpar-arr-resid by metis
  show ⋀t u. ⟦«t : a → b»; «u : a → b»; t ⌣ u⟧ ⟹
                ∃X. «X : a → b» ∧ X ∈ V.sources t ∧ X ∈ V.sources u
    using arr-coincidence
    by (metis (mono-tags, lifting) H.arr-iff-in-hom H.in-homE cod-src
        V.con-imp-coinitial-ax V.con-implies-arr(1) dom-src V.src-eqI
        V.src-in-sources)
qed

lemma extensional-rts-HOM:
assumes obj a and obj b
shows HOM a b ∈ Collect extensional-rts
```

```
proof −
  interpret HOM: sub-rts resid ‹λt. t ∈ H.hom a b›
    using assms sub-rts-HOM by fastforce
  show ?thesis
    using HOM.preserves-extensional-rts V.extensional-rts-axioms by auto
qed
```

Given an object a and an arrow t, horizontal composition with t determines a transformation $HOM^{\rightarrow}\ a\ t$ from $HOM\ a\ (H.dom\ t)$ to $HOM\ a\ (H.cod\ t)$.

```
abbreviation cov-HOM  (HOM→)
where HOM→ a t ≡
      (λx. if residuation.arr (HOM a (dom t)) x then t ⋆ x else null)

lemma simulation-cov-HOM-sta:
assumes obj a and sta f
shows simulation (HOM a (dom f)) (HOM a (cod f)) (HOM→ a f)
proof −
  interpret HOM-a: sub-rts resid ‹λt. «t : a → dom f»›
    using sub-rts-HOM by blast
  interpret HOM-b: sub-rts resid ‹λt. «t : a → cod f»›
    using sub-rts-HOM by blast
  show simulation HOM-a.resid HOM-b.resid (HOM→ a f)
  proof
    show ⋀x. ¬ HOM-a.arr x ⟹ HOM→ a f x = HOM-b.null
      using assms HOM-b.null-char by auto
    fix t u
    assume tu: HOM-a.con t u
    have 1: «f ⋆ t : a → cod f» ∧ «f ⋆ u : a → cod f»
    proof −
      have f ⌢ f
        using assms(2) by auto
      thus ?thesis
        using assms tu HOM-a.con-implies-arr HOM-a.con-char [of t u]
        by auto
    qed
    have 2: «f ⋆ t : a → cod f» ∧ «f ⋆ u : a → cod f» ∧ f ⋆ t ⌢ f ⋆ u
      using assms tu 1 HOM-a.con-char resid-hcomp(1) [of f f t u]
      by blast
    show HOM-b.con (HOM→ a f t) (HOM→ a f u)
      using assms(2) tu 2 HOM-a.con-implies-arr [of t u]
            HOM-a.con-char HOM-b.con-char
      by auto
    show HOM→ a f (HOM-a.resid t u) =
         HOM-b.resid (HOM→ a f t) (HOM→ a f u)
    proof −
      have HOM-a.arr (t \ u)
        using tu HOM-a.con-char [of t u] HOM-a.arr-char [of t \ u]
              dom-resid [of t u] cod-resid [of t u]
```

```
        by fastforce
      moreover have arr (f ⋆ t)
        using 2 by auto
      moreover have f ⌢ f
        using assms(2) by auto
      ultimately show ?thesis
        using assms(1-2) tu 1 2 HOM-b.resid-def HOM-a.resid-def
              HOM-a.con-implies-arr HOM-a.con-char resid-hcomp(2)
        by auto
    qed
  qed
qed

lemma transformation-cov-HOM-arr:
assumes obj a and arr t
shows transformation (HOM a (dom t)) (HOM a (cod t))
        (HOM→ a (src t)) (HOM→ a (trg t)) (HOM→ a t)
proof -
  interpret Dom': sub-rts resid ‹λx. x ∈ H.hom a (dom t)›
    using assms sub-rts-HOM by auto
  interpret Dom': sub-rts-of-extensional-rts resid ‹λx. x ∈ H.hom a (dom t)›
..
  interpret Cod': sub-rts resid ‹λx. x ∈ H.hom a (cod t)›
    using assms  sub-rts-HOM by auto
  interpret Cod': sub-rts-of-extensional-rts resid ‹λx. x ∈ H.hom a (cod t)› ..
  have Dom'-eq: Dom'.resid = HOM a (dom t)
    using assms Cod'.null-char by auto
  have Cod'-eq: Cod'.resid = HOM a (cod t)
    using assms Cod'.null-char by auto
  interpret Dom: extensional-rts ‹HOM a (dom t)›
    using Dom'-eq Dom'.extensional-rts-axioms by simp
  interpret Cod: extensional-rts ‹HOM a (cod t)›
    using Cod'-eq Cod'.extensional-rts-axioms by simp
  interpret Src: simulation
                 ‹HOM a (dom t)› ‹HOM a (cod t)› ‹HOM→ a (src t)›
    using assms simulation-cov-HOM-sta [of a src t] by auto
  interpret Trg: simulation
                 ‹HOM a (H.dom t)› ‹HOM a (cod t)› ‹HOM→ a (trg t)›
    using assms simulation-cov-HOM-sta [of a trg t] by auto
  show ?thesis
  proof
    show ⋀f. ¬ Dom.arr f ⟹ HOM→ a t f = Cod.null
      using Dom'-eq Cod'-eq Cod'.null-char by simp
    fix f
    assume f: Dom.ide f
    have 1: H.seq t f
      using assms f Dom'.ide-char Dom'.arr-char
      by (intro H.seqI) auto
    have 2: «t ⋆ f : a → cod t»
```

```
    using f 1 Dom'.ide-char Dom'.arr-char H.cod-comp
    by (intro H.in-homI) auto
show Cod.src (HOM→ a t f) = HOM→ a (src t) f
  using f 1 2 Cod'.src-char Dom'.ide-char Cod'.arr-char by simp
show Cod.trg (HOM→ a t f) = HOM→ a (trg t) f
  using f 1 2 Cod'.trg-char Dom'.ide-char Cod'.arr-char by simp
next
fix f
assume f: Dom.arr f
have arr-f: arr f
  using f Dom'.arr-char by auto
have 3: H.cod f = H.dom t
  using f Dom'.arr-char by auto
have 4: Dom.src f = src f
  using f Dom'-eq Dom'.src-char by auto
have 5: VV.con (t, Dom.src f) (src t, f)
  unfolding VV.con-char
  using assms f 3 4 Dom'-eq Dom'.arr-char
  by (intro conjI) auto
have 6: VV.con (src t, f) (t, Dom.src f)
  using assms arr-f 3 4 VV.con-char by auto
have 7: VV.resid (t, Dom.src f) (src t, f) = (t, trg f)
  using f 5 VV.resid-def VV.con-char Dom'.src-char V.con-implies-arr
  by auto
show HOM a (cod t) (HOM→ a t (Dom.src f)) (HOM→ a (src t) f) =
    HOM→ a t (Dom.trg f)
proof −
  have HOM a (cod t) (HOM→ a t (Dom.src f)) (HOM→ a (src t) f) =
      Cod'.resid (t ⋆ src f) (src t ⋆ f)
    using f 4 Dom.arr-src-iff-arr [of f] by simp
  also have ... = (t ⋆ src f) \ (src t ⋆ f)
   using assms f 4 5 Cod'.resid-def Dom'-eq Dom'.arr-char H.preserves-con
       VV.con-implies-arr
    by auto
  also have ... = t \ src t ⋆ src f \ f
    using assms f 4 5 7 VV.arr-char Dom'.arr-char
      H.preserves-resid [of (t, src f) (src t, f)]
    by auto
  also have ... = HOM→ a t (Dom.trg f)
    using assms f Dom'.arr-char Dom'.trg-char H.in-homI by auto
  finally show ?thesis by blast
qed
show HOM a (cod t) (HOM→ a (src t) f) (HOM→ a t (Dom.src f)) =
    HOM→ a (trg t) f
proof −
  have HOM a (cod t) (HOM→ a (src t) f) (HOM→ a t (Dom.src f)) =
      Cod'.resid (src t ⋆ f) (t ⋆ Dom.src f)
    using f by simp
  also have ... = (src t ⋆ f) \ (t ⋆ src f)
```

```
      using assms f 4 Cod'.resid-def Dom'.arr-char by auto
    also have ... = src t \ t ⋆ f \ src f
      using assms f arr-f 4 6 VV.arr-char Dom'.arr-char VV.resid-def
            H.preserves-resid [of (src t, f) (t, src f)]
      by auto
    also have ... = cov-HOM a (trg t) f
      using assms f arr-f by simp
    finally show ?thesis by blast
  qed
  show Cod.join-of (HOM→ a t (Dom.src f)) (HOM→ a (src t) f)
       (HOM→ a t f)
  proof −
    have 8: t ⋆ src f ⊔ src t ⋆ f = t ⋆ f
      using assms f arr-f V.join-src V.join-sym V.joinable-iff-arr-join
            join-hcomp Dom'.arr-char H.seqI
      by auto
    moreover have «f : a → dom t»
      using f Dom'.arr-char by auto
    moreover have «src f : a → dom t»
      using f Dom'.arr-char H.in-homI by auto
    moreover have V.joinable (t ⋆ src f) (src t ⋆ f)
    proof −
      have t ⋆ f ∈ H.hom a (cod t)
        using assms f Dom'.arr-char by auto
      thus ?thesis
        using 8 V.joinable-iff-arr-join by auto
    qed
    ultimately show ?thesis
      using assms 4 Dom'.arr-char Cod'.join-of-char
            V.join-is-join-of [of t ⋆ src f src t ⋆ f]
      by auto
  qed
 qed
qed
```

For fixed a, the mapping $HOM^{\rightarrow}\ a$ takes horizontal composite of arrows to function composition.

```
lemma cov-HOM-hcomp:
assumes obj a and H.seq t u
shows HOM→ a (t ⋆ u) = HOM→ a t ∘ HOM→ a u
proof
  interpret au: transformation ‹HOM a (dom u)› ‹HOM a (cod u)›
                ‹HOM→ a (src u)› ‹HOM→ a (trg u)› ‹HOM→ a u›
    using assms transformation-cov-HOM-arr [of a u] by fastforce
  interpret at: transformation ‹HOM a (dom t)› ‹HOM a (cod t)›
                ‹HOM→ a (src t)› ‹HOM→ a (trg t)› ‹HOM→ a t›
    using assms transformation-cov-HOM-arr [of a t] by fastforce
  fix x
  have (HOM→ a t ∘ HOM→ a u) x =
```

```
      (if au.A.arr x then (t ⋆ u) ⋆ x else null)
    using assms(2) H.comp-assoc au.preserves-arr H.seqE H.null-is-zero
    apply (cases au.A.arr x)
     apply auto[2]
    by metis
  thus HOM→ a (t ⋆ u) x = (HOM→ a t ∘ HOM→ a u) x
    using assms(2) by auto
qed
```

The mapping $HOM^{\rightarrow}\ a$ preserves consistency and residuation.

```
lemma cov-HOM-resid:
assumes obj a and V.con t u
shows cov-HOM a (t \ u) =
      consistent-transformations.resid
        (HOM a (dom t)) (HOM a (cod t)) (HOM→ a (trg u))
        (HOM→ a t) (HOM→ a u)
proof −
  have 1: HOM a (dom u) = HOM a (dom t)
    using assms V.con-imp-eq-src dom-src by metis
  have 2: HOM a (cod u) = HOM a (cod t)
    using assms V.con-imp-eq-src cod-src by metis
  interpret A: sub-rts resid ‹λx. «x : a → dom t»›
    using sub-rts-HOM by blast
  interpret A: extensional-rts A.resid
    using assms V.con-implies-arr extensional-rts-HOM by simp
  interpret B: sub-rts resid ‹λx. «x : a → cod t»›
    using sub-rts-HOM by blast
  interpret B: extensional-rts B.resid
    using assms V.con-implies-arr extensional-rts-HOM by simp
  interpret at: transformation A.resid B.resid
                ‹HOM→ a (src t)› ‹HOM→ a (trg t)› ‹HOM→ a t›
    using assms transformation-cov-HOM-arr [of a t] V.con-implies-arr
    by fastforce
  interpret au: transformation A.resid B.resid
                ‹HOM→ a (src t)› ‹HOM→ a (trg u)› ‹HOM→ a u›
    using assms 1 2 V.con-imp-eq-src transformation-cov-HOM-arr [of a u]
          V.con-implies-arr
    by presburger
  interpret at-au: consistent-transformations A.resid B.resid
                   ‹HOM→ a (src t)› ‹HOM→ a (trg t)› ‹HOM→ a (trg u)›
                   ‹HOM→ a t› ‹HOM→ a u›
  proof
    show ⋀x. A.ide x ⟶ HOM→ a t x ⌣B HOM→ a u x
    proof (intro impI)
      fix x
      assume x: A.ide x
      show HOM→ a t x ⌣B HOM→ a u x
        using assms x resid-hcomp B.con-char B.arr-char at.preserves-arr
              au.preserves-arr
```

```
      apply auto[1]
       apply (metis (no-types, lifting) B.inclusion V.arrE arr-coincidence
          A.ide-implies-arr H.seqE)
      by (metis (full-types) B.not-arr-null B.null-char A.ide-implies-arr
          rts-category.sub-rts-HOM)
  qed
qed
show HOM^→ a (t \ u) = at-au.resid
proof (intro transformation-eqI)
  show extensional-rts (HOM a (cod t)) ..
  show transformation (HOM a (dom t)) B.resid
         (cov-HOM a (trg u)) at-au.apex at-au.resid
    using at-au.transformation-resid by blast
  show transformation (HOM a (dom t)) B.resid
         (HOM^→ a (trg u)) at-au.apex (HOM^→ a (t \ u))
  proof −
    have dom (t \ u) = dom t
      using assms dom-resid by blast
    moreover have cod (t \ u) = cod t
      using assms cod-resid by blast
    moreover have (λx. if residuation.arr (HOM a (dom u)) x
                       then src (t \ u) ⋆ x else null) =
                  (λx. if residuation.arr (HOM a (dom u)) x
                       then trg u ⋆ x else null)
      using assms by auto
   moreover have (λx. if A.arr x then trg (t \ u) ⋆ x else null) = at-au.apex
    proof
      fix x
      show (if A.arr x then trg (t \ u) ⋆ x else null) = at-au.apex x
      proof (cases A.arr x)
        case False
        show ?thesis
          using False B.null-char by auto
        next
        case True
        show ?thesis
        proof −
          have 3: residuation.arr (HOM a (dom u)) (A.src x)
            using True 1 by auto
          have «t ⋆ A.src x : a → cod t»
            using True B.arr-char at.preserves-arr by force
          moreover have «u ⋆ A.src x : a → cod t»
           using True 1 A.arr-char A.arr-src-if-arr B.arr-char au.preserves-arr
            by force
          moreover have «trg u ⋆ x : a → cod t»
            using assms True A.arr-char con-implies-hpar by fastforce
          moreover have 4: «(t ⋆ A.src x) \ (u ⋆ A.src x) : a → cod t»
            using 1 3 A.ide-iff-src-self A.src-src B.con-char B.resid-closed
                  at-au.con
```

```
        by presburger
      moreover have t ⋆ A.src x ⌢ u ⋆ A.src x
        using 4 B.inclusion V.arr-resid-iff-con by blast
      moreover have trg (t \ u) ⋆ x =
                    (trg u ⋆ x) \ ((t ⋆ A.src x) \ (u ⋆ A.src x))
      proof −
        have (trg u ⋆ x) \ ((t ⋆ A.src x) \ (u ⋆ A.src x)) =
              (trg u ⋆ x) \ ((t \ u) ⋆ A.src x)
          by (metis (no-types, lifting) 1 3 A.con-arr-self A.con-char
              A.trg-char A.trg-src H.arrI V.trg-def assms(2)
              calculation(1) resid-hcomp(2))
        also have ... = trg (t \ u) ⋆ x
        proof −
          have «trg u ⋆ x : a → cod t»
            using assms True A.arr-char con-implies-hpar by fastforce
          thus ?thesis
            using assms True resid-hcomp [of trg u t \ u x A.src x]
            by (metis (no-types, lifting) A.con-arr-src(1) A.con-char
                A.resid-arr-src A.resid-def H.arrI V.arr-resid V.con-def
                V.con-imp-arr-resid V.resid-src-arr V.src-resid V.trg-def)
        qed
        finally show ?thesis by simp
      qed
      ultimately show ?thesis
        using True 1 B.resid-def by auto
    qed
  qed
  ultimately show ?thesis
    using assms transformation-cov-HOM-arr [of a t \ u] by auto
qed
show ⋀x. A.ide x ⟹ HOM→ a (t \ u) x = at-au.resid x
proof −
  fix x
  assume x: A.ide x
  have at-au.resid x = B.resid (HOM→ a t x) (HOM→ a u x)
    using x at-au.resid-ide by blast
  also have ... = HOM→ a (t \ u) x
    using assms x 1 A.ide-char A.arr-char resid-hcomp [of t u x x]
          B.resid-def
    apply clarsimp
    apply (intro conjI impI)
    subgoal by (metis B.inclusion V.ideE)
    subgoal using V.trg-def con-implies-hpar trg-dom by force
    subgoal using B.arr-char au.preserves-arr at.preserves-arr
      by auto (metis arr-coincidence category.in-homE
          rts-category-axioms rts-category-def)
    done
  finally show HOM→ a (t \ u) x = at-au.resid x by simp
```

```
    qed
  qed
qed
```

We can dualize the above, to define, given an object c and an arrow t, a contravariant mapping $HOM^{\leftarrow}\ c\ t$ from $HOM\ (H.cod\ t)\ c$ to $HOM\ (H.dom\ t)\ c$. I have not carried out a full development parallel to the covariant case, because the contravariant version is not used in an essential way in this article.

```
abbreviation cnt-HOM  (HOM^←)
where HOM^← c t ≡
      (λx. if residuation.arr (HOM (cod t) c) x then x ⋆ t else null)

lemma simulation-cnt-HOM-sta:
assumes sta f and «f : a → b» and obj c
shows simulation (HOM b c) (HOM a c) (HOM^← c f)
proof −
  interpret HOM-a: sub-rts resid ‹λt. «t : a → c»›
    using sub-rts-HOM by blast
  interpret HOM-b: sub-rts resid ‹λt. «t : b → c»›
    using sub-rts-HOM by blast
  show simulation HOM-b.resid HOM-a.resid (HOM^← c f)
  proof
    show ⋀x. ¬ HOM-b.arr x ⟹ HOM^← c f x = HOM-a.null
      using assms(2) HOM-a.null-char by auto
    fix t u
    assume tu: HOM-b.con t u
    show HOM-a.con (HOM^← c f t) (HOM^← c f u)
    proof −
      have f ⌢ f
        using assms(1) by auto
      hence «t ⋆ f : a → c» ∧ «u ⋆ f : a → c» ∧ t ⋆ f ⌢ u ⋆ f
        using assms tu HOM-b.con-implies-arr HOM-b.con-char [of t u]
              resid-hcomp(1) [of t u f f]
        by (intro conjI) blast+
      thus ?thesis
        using assms(2) tu HOM-b.con-implies-arr [of t u]
              HOM-a.con-char HOM-b.con-char
        by auto
    qed
    show HOM^← c f (HOM-b.resid t u) =
         HOM-a.resid (HOM^← c f t) (HOM^← c f u)
    proof −
      have HOM-b.arr (t \ u)
        using tu HOM-b.con-char [of t u] HOM-b.arr-char [of t \ u]
              dom-resid [of t u] cod-resid [of t u]
        by fastforce
      moreover have arr (t ⋆ f)
        using assms(2) tu HOM-b.con-char HOM-b.con-implies-arr
```

```
      by blast
    moreover
    have «t ⋆ f : a → c» ∧ «u ⋆ f : a → c» ∧ t ⋆ f ⌢ u ⋆ f
      using assms tu HOM-b.con-char [of t u] resid-hcomp(1)
      by (intro conjI) blast+
    moreover have f ⌢ f
      using assms(2) by auto
    ultimately show ?thesis
      using assms(1−2) tu HOM-a.resid-def HOM-b.resid-def
            HOM-b.con-implies-arr HOM-b.con-char resid-hcomp(2)
      by auto
  qed
 qed
qed

lemma HOM-preserves-isomorphic-left:
assumes H.isomorphic a b and obj c
shows isomorphic-rts (HOM a c) (HOM b c)
proof −
 interpret HOM-a: sub-rts resid ‹λt. «t : a → c»›
  using sub-rts-HOM by blast
 interpret HOM-b: sub-rts resid ‹λt. «t : b → c»›
  using sub-rts-HOM by blast
 obtain f g where fg: H.inverse-arrows f g ∧ dom f = a ∧ cod f = b
  using assms(1) by blast
 have 1: sta f ∧ sta g
  using fg iso-implies-sta by blast
 have f: «f : a → b»
  using fg H.inverse-arrows-def
  by (intro H.in-homI) auto
 have g: «g : b → a»
  using fg H.inverse-arrows-def
  by (intro H.in-homI) auto
 let ?F = λt. if HOM-b.arr t then t ⋆ f else null
 let ?G = λt. if HOM-a.arr t then t ⋆ g else null
 interpret F: simulation HOM-b.resid HOM-a.resid ?F
  using assms(2) f 1 simulation-cnt-HOM-sta by blast
 interpret G: simulation HOM-a.resid HOM-b.resid ?G
  using assms(2) g 1 simulation-cnt-HOM-sta by blast
 interpret FG: inverse-simulations HOM-a.resid HOM-b.resid ?F ?G
 proof
  show ?F ∘ ?G = I HOM-a.resid
  proof
   fix x
   show (?F ∘ ?G) x = I HOM-a.resid x
    using G.preserves-reflects-arr H.comp-arr-dom H.comp-assoc
          H.comp-inv-arr HOM-a.arr-char HOM-a.null-char fg
    by auto
  qed
```

```
      show ?G ∘ ?F = I HOM-b.resid
      proof
        fix x
        show (?G ∘ ?F) x = I HOM-b.resid x
        proof (cases HOM-b.arr x)
          show ¬ HOM-b.arr x ⟹ ?thesis
            using HOM-b.null-char H.null-is-zero by auto
          assume x: HOM-b.arr x
          show ?thesis
          proof −
            have obj (f ⋆ g)
              using fg by blast
            moreover have arr (x ⋆ f ⋆ g)
              using f g x HOM-b.arr-char by blast
            moreover have «x ⋆ f : dom f → c»
              using f x HOM-b.arr-char H.dom-comp by blast
            ultimately show ?thesis
              using fg x H.comp-assoc H.comp-arr-ide HOM-a.arr-char
              by auto
          qed
        qed
      qed
    qed
    show ?thesis
      using FG.inverse-simulations-axioms isomorphic-rts-def by blast
  qed

end
```

4.1.3 Additional Notions

An RTS-category is *locally small* if each of the hom-RTS's is a small RTS.

```
locale locally-small-rts-category =
  rts-category +
assumes small-homs: ⟦obj a; obj b⟧ ⟹ small (H.hom a b)
begin

  lemma HOM-is-small-extensional-rts:
  assumes obj a and obj b
  shows HOM a b ∈ Collect extensional-rts ∩ Collect small-rts
  proof −
    interpret HOM: sub-rts resid ‹λt. t ∈ H.hom a b›
      using assms sub-rts-HOM by fastforce
    interpret HOM: small-rts HOM.resid
      using assms small-homs [of a b] smaller-than-small HOM.arr-char
      apply unfold-locales
      by (simp add: smaller-than-small subset-eq)
    show ?thesis
      using HOM.preserves-extensional-rts V.extensional-rts-axioms
```

```
        HOM.small-rts-axioms
    by auto
  qed

end
```

An *RTS-functor* is a mapping between RTS-categories that is functor with respect to the horizontal composition and a simulation with respect to the vertical residuation. An *RTS-category isomorphism* is an RTS-functor that is invertible as a simulation, from which it follows that it is also invertible as a functor.

```
locale rts-functor =
  A: rts-category resid_A comp_A +
  B: rts-category resid_B comp_B +
  functor comp_A comp_B F +
  simulation resid_A resid_B F
for resid_A :: 'a resid  (infix \_A 70)
and comp_A :: 'a comp    (infixr ⋆_A 53)
and resid_B :: 'b resid  (infix \_B 70)
and comp_B :: 'b comp    (infixr ⋆_B 53)
and F :: 'a ⇒ 'b
begin

  notation A.V.con  (infix ⌢_A 50)
  notation B.V.con  (infix ⌢_B 50)

  lemma is-invertible-simulation-if:
  assumes invertible-functor comp_A comp_B F
  and ⋀t u. F t ⌢_B F u ⟹ t ⌢_A u
  shows invertible-simulation resid_A resid_B F
  proof −
    obtain G where G: inverse-functors comp_A comp_B G F
      using assms(1) invertible-functor.invertible by blast
    interpret FG: inverse-functors comp_A comp_B G F
      using G by blast
    interpret FG: inverse-simulations resid_A resid_B G F
    proof
      show ⋀t. ¬ B.arr t ⟹ G t = A.null
        using FG.F.is-extensional by simp
      show inv: F ∘ G = I resid_B and inv': G ∘ F = I resid_A
        using FG.inv FG.inv' A.H.map-def B.H.map-def by auto
      fix t u
      assume tu: t ⌢_B u
      have F (G t) ⌢_B F (G u)
        using tu inv
        by (metis B.V.con-implies-arr(1−2) comp-apply)
      thus G t ⌢_A G u
        using assms(2) by blast
      show G (t \_B u) = G t \_A G u
```

```
      by (metis A.V.arr-resid B.V.con-implies-arr(1-2)
          ‹G t ⌢_A G u› comp-apply inv inv' preserves-resid tu)
  qed
  show ?thesis
    using invertible-simulation-def' FG.inverse-simulations-axioms by auto
qed

lemma is-invertible-if:
assumes invertible-simulation resid_A resid_B F
shows invertible-functor comp_A comp_B F
proof -
  obtain G where G: inverse-simulations resid_A resid_B G F
    using assms(1) invertible-simulation-def' by blast
  interpret FG: inverse-simulations resid_A resid_B G F
    using G by blast
  interpret FG: inverse-functors comp_A comp_B G F
  proof
    show ⋀f. ¬ B.H.arr f ⟹ G f = A.H.null
      using FG.F.extensional by auto
    show 1: ⋀f. B.H.arr f ⟹ A.H.arr (G f)
      by auto
    show 2: ⋀f. B.H.arr f ⟹ A.H.dom (G f) = G (B.H.dom f)
      by (metis 1 A.H.arr-dom-iff-arr A.arr-coincidence B.arr-coincidence
          FG.inv FG.inv' o-apply preserves-dom)
    show 3: ⋀f. B.H.arr f ⟹ A.H.cod (G f) = G (B.H.cod f)
      by (metis 1 A.H.arr-cod-iff-arr A.arr-coincidence B.arr-coincidence
          FG.inv FG.inv' o-apply preserves-cod)
    show F ∘ G = B.H.map
      using B.H.map-def FG.inv by auto
    show G ∘ F = A.H.map
      using A.H.map-def FG.inv' by auto
    show ⋀g f. B.H.seq g f ⟹ G (g ⋆_B f) = G g ⋆_A G f
      by (metis (full-types) 1 2 3 A.arr-coincidence B.H.seqE B.arr-coincidence
          FG.inv-simp FG.inv'-simp as-nat-trans.preserves-comp-2 A.H.seqI)
  qed
  show ?thesis
    using FG.inverse-functors-axioms
    by unfold-locales blast
qed

end

locale rts-category-isomorphism =
  rts-functor +
  invertible-simulation resid_A resid_B F
begin

  sublocale invertible-functor comp_A comp_B F
    using invertible-simulation-axioms is-invertible-if by simp
```

end

end

theory *ConcreteRTSCategory*
imports *Main RTSCategory*
begin

4.2 Concrete RTS-Categories

If we are given a set *Obj* of "objects", a mapping *Hom* that assigns to every two objects A and B an extensional "hom-RTS" RTS *Hom A B*, a mapping *Id* that assigns to each object A an "identity arrow" *Id A* of *Hom A B*, and a mapping *Comp* that assigns to every three objects A, B, C a "composition law" *Comp A B C* from *Hom B C* $\times$ *Hom A B* to *Hom A B*, subject to suitable identity and associativity conditions, then we can form from this data an RTS-category whose underlying set of arrows is the disjoint union of the sets of arrows of the hom-RTS's.

locale *concrete-rts-category* =
fixes *obj-type* :: *'O itself*
and *arr-type* :: *'A itself*
and *Obj* :: *'O set*
and *Hom* :: *'O* $\Rightarrow$ *'O* $\Rightarrow$ *'A resid*
and *Id* :: *'O* $\Rightarrow$ *'A*
and *Comp* :: *'O* $\Rightarrow$ *'O* $\Rightarrow$ *'O* $\Rightarrow$ *'A* $\Rightarrow$ *'A* $\Rightarrow$ *'A*
assumes *rts-Hom*: $[\![A \in Obj;\ B \in Obj]\!] \Longrightarrow$ *extensional-rts* (*Hom A B*)
and *binary-simulation-Comp*:
$[\![A \in Obj;\ B \in Obj;\ C \in Obj]\!]$
$\Longrightarrow$ *binary-simulation*
(*Hom B C*) (*Hom A B*) (*Hom A C*) ($\lambda(t,\ u).$ *Comp A B C t u*)
and *ide-Id*: $A \in Obj \Longrightarrow$ *residuation.ide* (*Hom A A*) (*Id A*)
and *Comp-Hom-Id*: $[\![A \in Obj;\ B \in Obj;$ *residuation.arr* (*Hom A B*) $t]\!]$
$\Longrightarrow$ *Comp A A B t* (*Id A*) = *t*
and *Comp-Id-Hom*: $[\![A \in Obj;\ B \in Obj;$ *residuation.arr* (*Hom A B*) $u]\!]$
$\Longrightarrow$ *Comp A B B* (*Id B*) *u* = *u*
and *Comp-assoc*: $[\![A \in Obj;\ B \in Obj;\ C \in Obj;\ D \in Obj;$
residuation.arr (*Hom C D*) *t*; *residuation.arr* (*Hom B C*) *u*;
residuation.arr (*Hom A B*) *v* $]\!] \Longrightarrow$
Comp A B D (*Comp B C D t u*) *v* =
Comp A C D t (*Comp A B C u v*)
begin

datatype (*'o*, *'a*) *arr* =
Null
| *MkArr 'o 'o 'a*

```
fun Dom :: ('O, 'A) arr ⇒ 'O
where Dom (MkArr a - -) = a
   | Dom - = undefined

fun Cod :: ('O, 'A) arr ⇒ 'O
where Cod (MkArr - b -) = b
   | Cod - = undefined

fun Trn :: ('O, 'A) arr ⇒ 'A
where Trn (MkArr - - t) = t
   | Trn - = undefined

abbreviation Arr :: ('O, 'A) arr ⇒ bool
where Arr ≡ λt. t ≠ Null ∧ Dom t ∈ Obj ∧ Cod t ∈ Obj ∧
              residuation.arr (Hom (Dom t) (Cod t)) (Trn t)

abbreviation Ide :: ('O, 'A) arr ⇒ bool
where Ide ≡ λt. t ≠ Null ∧ Dom t ∈ Obj ∧ Cod t ∈ Obj ∧
              residuation.ide (Hom (Dom t) (Cod t)) (Trn t)

abbreviation Con :: ('O, 'A) arr ⇒ ('O, 'A) arr ⇒ bool
where Con t u ≡ Arr t ∧ Arr u ∧ Dom t = Dom u ∧ Cod t = Cod u ∧
              residuation.con (Hom (Dom t) (Cod t)) (Trn t) (Trn u)

fun resid  (infix \ 70)
where resid Null u = Null
   | resid t Null = Null
   | resid t u =
     (if Con t u
      then MkArr (Dom t) (Cod t) (Hom (Dom t) (Cod t) (Trn t) (Trn u))
      else Null)

sublocale V: ResiduatedTransitionSystem.partial-magma resid
  apply unfold-locales
  by (metis Trn.cases resid.simps(1−2))

lemma null-char:
shows V.null = Null
  by (metis V.null-is-zero(2) resid.simps(1))

sublocale V: residuation resid
proof
  fix t u :: ('O, 'A) arr
  assume tu: t \ u ≠ V.null
  interpret hom: extensional-rts ‹Hom (Dom t) (Cod t)›
    using tu null-char rts-Hom
    by (metis arr.exhaust V.null-is-zero(2) resid.simps(1,3))
  show t \ u ≠ V.null ⟹ u \ t ≠ V.null
```

```
    using null-char hom.con-sym
    apply (cases t; cases u)
      apply auto[3]
    by (metis arr.simps(2) resid.simps(3))
  show (t \ u) \ (t \ u) ≠ V.null
    using tu null-char hom.arr-resid hom.con-arr-self
    apply (cases t \ u)
     apply force
    apply (cases t; cases u)
      apply auto[3]
    by (metis Cod.simps(1) Dom.simps(1) Trn.simps(1)
        arr.simps(2) resid.simps(3))
  next
  fix t u v :: ('O, 'A) arr
  assume tuv: (v \ t) \ (u \ t) ≠ V.null
  have tu: Con t u
    using tuv null-char V.null-is-zero(2) resid.simps(3)
    apply (cases t; cases u; cases v)
          apply auto[7]
    by (metis extensional-rts-def residuation.con-sym
        rts-Hom rts-def)
  have tv: Con t v
    using tu tuv null-char resid.simps(1,3)
    apply (cases t; cases u; cases v)
          apply auto[7]
    by (metis extensional-rts-def residuation.con-sym
        rts-Hom rts-def)
  interpret hom: extensional-rts ‹Hom (Dom t) (Cod t)›
    using tu null-char rts-Hom by metis
  have uv: Con u v
    using tu tv tuv null-char hom.con-sym
          resid.simps(1,3) rts-Hom
    apply (cases t; cases u; cases v)
          apply auto[7]
    apply (intro conjI)
            apply auto[10]
    by auto (metis Cod.simps(1) Dom.simps(1) hom.resid-reflects-con)
  interpret hom: extensional-rts ‹Hom (Dom t) (Cod t)›
    using tu null-char rts-Hom by blast
  show (v \ t) \ (u \ t) = (v \ u) \ (t \ u)
    using tu tv uv null-char hom.arr-resid-iff-con
          hom.con-sym hom.cube
    apply (cases t; cases u; cases v)
    by auto metis+
qed

notation V.con  (infix ⌢ 50)

lemma con-char:
```

shows $t \frown u \longleftrightarrow Con\ t\ u$
proof
  show $t \frown u \Longrightarrow Con\ t\ u$
    using *V.con-def V.not-con-null*(*2*) *null-char*
    apply (*cases t*; *cases u*)
      apply *auto*[*3*]
    by *fastforce*
  show $Con\ t\ u \Longrightarrow t \frown u$
    using *null-char*
    by (*cases t*; *cases u*) *auto*
qed

lemma *arr-char*:
shows $V.arr\ t \longleftrightarrow Arr\ t$
  by (*metis V.arr-def con-char extensional-rts-def*
      residuation.arr-def rts-Hom rts-def)

lemma *arrI* [*intro*]:
assumes $t \neq V.null$ **and** $Dom\ t \in Obj$ **and** $Cod\ t \in Obj$
and *residuation.arr* (*Hom* (*Dom t*) (*Cod t*)) (*Trn t*)
shows $V.arr\ t$
  using *assms arr-char null-char* **by** *auto*

lemma *arrE* [*elim*]:
assumes $V.arr\ t$
and ⟦$t \neq V.null$; $Dom\ t \in Obj$; $Cod\ t \in Obj$;
      residuation.arr (*Hom* (*Dom t*) (*Cod t*)) (*Trn t*)⟧
        $\Longrightarrow T$
shows T
  using *assms arr-char* **by** *fastforce*

lemma *sta-char*:
shows $V.ide\ t \longleftrightarrow Ide\ t$
proof
  assume *t*: $V.ide\ t$
  interpret *hom*: *extensional-rts* ‹*Hom* (*Dom t*) (*Cod t*)›
    using *t*
    by (*meson V.ide-implies-arr arrE rts-Hom*)
  show $Ide\ t$
    using *t null-char V.not-ide-null*
    apply (*cases t*)
     apply *force*
    by (*metis Trn.simps*(*1*) *V.ideE hom.ideI resid.simps*(*3*))
  next
  assume *t*: $Ide\ t$
  show $V.ide\ t$
  proof (*cases t*)
    show $t = Null \Longrightarrow ?thesis$
      using *t V.ide-def con-char* **by** *blast*

```
    fix A B F
    assume 1: t = MkArr A B F
    interpret hom: extensional-rts ‹Hom A B›
      using t 1 rts-Hom by simp
    show V.ide t
      using t 1 V.con-def V.ide-def null-char by auto
  qed
qed

lemma staI [intro]:
assumes t ≠ V.null and Dom t ∈ Obj and Cod t ∈ Obj
and residuation.ide (Hom (Dom t) (Cod t)) (Trn t)
shows V.ide t
  using assms sta-char null-char by auto

lemma staE [elim]:
assumes V.ide t
and ⟦t ≠ V.null; Dom t ∈ Obj; Cod t ∈ Obj;
      residuation.ide (Hom (Dom t) (Cod t)) (Trn t)⟧
        ⟹ T
shows T
  using assms sta-char by fastforce

lemma trg-char:
shows V.trg t =
      (if V.arr t
       then MkArr (Dom t) (Cod t)
            (residuation.trg (Hom (Dom t) (Cod t)) (Trn t))
       else Null)
proof (cases V.arr t)
  show ¬ V.arr t ⟹ ?thesis
    using V.trg-def null-char V.con-def by auto
  assume t: V.arr t
  interpret hom: extensional-rts ‹Hom (Dom t) (Cod t)›
    using t
    by (meson V.ide-implies-arr arrE rts-Hom)
  show ?thesis
     using t null-char
     apply (cases t)
      apply fastforce
     by (metis V.trg-def arrE hom.conI hom.ide-trg
         hom.not-ide-null hom.trg-def resid.simps(3))
qed

lemma con-implies-Par:
assumes t ⌢ u
shows Dom t = Dom u and Cod t = Cod u
  using assms con-char by blast+
```

lemma *Dom-resid* [*simp*]:
assumes $t \frown u$
shows $Dom\ (t \setminus u) = Dom\ t$
using *assms con-char*
by (*cases t*; *cases u*) *auto*

lemma *Cod-resid* [*simp*]:
assumes $t \frown u$
shows $Cod\ (t \setminus u) = Cod\ t$
using *assms con-char*
by (*cases t*; *cases u*) *auto*

lemma *Trn-resid* [*simp*]:
assumes $t \frown u$
shows $Trn\ (t \setminus u) = Hom\ (Dom\ t)\ (Cod\ t)\ (Trn\ t)\ (Trn\ u)$
using *assms con-char*
by (*cases t*; *cases u*) *auto*

sublocale *rts resid*
proof
show $\bigwedge t.\ V.arr\ t \Longrightarrow V.ide\ (V.trg\ t)$
by (*simp add*: *arr-char extensional-rts.axioms*(*1*) *rts.ide-trg*
rts-Hom sta-char trg-char)
show *1*: $\bigwedge a\ t.\ [\![V.ide\ a;\ t \frown a]\!] \Longrightarrow t \setminus a = t$
proof −
fix *a t*
assume *a*: $V.ide\ a$
assume *con*: $t \frown a$
have $t \setminus a \neq Null \wedge t \neq Null$
using *con null-char* **by** *auto*
moreover have $Dom\ (t \setminus a) = Dom\ t$
using *a con* **by** *simp*
moreover have $Cod\ (t \setminus a) = Cod\ t$
using *a con* **by** *simp*
moreover have $Trn\ (t \setminus a) = Trn\ t$
using *a con* **apply** *simp*
by (*metis con-char extensional-rts-def sta-char rts.resid-arr-ide rts-Hom*)
ultimately show $t \setminus a = t$
by (*metis Cod.elims Dom.simps*(*1*) *Trn.simps*(*1*))
qed
show $\bigwedge a\ t.\ [\![V.ide\ a;\ a \frown t]\!] \Longrightarrow V.ide\ (a \setminus t)$
using *sta-char con-char*
by (*metis 1 V.arr-resid V.con-arr-self V.con-sym V.cube V.ideE V.ideI*)
show $\bigwedge t\ u.\ t \frown u \Longrightarrow \exists a.\ V.ide\ a \wedge a \frown t \wedge a \frown u$
proof −
fix *t u*
assume *tu*: $t \frown u$
interpret *hom*: *extensional-rts* ‹*Hom* (*Dom t*) (*Cod t*)›
using *tu con-char arr-char* [*of t*] *rts-Hom* **by** *auto*

```
    have 1: hom.con (Trn t) (Trn u)
      using tu con-char by auto
    obtain α where α: hom.ide α ∧ hom.con α (Trn t) ∧ hom.con α (Trn u)
      using 1 hom.con-imp-coinitial-ax by auto
    have V.ide (MkArr (Dom t) (Cod t) α)
      using tu α V.con-implies-arr arr-char sta-char by auto
    moreover have MkArr (Dom t) (Cod t) α ⌢ t
      using α tu con-char hom.ide-implies-arr by auto
    moreover have MkArr (Dom t) (Cod t) α ⌢ u
      using α tu con-char hom.ide-implies-arr by auto
    ultimately show ∃a. V.ide a ∧ a ⌢ t ∧ a ⌢ u by blast
  qed
  show ⋀t u v. ⟦V.ide (t \ u); u ⌢ v⟧ ⟹ t \ u ⌢ v \ u
  proof −
    fix t u v
    assume tu: V.ide (t \ u)
    assume uv: u ⌢ v
    have 1: t \ u ≠ V.null
      using tu by auto
    interpret HOM: extensional-rts ‹Hom (Dom t) (Cod t)›
      using 1 con-char arr-char [of t] rts-Hom by auto
    have 2: HOM.con (Hom (Dom t) (Cod t) (Trn t) (Trn u))
                    (Hom (Dom t) (Cod t) (Trn v) (Trn u))
      by (metis (no-types, lifting) Cod-resid Dom-resid HOM.con-target
          Trn-resid V.conI con-char staE tu uv)
    have 3: Con t u
      using 1 con-char by blast
    have 4: Con v u
      using uv con-char V.con-sym by blast
    have 5: t \ u = MkArr (Dom t) (Cod t)
                          (Hom (Dom t) (Cod t) (Trn t) (Trn u))
      using 1 3
      by (cases t; cases u) auto
    have 6: v \ u = MkArr (Dom v) (Cod v)
                          (Hom (Dom v) (Cod t) (Trn v) (Trn u))
      using 3 4
      by (cases v; cases u) auto
    show t \ u ⌢ v \ u
      using 2 3 4 5 6 HOM.con-implies-arr(1−2) con-char by auto
  qed
qed

lemma is-rts:
shows rts resid
  ..

sublocale V: extensional-rts resid
proof
  fix t u
```

```
  assume tu: cong t u
  have 1: t \ u ≠ V.null
    using tu by auto
  interpret HOM: extensional-rts ‹Hom (Dom t) (Cod t)›
    using 1 con-char arr-char [of t] rts-Hom by auto
  have t ≠ Null ∧ u ≠ Null
    using tu con-char by blast
  moreover have Dom t = Dom u
    using tu con-char by blast
  moreover have Cod t = Cod u
    using tu con-char by blast
  moreover have Trn t = Trn u
    by (metis calculation(2−3) Cod-resid Dom-resid Trn-resid sta-char
        congE HOM.cong-char tu)
  ultimately show t = u
    by (metis Cod.elims Dom.simps(1) Trn.simps(1))
qed

lemma is-extensional-rts:
shows extensional-rts resid
  ..

lemma arr-MkArr [intro]:
assumes a ∈ Obj and b ∈ Obj
and residuation.arr (Hom a b) t
shows V.arr (MkArr a b t)
  using assms arr-char by auto

lemma arr-eqI:
assumes t ≠ Null and u ≠ Null
and Dom t = Dom u and Cod t = Cod u and Trn t = Trn u
shows t = u
  using assms null-char
  by (metis Cod.elims Dom.simps(1) Trn.simps(1))

lemma MkArr-Trn:
assumes V.arr t
shows MkArr (Dom t) (Cod t) (Trn t) = t
  using assms null-char V.not-arr-null
  by (intro arr-eqI) auto

lemma src-char:
shows V.src t = (if V.arr t
                 then MkArr (Dom t) (Cod t)
                      (weakly-extensional-rts.src (Hom (Dom t) (Cod t)) (Trn t))
                 else Null)
proof (cases V.arr t)
  show ¬ V.arr t ⟹ ?thesis
    by (simp add: V.src-def null-char)
```

```
  assume t: V.arr t
  interpret HOM: extensional-rts ‹Hom (Dom t) (Cod t)›
    using t con-char arr-char rts-Hom by auto
  show ?thesis
    using t
    by (metis (no-types, lifting) Cod.simps(1) Dom.simps(1) HOM.src-eqI
        Trn.simps(1) V.con-arr-src(2) V.ide-src arr.simps(2) arr-eqI
        con-char staE)
qed

definition hcomp  (infixr ⋆ 53)
where t ⋆ u ≡ if V.arr t ∧ V.arr u ∧ Dom t = Cod u
              then MkArr (Dom u) (Cod t)
                         (Comp (Dom u) (Cod u) (Cod t) (Trn t) (Trn u))
              else V.null

lemma arr-hcomp_CRC:
assumes V.arr t and V.arr u and Dom t = Cod u
shows V.arr (t ⋆ u)
proof −
  let ?a = Dom u and ?b = Cod u and ?c = Cod t
  interpret Comp: binary-simulation ‹Hom ?b ?c› ‹Hom ?a ?b› ‹Hom ?a ?c›
                  ‹λ(t, u). Comp ?a ?b ?c t u›
    using assms arr-char binary-simulation-Comp by auto
  let ?tu = MkArr ?a ?c (Comp ?a ?b ?c (Trn t) (Trn u))
  have V.arr ?tu
    using assms arr-char Comp.preserves-reflects-arr [of (Trn t, Trn u)]
    by simp
  thus ?thesis
    using assms hcomp-def by simp
qed

lemma Dom-hcomp [simp]:
assumes V.arr t and V.arr u and Dom t = Cod u
shows Dom (t ⋆ u) = Dom u
  using assms hcomp-def by auto

lemma Cod-hcomp [simp]:
assumes V.arr t and V.arr u and Dom t = Cod u
shows Cod (t ⋆ u) = Cod t
  using assms hcomp-def by auto

lemma Trn-hcomp [simp]:
assumes V.arr t and V.arr u and Dom t = Cod u
shows Trn (t ⋆ u) = Comp (Dom u) (Cod u) (Cod t) (Trn t) (Trn u)
  using assms hcomp-def by auto

lemma hcomp-Null [simp]:
shows t ⋆ Null = Null and Null ⋆ u = Null
```

using *hcomp-def null-char* **by** *fastforce+*

sublocale H: *Category.partial-magma hcomp*
using *hcomp-def V.not-arr-null*
by *unfold-locales metis*

lemma *null-coincidence*$_{CRC}$ [*simp*]:
shows $H.null = V.null$
using *hcomp-def*
by (*metis H.null-eqI V.not-arr-null*)

sublocale H: *partial-composition hcomp* **..**

lemma *H-composable-char*:
shows $t \star u \neq V.null \longleftrightarrow V.arr\ t \wedge V.arr\ u \wedge Dom\ t = Cod\ u$
using *hcomp-def null-char* **by** *auto*

lemma *objI* [*intro*]:
assumes $t \neq V.null$
and $Dom\ t \in Obj$ **and** $Cod\ t = Dom\ t$ **and** $Trn\ t = Id\ (Dom\ t)$
shows $H.ide\ t$
using *assms*
by (*metis Comp-Hom-Id Comp-Id-Hom H.ide-def MkArr-Trn*
V.ide-implies-arr arrE hcomp-def ide-Id null-coincidence$_{CRC}$ *staI*)

lemma *objE* [*elim*]:
assumes $H.ide\ a$
and $[\![a \neq V.null;\ Dom\ a \in Obj;\ Cod\ a = Dom\ a;\ Trn\ a = Id\ (Dom\ a)]\!] \Longrightarrow T$
shows T
using *assms*
by (*metis Cod.simps*(*1*) *Dom.simps*(*1*) *H.ide-def H-composable-char*
Trn.simps(*1*) *arr.simps*(*2*) *arr-char null-char null-coincidence*$_{CRC}$ *objI*)

definition *mkobj*
where $mkobj\ A \equiv MkArr\ A\ A\ (Id\ A)$

lemma *mkobj-simps* [*simp*]:
shows $Dom\ (mkobj\ A) = A$ **and** $Cod\ (mkobj\ A) = A$
and $Trn\ (mkobj\ A) = Id\ A$
using *mkobj-def* **by** *auto*

lemma *obj-mkobj*:
assumes $A \in Obj$
shows $H.ide\ (mkobj\ A)$
using *assms*
by (*simp add*: *mkobj-def null-char objI*)

lemma *obj-char*:
shows $H.ide\ a \longleftrightarrow V.arr\ a \wedge mkobj\ (Dom\ a) = a$

by (*metis H.ide-def H-composable-char MkArr-Trn arrE mkobj-def objE obj-mkobj*)

lemma *obj-is-sta*:
assumes *H.ide a*
shows *V.ide a*
using *assms ide-Id* **by** *fastforce*

lemma *obj-simps*:
assumes *H.ide a*
shows *Cod a = Dom a* **and** *Trn a = Id (Dom a)*
using *assms* **by** *auto*

lemma *domains-char*:
shows *H.domains t = {a. V.arr t ∧ mkobj (Dom t) = a}*
unfolding *H.domains-def*
using *H-composable-char obj-char obj-mkobj obj-simps(1)* **by** *auto*

lemma *codomains-char*:
shows *H.codomains t = {a. V.arr t ∧ mkobj (Cod t) = a}*
unfolding *H.codomains-def*
using *H-composable-char obj-char obj-mkobj obj-simps(1)* **by** *auto*

lemma *H-arr-char*:
shows *H.arr t ⟷ t ≠ Null ∧ Dom t ∈ Obj ∧ Cod t ∈ Obj ∧*
residuation.arr (Hom (Dom t) (Cod t)) (Trn t)
using *H.arr-def codomains-char domains-char arr-char* **by** *force*

lemma *H-arrI* [*intro*]:
assumes *t ≠ V.null* **and** *Dom t ∈ Obj* **and** *Cod t ∈ Obj*
and *residuation.arr (Hom (Dom t) (Cod t)) (Trn t)*
shows *H.arr t*
using *assms H-arr-char null-char* **by** *auto*

lemma *H-seq-char*:
shows *H.seq t u ⟷ V.arr t ∧ V.arr u ∧ Dom t = Cod u*
by (*metis H-arr-char H-composable-char arr-char arr-hcomp$_{CRC}$ null-char*)

lemma *H-seqI* [*intro*]:
assumes *V.arr t* **and** *V.arr u* **and** *Dom t = Cod u*
shows *H.seq t u*
using *assms H-seq-char* **by** *blast*

sublocale *H*: *category hcomp*
proof
show *⋀t u. hcomp t u ≠ H.null ⟹ H.seq t u*
using *hcomp-def H-seq-char null-char*
by (*metis null-coincidence$_{CRC}$*)
show *⋀t. (H.domains t ≠ {}) = (H.codomains t ≠ {})*

by (*simp add*: *codomains-char domains-char*)
show $\bigwedge h\ g\ f.\ [\![H.seq\ h\ g;\ H.seq\ (h \star g)\ f]\!] \Longrightarrow H.seq\ g\ f$
using *H-seq-char*
by (*metis Dom.simps*(*1*) *hcomp-def*)
show $\bigwedge h\ g\ f.\ [\![H.seq\ h\ (g \star f);\ H.seq\ g\ f]\!] \Longrightarrow H.seq\ h\ g$
using *H-seq-char*
by (*metis Cod.simps*(*1*) *hcomp-def*)
show $\bigwedge g\ f\ h.\ [\![H.seq\ g\ f;\ H.seq\ h\ g]\!] \Longrightarrow H.seq\ (h \star g)\ f$
using *H-seq-char*
by (*metis Dom.simps*(*1*) *arr-hcomp*$_{CRC}$ *hcomp-def*)
show $\bigwedge t\ u\ v.\ [\![H.seq\ u\ v;\ H.seq\ t\ u]\!] \Longrightarrow (t \star u) \star v = t \star u \star v$
using *arr-hcomp*$_{CRC}$ *H-seq-char H-composable-char null-char Comp-assoc*
apply (*intro arr-eqI*)
apply *auto*[*4*]
apply *auto*
using *H-arr-char arr-char* **by** *auto*
qed

lemma *is-category*:
shows *category hcomp*
..

lemma *arr-coincidence*$_{CRC}$ [*simp*]:
shows $H.arr = V.arr$
using *H-arr-char arr-char* **by** *blast*

lemma *dom-char*:
shows $H.dom = (\lambda t.\ if\ H.arr\ t\ then\ mkobj\ (Dom\ t)\ else\ V.null)$
using *domains-char H.dom-in-domains H.has-domain-iff-arr H.dom-def*
by *auto*

lemma *dom-mkobj* [*simp*]:
assumes $A \in Obj$
shows $H.dom\ (mkobj\ A) = mkobj\ A$
using *assms obj-mkobj* **by** *auto*

lemma *mkobj-Dom* [*simp*]:
assumes *H.ide a*
shows $mkobj\ (Dom\ a) = H.dom\ a$
using *assms obj-char* **by** *auto*

lemma *cod-char*:
shows $H.cod = (\lambda t.\ if\ H.arr\ t\ then\ mkobj\ (Cod\ t)\ else\ V.null)$
using *codomains-char H.cod-in-codomains H.has-codomain-iff-arr H.cod-def*
by *auto*

lemma *cod-mkobj* [*simp*]:
assumes $A \in Obj$
shows $H.cod\ (mkobj\ A) = mkobj\ A$

using *assms obj-mkobj* **by** *auto*

lemma *Dom-dom* [*simp*]:
assumes *V.arr t*
shows *Dom* (*H.dom t*) = *Dom t*
using *assms dom-char mkobj-def* **by** *auto*

lemma *Dom-cod* [*simp*]:
assumes *V.arr t*
shows *Dom* (*H.cod t*) = *Cod t*
using *assms cod-char mkobj-def* **by** *auto*

lemma *Cod-dom* [*simp*]:
assumes *V.arr t*
shows *Cod* (*H.dom t*) = *Dom t*
using *assms dom-char mkobj-def* **by** *auto*

lemma *Cod-cod* [*simp*]:
assumes *V.arr t*
shows *Cod* (*H.cod t*) = *Cod t*
using *assms cod-char mkobj-def* **by** *auto*

lemma *con-implies-par*:
assumes *V.con t u*
shows *H.par t u*
using *assms cod-char dom-char V.con-implies-arr*(*1−2*) *con-implies-Par*(*1−2*)
by *auto*

lemma *par-resid*:
assumes *V.con t u*
shows *H.par t* (*resid t u*)
using *assms cod-char dom-char V.con-implies-arr*(*1−2*) *con-implies-Par*(*1−2*)
by *auto*

lemma *simulation-dom*:
shows *simulation resid resid H.dom*
using *dom-char arr-char H-arr-char con-char*
apply *unfold-locales*
apply *presburger*
apply (*metis* (*no-types*, *lifting*) *H.arr-dom V.arrE*)
by (*metis* (*no-types*, *lifting*) *H.ide-dom par-resid V.ideE obj-is-sta*)

lemma *simulation-cod*:
shows *simulation resid resid H.cod*
using *cod-char arr-char H-arr-char con-char*
apply *unfold-locales*
apply *presburger*
apply (*metis* (*no-types*, *lifting*) *H.arr-cod V.arr-def*)
by (*metis* (*no-types*, *lifting*) *H.ide-cod par-resid V.ideE obj-is-sta*)

sublocale *dom*: *simulation resid resid H.dom*
using *simulation-dom* **by** *blast*
sublocale *cod*: *simulation resid resid H.cod*
using *simulation-cod* **by** *blast*

sublocale *VV*: *fibered-product-rts resid resid resid H.dom H.cod* **..**

sublocale H_{VV}: *simulation VV.resid resid*
‹λ*t*. *if VV.arr t then fst t ⋆ snd t else V.null*›
proof
let *?C* = λ*t*. *if VV.arr t then fst t ⋆ snd t else V.null*
show ⋀*t*. ¬ *VV.arr t* ⟹ *?C t* = *V.null*
by *simp*
fix *t u*
assume *tu*: *VV.con t u*
have *arr-t*: *VV.arr t* **and** *arr-u*: *VV.arr u*
using *tu VV.con-implies-arr* **by** *blast+*
have *t*: *V.arr* (*fst t*) ∧ *V.arr* (*snd t*) ∧ *Dom* (*fst t*) = *Cod* (*snd t*)
by (*metis Cod-cod Cod-dom VV.arr-char arr-t*)
have *u*: *V.arr* (*fst u*) ∧ *V.arr* (*snd u*) ∧ *Dom* (*fst u*) = *Cod* (*snd u*)
by (*metis Cod-cod Cod-dom VV.arr-char arr-u*)
let *?a* = *Dom* (*snd t*) **and** *?b* = *Cod* (*snd t*) **and** *?c* = *Cod* (*fst t*)
have *a*: *?a* ∈ *Obj* **and** *b*: *?b* ∈ *Obj* **and** *c*: *?c* ∈ *Obj*
using *tu VV.con-char VV.con-implies-arr sta-char arr-char* **by** *fast+*
interpret *AB*: *extensional-rts* ‹*Hom ?a ?b*›
using *a b rts-Hom* **by** *simp*
interpret *BC*: *extensional-rts* ‹*Hom ?b ?c*›
using *b c rts-Hom* **by** *simp*
interpret *AC*: *extensional-rts* ‹*Hom ?a ?c*›
using *a c rts-Hom* **by** *simp*
interpret *BCxAB*: *product-rts* ‹*Hom ?b ?c*› ‹*Hom ?a ?b*› **..**
interpret *Comp*: *binary-simulation* ‹*Hom ?b ?c*› ‹*Hom ?a ?b*› ‹*Hom ?a ?c*›
‹λ(*t'*, *u'*). *Comp ?a ?b ?c t' u'*›
using *a b c binary-simulation-Comp* **by** *simp*
have *0*: *BCxAB.con* (*Trn* (*fst t*), *Trn* (*snd t*)) (*Trn* (*fst u*), *Trn* (*snd u*))
by (*metis BCxAB.con-char VV.con-char con-char fst-conv snd-conv t tu*)
have *1*: *Dom* (*snd u*) = *?a* **and** *2*: *Cod* (*fst u*) = *?c*
and *3*: *Cod* (*snd u*) = *?b*
using *VV.con-char con-char tu* **by** *metis+*
show *4*: *?C t* ⌢ *?C u*
using *a c t u tu 0 1 2 3 VV.arr-char VV.con-char hcomp-def*
AC.con-implies-arr con-char
Comp.preserves-con
[*of* (*Trn* (*fst t*), *Trn* (*snd t*)) (*Trn* (*fst u*), *Trn* (*snd u*))]
by *simp*
show *?C* (*VV.resid t u*) = *?C t* \ *?C u*
proof −
have *Con*: *VV.Con t u*

```
      using tu VV.con-char by blast
    have ?C (VV.resid t u) = fst t \ fst u ⋆ snd t \ snd u
      using tu Con VV.arr-char VV.arr-resid VV.resid-def by auto
    also have ... = (fst t ⋆ snd t) \ (fst u ⋆ snd u)
        using t u tu arr-t arr-u 0 1 2 3 4 Con VV.con-char null-char
              H-composable-char BCxAB.resid-def Comp.preserves-resid
        by (intro arr-eqI) auto
    also have ... = ?C t \ ?C u
      using arr-t arr-u by auto
    finally show ?thesis by blast
  qed
qed

lemma simulation-hcomp:
shows simulation VV.resid resid (λt. if VV.arr t then fst t ⋆ snd t else V.null)
  ..

lemma Dom-src [simp]:
assumes V.arr t
shows Dom (V.src t) = Dom t
  using assms con-implies-Par(1) by simp

lemma Dom-trg [simp]:
assumes V.arr t
shows Dom (V.trg t) = Dom t
  using assms V.trg-def by simp

lemma Cod-src [simp]:
assumes V.arr t
shows Cod (V.src t) = Cod t
  using assms con-implies-Par(2) by simp

lemma Cod-trg [simp]:
assumes V.arr t
shows Cod (V.trg t) = Cod t
  using assms V.trg-def by simp

lemma dom-src_CRC [simp]:
shows H.dom (V.src t) = H.dom t
 by (metis V.arr-src-iff-arr V.con-arr-src(2) con-implies-par dom.extensional)

lemma dom-trg_CRC [simp]:
shows H.dom (V.trg t) = H.dom t
  by (metis H.dom-dom V.arr-def V.trg-def dom.extensional null-char
      par-resid trg-char)

lemma cod-src_CRC [simp]:
shows H.cod (V.src t) = H.cod t
  by (simp add: cod-char V.arr-src-iff-arr)
```

lemma *cod-trg*$_{CRC}$ [*simp*]:
shows *H.cod* (*V.trg t*) = *H.cod t*
by (*simp add*: *cod-char V.arr-trg-iff-arr*)

lemma *src-dom*$_{CRC}$ [*simp*]:
shows *V.src* (*H.dom t*) = *H.dom t*
by (*metis H.ide-dom dom-char V.src-ide arr-char src-char null-char obj-is-sta*)

lemma *trg-dom*$_{CRC}$ [*simp*]:
shows *V.trg* (*H.dom t*) = *H.dom t*
by (*metis H.cod-dom V.ide-iff-src-self V.trg-ide cod.extensional*
null-char src-dom$_{CRC}$ *trg-char*)

lemma *src-cod*$_{CRC}$ [*simp*]:
shows *V.src* (*H.cod t*) = *H.cod t*
by (*metis H.dom-cod src-dom*$_{CRC}$)

lemma *trg-cod*$_{CRC}$ [*simp*]:
shows *V.trg* (*H.cod t*) = *H.cod t*
by (*metis cod.extensional cod.preserves-trg cod-trg*$_{CRC}$ *dom.extensional*
trg-dom$_{CRC}$)

sublocale *rts-category resid hcomp*
by *unfold-locales auto*

proposition *is-rts-category*:
shows *rts-category resid hcomp*
..

The elements of the originally given set *Obj* are in bijective correspondence with the objects of the constructed RTS-category.

lemma *bij-mkobj*:
shows *mkobj* ∈ *Obj* → *Collect obj*
and *Dom* ∈ *Collect obj* → *Obj*
and ⋀*A*. *Dom* (*mkobj A*) = *A*
and ⋀*a*. *obj a* ⟹ *mkobj* (*Dom a*) = *a*
and *bij-betw mkobj Obj* (*Collect obj*)
and *bij-betw Dom* (*Collect obj*) *Obj*
using *obj-mkobj obj-char arr-char*
apply *auto*[*6*]
by (*intro bij-betwI*, *auto*)+

abbreviation *MkArr*$_{ext}$
where *MkArr*$_{ext}$ *A B* ≡
λ*t*. *if residuation.arr* (*Hom A B*) *t then MkArr A B t else Null*

abbreviation *Trn*$_{ext}$
where *Trn*$_{ext}$ *a b* ≡

```
      λt. if residuation.arr (HOM a b) t then Trn t
          else ResiduatedTransitionSystem.partial-magma.null
               (Hom (Dom a) (Dom b))

lemma inverse-simulations-Trn-MkArr:
assumes A ∈ Obj and B ∈ Obj
shows inverse-simulations (Hom A B) (HOM (mkobj A) (mkobj B))
        (Trn_ext (mkobj A) (mkobj B)) (MkArr_ext A B)
proof −
  interpret Hom-AB: rts ‹Hom A B›
    using assms
    by (simp add: extensional-rts.axioms(1) rts-Hom)
  let ?a = mkobj A and ?b = mkobj B
  let ?F = MkArr_ext A B
  let ?G = Trn_ext (mkobj A) (mkobj B)
  have a: obj ?a and b: obj ?b
    using assms obj-char obj-mkobj by auto
  interpret HOM-ab: sub-rts resid ‹λt. «t : ?a → ?b»›
    using assms sub-rts-HOM by simp
  interpret F: simulation ‹Hom A B› ‹HOM ?a ?b› ?F
    using assms a b HOM-ab.con-char HOM-ab.null-char null-char
          Hom-AB.con-implies-arr(1−2) H.in-homI cod-char dom-char arr-char
          HOM-ab.resid-def con-char
    by unfold-locales auto
  interpret G: simulation ‹HOM ?a ?b› ‹Hom A B› ?G
  proof
    show ⋀t. ¬ HOM-ab.arr t ⟹ Trn_ext ?a ?b t = Hom-AB.null
      by simp
    fix t u
    assume tu: HOM-ab.con t u
    show con: Hom-AB.con (?G t) (?G u)
      using assms a b tu HOM-ab.arr-char HOM-ab.con-char con-char
      apply auto[1]
     by (metis Cod.simps(1) Cod-cod Cod-dom H.in-homE arr-char mkobj-def)
    show ?G (HOM ?a ?b t u) = Hom A B (?G t) (?G u)
   using assms tu con HOM-ab.con-char HOM-ab.resid-def HOM-ab.resid-closed
      apply auto[1]
   by (metis Cod-dom Dom-cod H.in-homE HOM-ab.inclusion mkobj-simps(1−2))
  qed
  show inverse-simulations (Hom A B) (HOM ?a ?b) ?G ?F
  proof
    show ?F ∘ ?G = I HOM-ab.resid
    proof
      fix t
      show (?F ∘ ?G) t = I HOM-ab.resid t
        using assms a b MkArr-Trn HOM-ab.arr-char HOM-ab.null-char
              arr-char null-char
        apply auto[1]
     apply (metis Cod-dom Dom-cod H.in-homE HOM-ab.inclusion mkobj-simps(1−2))
```

```
      by (metis Cod-dom Dom-cod H.in-homE H-arr-char mkobj-simps(1-2))
    qed
    show ?G ∘ ?F = I (Hom A B)
      using assms a b by auto
  qed
qed
```

Each hom-RTS of the constructed RTS-category is isomorphic to the corresponding RTS given by *Hom*.

```
  lemma isomorphic-rts-Hom-HOM:
  assumes A ∈ Obj and B ∈ Obj
  shows isomorphic-rts (Hom A B) (HOM (mkobj A) (mkobj B))
    using assms inverse-simulations-Trn-MkArr isomorphic-rts-def by blast

end

end
```

4.3 The RTS-Category of RTS's and Transformations

```
theory RTSCatx
imports Main ConcreteRTSCategory
begin
```

In this section we apply the *concrete-rts-category* construction to create an RTS-category, taking the set of all small extensional RTS's at a given arrow type as the objects and the exponential RTS's formed from these as the hom's, so that the arrows correspond to transformations and the arrows that are identities with respect to the residuation correspond to simulations. We prove that the resulting category, which we will refer to in informal text as $\mathbf{RTS}^{\dagger}$, is cartesian closed. For that to hold, we need to start with the assumption that the underlying arrow type is a universe.

```
locale rtscatx =
  universe arr-type
for arr-type :: 'A itself
begin

  sublocale concrete-rts-category
            ‹TYPE('A resid)› ‹TYPE(('A, 'A) exponential-rts.arr)›
            ‹Collect extensional-rts ∩ Collect small-rts›
            ‹λA B. exponential-rts.resid A B›
            ‹λA. exponential-rts.MkArr (I A) (I A) (I A)›
            ‹λA B C f g. COMP.map A B C (f, g)›
  proof (intro concrete-rts-category.intro)
    show ⋀A B. ⟦A ∈ Collect extensional-rts ∩ Collect small-rts;
                B ∈ Collect extensional-rts ∩ Collect small-rts⟧
```

```
            ⟹ extensional-rts (exponential-rts.resid A B)
  by (metis CollectD Int-Collect exponential-rts.intro
      exponential-rts.is-extensional-rts extensional-rts.extensional
      small-rts.axioms(1) weakly-extensional-rts.intro
      weakly-extensional-rts-axioms.intro)
show ⋀A. A ∈ Collect extensional-rts ∩ Collect small-rts ⟹
              residuation.ide (exponential-rts.resid A A)
                (exponential-rts.MkArr (I A) (I A) (I A))
proof −
  fix A :: 'a resid
  assume A: A ∈ Collect extensional-rts ∩ Collect small-rts
  show residuation.ide (exponential-rts.resid A A)
         (exponential-rts.MkIde (I A))
  proof −
    interpret A: extensional-rts A using A by blast
    interpret I: identity-simulation A ..
    interpret AA: exponential-rts A A ..
    show AA.ide (AA.MkIde (I A))
      using AA.ide-MkIde I.simulation-axioms by blast
  qed
qed
fix A :: 'a resid and B :: 'a resid
assume A: A ∈ Collect extensional-rts ∩ Collect small-rts
and B: B ∈ Collect extensional-rts ∩ Collect small-rts
interpret A: extensional-rts A using A by blast
interpret B: extensional-rts B using B by blast
interpret IA: identity-simulation A ..
interpret IA: simulation-as-transformation A A ‹I A› ..
interpret IB: identity-simulation B ..
interpret IB: simulation-as-transformation B B ‹I B› ..
interpret AA: exponential-rts A A ..
interpret BB: exponential-rts B B ..
interpret AB: exponential-rts A B ..
interpret Cmp-AAB: COMP A A B ..
interpret Cmp-ABB: COMP A B B ..
show ⋀t. AB.arr t ⟹ Cmp-AAB.map (t, AA.MkIde (I A)) = t
proof −
  fix t
  assume t: AB.arr t
  interpret t: transformation A B ‹AB.Dom t› ‹AB.Cod t› ‹AB.Map t›
    using t AB.arr-char by blast
  show Cmp-AAB.map (t, AA.MkIde (I A)) = t
  proof −
    have AB.Dom t ∘ AA.Dom (AA.MkIde (I A)) = AB.Dom t
      using t.F.simulation-axioms comp-simulation-identity by auto
    moreover have AB.Cod t ∘ AA.Cod (AA.MkIde (I A)) = AB.Cod t
      using t.G.simulation-axioms comp-simulation-identity by auto
    moreover have AB.Map t ∘ AA.Map (AA.MkIde (I A)) = AB.Map t
      using t.extensional by auto
```

```
    ultimately show ?thesis
      using t Cmp-AAB.map-eq AB.MkArr-Map AB.arr-char
          IA.transformation-axioms
      by auto
  qed
qed
show ⋀u. AB.arr u ⟹ Cmp-ABB.map (BB.MkIde (I B), u) = u
proof −
  fix u
  assume u: AB.arr u
  interpret u: transformation A B ‹AB.Dom u› ‹AB.Cod u› ‹AB.Map u›
    using u AB.arr-char by blast
  show Cmp-ABB.map (BB.MkIde (I B), u) = u
  proof −
    have BB.Dom (BB.MkIde (I B)) ∘ AB.Dom u = AB.Dom u
      using u.F.simulation-axioms comp-identity-simulation by auto
    moreover have BB.Cod (BB.MkIde (I B)) ∘ AB.Cod u = AB.Cod u
      using u.G.simulation-axioms comp-identity-simulation by auto
    moreover have BB.Map (BB.MkIde (I B)) ∘ AB.Map u = AB.Map u
    proof
      fix x
      show (BB.Map (BB.MkIde (I B)) ∘ AB.Map u) x = AB.Map u x
        using u.extensional u.preserves-arr by auto metis
    qed
    ultimately show ?thesis
      using u Cmp-ABB.map-eq AB.MkArr-Map AB.arr-char
          IB.transformation-axioms
      by auto
  qed
qed
fix C :: 'a resid
assume C: C ∈ Collect extensional-rts ∩ Collect small-rts
interpret C: extensional-rts C using C by blast
interpret BC: exponential-rts B C ..
interpret AC: exponential-rts A C ..
interpret BCxAB: product-rts BC.resid AB.resid ..
interpret Cmp-ABC: COMP A B C ..
show binary-simulation BC.resid AB.resid AC.resid
       (λ(t, u). Cmp-ABC.map (t, u))
  using Cmp-ABC.simulation-axioms BCxAB.product-rts-axioms
       BC.rts-axioms AB.rts-axioms AC.rts-axioms binary-simulation.intro
  by auto
fix D :: 'a resid
assume D: D ∈ Collect extensional-rts ∩ Collect small-rts
interpret D: extensional-rts D using D by blast
interpret BD: exponential-rts B D ..
interpret CD: exponential-rts C D ..
interpret Cmp-ABD: COMP A B D ..
interpret Cmp-BCD: COMP B C D ..
```

```
  interpret Cmp-ACD: COMP A C D ..
  show ⋀t u v. ⟦CD.arr t; BC.arr u; AB.arr v⟧ ⟹
                 COMP.map A B D (COMP.map B C D (t, u), v) =
                 COMP.map A C D (t, COMP.map A B C (u, v))
  proof −
    fix t u v
    assume t: CD.arr t and u: BC.arr u and v: AB.arr v
    have transformation A C
           (AC.Dom u ∘ AC.Dom v) (AC.Cod u ∘ AC.Cod v)
           (AC.Map u ∘ AC.Map v)
      using t u v Preliminaries.horizontal-composite
      by (metis A.extensional-rts-axioms AB.arrE B.extensional-rts-axioms
          BC.arrE C.extensional-rts-axioms)
    moreover
    have transformation B D
           (BD.Dom t ∘ BD.Dom u) (BD.Cod t ∘ BD.Cod u)
           (BD.Map t ∘ BD.Map u)
      using t u v Preliminaries.horizontal-composite
      by (metis B.extensional-rts-axioms BC.arrE C.extensional-rts-axioms
          CD.arrE D.extensional-rts-axioms)
    ultimately
    show COMP.map A B D (COMP.map B C D (t, u), v) =
         COMP.map A C D (t, COMP.map A B C (u, v))
      using t u v Cmp-ABD.map-eq Cmp-BCD.map-eq Cmp-ACD.map-eq
            Cmp-ABC.map-eq
      by auto
  qed
qed

type-synonym 'a arr =
  ('a resid, ('a, 'a) exponential-rts.arr) concrete-rts-category.arr

notation resid  (infix \ 70)
notation hcomp  (infixr ⋆ 53)
```

The mapping *Trn* that takes arrow $t \in H.hom\ a\ b$ to the underlying transition of the exponential RTS $[Dom\ a,\ Dom\ b]$, is injective.

```
lemma inj-Trn:
assumes obj a and obj b
shows Trn ∈ H.hom a b →
            Collect (residuation.arr (exponential-rts.resid (Dom a) (Dom b)))
and inj-on Trn (H.hom a b)
proof
  interpret A: extensional-rts ‹Dom a›
    using assms obj-char arr-char by blast
  interpret B: extensional-rts ‹Dom b›
    using assms obj-char arr-char by blast
  interpret AB: exponential-rts ‹Dom a› ‹Dom b› ..
  show ⋀x. x ∈ H.hom a b ⟹ Trn x ∈ Collect AB.arr
```

```
    using assms arr-char H.in-homE by auto
  show inj-on Trn (H.hom a b)
  proof
    fix t u
    assume t: t ∈ H.hom a b and u: u ∈ H.hom a b
    assume tu: Trn t = Trn u
    show t = u
      using t u tu AB.arr-eqI
      apply auto[1]
      by (metis H.comp-arr-dom H.comp-cod-arr H.in-homE H-seq-char
          MkArr-Trn)
  qed
qed

sublocale locally-small-rts-category resid hcomp
proof
  fix a b
  assume a: obj a and b: obj b
  interpret A: extensional-rts ‹Dom a›
    using a obj-char arr-char by blast
  interpret A: small-rts ‹Dom a›
    using a obj-char arr-char by blast
  interpret B: extensional-rts ‹Dom b›
    using b obj-char arr-char by blast
  interpret B: small-rts ‹Dom b›
    using b obj-char arr-char by blast
  interpret AB: exponential-of-small-rts ‹Dom a› ‹Dom b› ..
  have Trn ‘ H.hom a b ⊆ Collect AB.arr
    using H-arr-char image-subset-iff by auto
  moreover have inj-on Trn (H.hom a b)
    using a b inj-Trn by blast
  ultimately show small (H.hom a b)
    using a b AB.small smaller-than-small small-image-iff inj-Trn by metis
qed

abbreviation sta-in-hom   («- : - →_sta -»)
where sta-in-hom f a b ≡ H.in-hom f a b ∧ sta f

abbreviation trn-to   («- : - ⇒ -»)
where trn-to t f g ≡ arr t ∧ src t = f ∧ trg t = g

definition mkarr :: 'A resid ⇒ 'A resid ⇒
                    ('A ⇒ 'A) ⇒ ('A ⇒ 'A) ⇒ ('A ⇒ 'A) ⇒
                      'A arr
where mkarr A B F G τ ≡
      MkArr A B (exponential-rts.MkArr F G τ)

abbreviation mksta
where mksta A B F ≡ mkarr A B F F F
```

lemma *Dom-mkarr* [*simp*]:
shows *Dom* (*mkarr A B F G τ*) = *A*
unfolding *mkarr-def* **by** *simp*

lemma *Cod-mkarr* [*simp*]:
shows *Cod* (*mkarr A B F G τ*) = *B*
unfolding *mkarr-def* **by** *simp*

lemma *arr-mkarr* [*intro*]:
assumes *small-rts A* **and** *extensional-rts A*
and *small-rts B* **and** *extensional-rts B*
and *transformation A B F G τ*
shows *arr* (*mkarr A B F G τ*)
and *src* (*mkarr A B F G τ*) = *mksta A B F*
and *trg* (*mkarr A B F G τ*) = *mksta A B G*
and *dom* (*mkarr A B F G τ*) = *mkobj A*
and *cod* (*mkarr A B F G τ*) = *mkobj B*
proof −
interpret *A*: *extensional-rts A*
using *assms* **by** *simp*
interpret *B*: *extensional-rts B*
using *assms* **by** *simp*
interpret *AB*: *exponential-rts A B* **..**
interpret *τ*: *transformation A B F G τ*
using *assms*(*5*) **by** *blast*
show *1*: *arr* (*mkarr A B F G τ*)
unfolding *mkarr-def*
using *assms arr-char* **by** *auto*
show *src* (*mkarr A B F G τ*) = *mksta A B F*
and *trg* (*mkarr A B F G τ*) = *mksta A B G*
and *dom* (*mkarr A B F G τ*) = *mkobj A*
and *cod* (*mkarr A B F G τ*) = *mkobj B*
unfolding *mkarr-def*
using *assms 1 src-char trg-char AB.src-char AB.trg-char*
dom-char cod-char
by *auto*
qed

lemma *mkarr-simps* [*simp*]:
assumes *arr* (*mkarr A B F G σ*)
shows *dom* (*mkarr A B F G σ*) = *mkobj A*
and *cod* (*mkarr A B F G σ*) = *mkobj B*
and *src* (*mkarr A B F G σ*) = *mksta A B F*
and *trg* (*mkarr A B F G σ*) = *mksta A B G*
using *assms arr-mkarr dom-char cod-char* **apply** *auto*[*4*]
by (*metis* (*no-types*, *lifting*) *Cod-mkarr CollectD Dom-mkarr Int-Collect*
Trn.simps(*1*) *arrE exponential-rts.arr-MkArr exponential-rts.intro*
extensional-rts.axioms(*1*) *extensional-rts.extensional mkarr-def*

weakly-extensional-rts.intro weakly-extensional-rts-axioms.intro)+

lemma *sta-mksta* [*intro*]:
assumes *small-rts A* **and** *extensional-rts A*
and *small-rts B* **and** *extensional-rts B*
and *simulation A B F*
shows *sta* (*mksta A B F*)
and *dom* (*mksta A B F*) = *mkobj A* **and** *cod* (*mksta A B F*) = *mkobj B*
proof −
  interpret *A*: *extensional-rts A*
    using *assms* **by** *blast*
  interpret *B*: *extensional-rts B*
    using *assms* **by** *blast*
  interpret *F*: *simulation A B F*
    using *assms* **by** *blast*
  interpret *F*: *simulation-as-transformation A B F* **..**
  show *sta* (*mksta A B F*)
    using *assms F.transformation-axioms arr-mkarr V.ide-iff-src-self*
    by *presburger*
  show *dom* (*mksta A B F*) = *mkobj A*
    using *assms F.transformation-axioms arr-mkarr*(*4*) **by** *blast*
  show *cod* (*mksta A B F*) = *mkobj B*
    using *assms F.transformation-axioms arr-mkarr*(*5*) **by** *blast*
qed

abbreviation *Src*
where *Src* ≡ *exponential-rts.Dom* ∘ *Trn*

abbreviation *Trg*
where *Trg* ≡ *exponential-rts.Cod* ∘ *Trn*

abbreviation *Map*
where *Map* ≡ *exponential-rts.Map* ∘ *Trn*

lemma *Src-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F G* σ)
shows *Src* (*mkarr A B F G* σ) = *F*
  using *assms*
  by (*metis* (*mono-tags*, *lifting*) *Int-Collect Trn.simps*(*1*) *arrE comp-apply*
      exponential-rts.Dom.simps(*1*) *exponential-rts.intro*
      extensional-rts.axioms(*1*) *extensional-rts.extensional mem-Collect-eq*
      mkarr-def weakly-extensional-rts.intro
      weakly-extensional-rts-axioms.intro)

lemma *Trg-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F G* σ)
shows *Trg* (*mkarr A B F G* σ) = *G*
  using *assms*
  by (*metis* (*mono-tags*, *lifting*) *Int-Collect Trn.simps*(*1*) *arrE comp-apply*

exponential-rts.Cod.simps(1) exponential-rts.intro
extensional-rts.axioms(1) extensional-rts.extensional mem-Collect-eq
mkarr-def weakly-extensional-rts.intro
weakly-extensional-rts-axioms.intro)

lemma *Map-simps* [*simp*]:
assumes *arr t*
shows *Map* (*dom t*) = *I* (*Dom t*)
and *Map* (*cod t*) = *I* (*Cod t*)
and *Map* (*src t*) = *Src t*
and *Map* (*trg t*) = *Trg t*
proof −
  interpret *A*: *extensional-rts* ‹*Dom t*›
    using *assms arr-char* **by** *blast*
  interpret *B*: *extensional-rts* ‹*Cod t*›
    using *assms arr-char* **by** *blast*
  interpret *AB*: *exponential-rts* ‹*Dom t*› ‹*Cod t*› **..**
  show *Map* (*dom t*) = *I* (*Dom t*)
    using *assms dom-char* **by** *simp*
  show *Map* (*cod t*) = *I* (*Cod t*)
    using *assms cod-char* **by** *simp*
  show *Map* (*src t*) = *Src t*
    using *assms arr-char src-char* **by** *simp*
  show *Map* (*trg t*) = *Trg t*
    using *assms arr-char trg-char* **by** *simp*
qed

lemma *src-simp*:
assumes *arr t*
shows *src t* = *mksta* (*Dom t*) (*Cod t*) (*Src t*)
proof −
  interpret *A*: *extensional-rts* ‹*Dom t*›
    using *assms arr-char* **by** *blast*
  interpret *B*: *extensional-rts* ‹*Cod t*›
    using *assms arr-char* **by** *blast*
  interpret *AB*: *exponential-rts* ‹*Dom t*› ‹*Cod t*› **..**
  show *?thesis*
    using *assms mkarr-def src-char AB.src-char* **by** *auto*
qed

lemma *trg-simp*:
assumes *arr t*
shows *trg t* = *mksta* (*Dom t*) (*Cod t*) (*Trg t*)
proof −
  interpret *A*: *extensional-rts* ‹*Dom t*›
    using *assms arr-char* **by** *blast*
  interpret *B*: *extensional-rts* ‹*Cod t*›
    using *assms arr-char* **by** *blast*
  interpret *AB*: *exponential-rts* ‹*Dom t*› ‹*Cod t*› **..**

show *?thesis*
using *assms mkarr-def trg-char AB.trg-char* **by** *auto*
qed

The mapping *Map* that takes a transition to its underlying transformation, is a bijection, which cuts down to a bijection between states and simulations.

lemma *bij-mkarr*:
assumes *small-rts A* **and** *extensional-rts A*
and *small-rts B* **and** *extensional-rts B*
and *simulation A B F* **and** *simulation A B G*
shows *mkarr A B F G* ∈ *Collect* (*transformation A B F G*)
→ {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
and *Map* ∈ {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
→ *Collect* (*transformation A B F G*)
and [*simp*]: *Map* (*mkarr A B F G τ*) = *τ*
and [*simp*]: *t* ∈ {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
⟹ *mkarr A B F G* (*Map t*) = *t*
and *bij-betw* (*mkarr A B F G*) (*Collect* (*transformation A B F G*))
{*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
and *bij-betw Map* {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
(*Collect* (*transformation A B F G*))
proof −
interpret *A*: *extensional-rts A*
using *assms* **by** *simp*
interpret *B*: *extensional-rts B*
using *assms* **by** *simp*
interpret *AB*: *exponential-rts A B* **..**
show *1*: *mkarr A B F G* ∈
Collect (*transformation A B F G*)
→ {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
using *assms*(*1*,*3*) *src-char A.extensional-rts-axioms*
B.extensional-rts-axioms arr-mkarr(*1*−*3*)
by *auto*
show *2*: *Map* ∈ {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
→ *Collect* (*transformation A B F G*)
using *assms arr-char src-char trg-char mkarr-def*
apply *auto*[*1*]
by (*metis AB.Map.simps*(*1*) *AB.Map-src AB.Map-trg AB.arr-char*)
show *3*: ⋀*τ*. *Map* (*mkarr A B F G τ*) = *τ*
using *mkarr-def* **by** *simp*
show *4*: ⋀*t*. *t* ∈ {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
⟹ *mkarr A B F G* (*Map t*) = *t*
using *AB.MkArr-Map AB.arr-char MkArr-Trn arr-char src-char trg-char*
mkarr-def
apply *auto*[*1*]
by (*metis AB.Map.simps*(*1*) *AB.Map-src AB.Map-trg*)
show *bij-betw* (*mkarr A B F G*) (*Collect* (*transformation A B F G*))
{*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}

using *1 2 3 4* **by** (*intro bij-betwI*)
show *bij-betw Map* {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
(*Collect* (*transformation A B F G*))
using *1 2 3 4* **by** (*intro bij-betwI*)
qed

lemma *bij-mksta*:
assumes *small-rts A* **and** *extensional-rts A*
and *small-rts B* **and** *extensional-rts B*
shows *mksta A B* ∈ *Collect* (*simulation A B*)
→ {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
and *Map* ∈ {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
→ *Collect* (*simulation A B*)
and [*simp*]: *Map* (*mksta A B F*) = *F*
and [*simp*]: *t* ∈ {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
⟹ *mksta A B* (*Map t*) = *t*
and *bij-betw* (*mksta A B*) (*Collect* (*simulation A B*))
{*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
and *bij-betw Map* {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
(*Collect* (*simulation A B*))
proof −
interpret *A*: *extensional-rts A*
using *assms* **by** *simp*
interpret *A*: *small-rts A*
using *assms* **by** *simp*
interpret *B*: *extensional-rts B*
using *assms* **by** *simp*
interpret *B*: *small-rts B*
using *assms* **by** *simp*
interpret *AB*: *exponential-rts A B* **..**
show *1*: *mksta A B* ∈ *Collect* (*simulation A B*)
→ {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
proof
fix *F*
assume *F*: *F* ∈ *Collect* (*simulation A B*)
interpret *F*: *simulation A B F*
using *F* **by** *blast*
interpret *F*: *simulation-as-transformation A B F* **..**
show *mksta A B F* ∈ {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
using *assms F sta-mksta A.small-rts-axioms F.transformation-axioms*
by *auto*
qed
show *2*: *Map* ∈ {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
→ *Collect* (*simulation A B*)
using *AB.ide-char*$_{ERTS}$ *cod-char dom-char mkobj-def sta-char* **by** *auto*
show *3*: ⋀*F*. *Map* (*mksta A B F*) = *F*
using *mkarr-def* **by** *auto*
show *4*: ⋀*t*. *t* ∈ {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
⟹ *mksta A B* (*Map t*) = *t*

using *AB.Map.simps(1) Trn.simps(1) AB.MkArr-Map AB.arr-char mkarr-def*
apply *auto*[*1*]
by (*metis* (*no-types*, *lifting*) *AB.MkIde-Map Dom-cod Dom-dom H.in-homE MkArr-Trn V.ide-implies-arr mkobj-simps*(*1*) *sta-char*)
show *bij-betw* (*mksta A B*) (*Collect* (*simulation A B*)) {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
using *assms 1 2 3 4 sta-mksta*
apply (*intro bij-betwI*)
by (*auto simp add*: *dom-char cod-char*)
show *bij-betw Map* {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»} (*Collect* (*simulation A B*))
using *assms 1 2 3 4 sta-mksta*
apply (*intro bij-betwI*)
by (*auto simp add*: *dom-char cod-char*)
qed

lemma *mkarr-comp*:
assumes *small-rts A* **and** *extensional-rts A*
and *small-rts B* **and** *extensional-rts B*
and *small-rts C* **and** *extensional-rts C*
and *transformation A B F G σ*
and *transformation B C H K τ*
shows *mkarr A C* (*H* ∘ *F*) (*K* ∘ *G*) (*τ* ∘ *σ*) = *mkarr B C H K τ* ⋆ *mkarr A B F G σ*
proof −
interpret *COMP*: *COMP A B C*
using *assms COMP.intro* **by** *blast*
interpret *σ*: *transformation A B F G σ*
using *assms* **by** *simp*
interpret *τ*: *transformation B C H K τ*
using *assms* **by** *simp*
show *?thesis*
unfolding *hcomp-def mkarr-def*
using *assms sta-mksta COMP.map-eq* **by** *auto*
qed

lemma *mkarr-resid*:
assumes *small-rts A* ∧ *extensional-rts A*
and *small-rts B* ∧ *extensional-rts B*
and *consistent-transformations A B F G H σ τ*
shows *mkarr A B F G σ* ⌢ *mkarr A B F H τ*
and *mkarr A B H* (*consistent-transformations.apex A B H σ τ*) (*consistent-transformations.resid A B H σ τ*) = *mkarr A B F G σ* \ *mkarr A B F H τ*
proof −
interpret *A*: *extensional-rts A*
using *assms* **by** *simp*
interpret *B*: *extensional-rts B*
using *assms* **by** *simp*

```
  interpret AB: exponential-rts A B ..
  interpret στ: consistent-transformations A B F G H σ τ
    using assms by blast
  show 1: mkarr A B F G σ ⌢ mkarr A B F H τ
    using assms στ.con con-char AB.con-char mkarr-def
    by (simp add: στ.σ.transformation-axioms στ.τ.transformation-axioms)
  show mkarr A B H στ.apex στ.resid =
       mkarr A B F G σ \ mkarr A B F H τ
    unfolding mkarr-def
    using assms Trn-resid AB.resid-def AB.Apex-def AB.con-char
          στ.σ.transformation-axioms στ.τ.transformation-axioms
          στ.con
    by (intro arr-eqI) auto
qed

lemma Map-hcomp:
assumes H.seq t u
shows Map (t ⋆ u) = Map t ∘ Map u
proof −
  interpret COMP ‹Dom u› ‹Cod u› ‹Cod t›
    using assms arr-char COMP.intro H-seq-char by auto
  have t: arr t and u: arr u
    using assms by fastforce+
  have tu: Dom t = Cod u
    using assms H-seq-char by blast
  show ?thesis
    unfolding hcomp-def
    using assms tu t u map-eq H.ext arr-char by auto
qed

lemma Map-resid:
assumes V.con t u
shows consistent-transformations (Dom t) (Cod t)
        (Src t) (Trg t) (Trg u) (Map t) (Map u)
and Map (t \ u) =
      consistent-transformations.resid (Dom t) (Cod t) (Trg u)
        (Map t) (Map u)
proof −
  interpret A: extensional-rts ‹Dom t›
    using assms arr-char V.con-implies-arr by blast
  interpret B: extensional-rts ‹Cod t›
    using assms arr-char V.con-implies-arr by blast
  interpret AB: exponential-rts ‹Dom t› ‹Cod t› ..
  have 1: Dom t = Dom u and Cod t = Cod u
    using assms con-implies-Par(1−2) by auto
  have 2: Src t = Src u
    using assms con-char AB.con-char by simp
  interpret T: transformation ‹Dom t› ‹Cod t›
                ‹Src t› ‹Trg t› ‹Map t›
```

```
    using assms arr-char AB.arr-char [of Trn t] V.con-implies-arr
    by simp
  interpret U: transformation ‹Dom t› ‹Cod t›
               ‹Src t› ‹Trg u› ‹Map u›
    using assms 1 2 con-char arr-char [of u] AB.arr-char V.con-implies-arr
    by simp
  interpret TU: consistent-transformations ‹Dom t› ‹Cod t›
                ‹Src t› ‹Trg t› ‹Trg u›
                ‹Map t› ‹Map u›
    using assms con-char AB.con-char
    by unfold-locales auto
  show consistent-transformations (Dom t) (Cod t)
         (Src t) (Trg t) (Trg u)
         (Map t) (Map u)
    ..
  show Map (t \ u) =
       consistent-transformations.resid (Dom t) (Cod t) (Trg u)
         (Map t) (Map u)
    using assms con-char AB.con-char  AB.Map-resid by auto
qed

lemma simulation-Map-sta:
assumes sta f
shows simulation (Dom f) (Cod f) (Map f)
  by (metis Map-resid(1) Map-simps(3) V.ideE V.ide-implies-arr
      V.src-ide assms consistent-transformations.axioms(6)
      transformation.axioms(3))

lemma transformation-Map-arr:
assumes arr t
shows transformation (Dom t) (Cod t) (Src t) (Trg t) (Map t)
  by (meson Map-resid(1) V.arrE assms
      consistent-transformations.axioms(6))

lemma iso-char:
shows H.iso t ⟷ arr t ∧ Src t = Map t ∧ Trg t = Map t ∧
                 invertible-simulation (Dom t) (Cod t) (Map t)
proof
  assume t: H.iso t
  have 1: arr t
    using t H.iso-is-arr by simp
  interpret A: extensional-rts ‹Dom t›
    using 1 arr-char by blast
  interpret B: extensional-rts ‹Cod t›
    using 1 arr-char by blast
  interpret AB: exponential-rts ‹Dom t› ‹Cod t› ..
  interpret BA: exponential-rts ‹Cod t› ‹Dom t› ..
  show arr t ∧ Src t = Map t ∧ Trg t = Map t ∧
       invertible-simulation (Dom t) (Cod t) (Map t)
```

proof (*intro conjI*)
show *arr t* **by** *fact*
obtain *u* **where** *tu*: *H.inverse-arrows t u*
using *t H.iso-def* **by** *blast*
have *2*: *V.ide t ∧ V.ide u*
using *tu iso-implies-sta* **by** *auto*
have *3*: *Dom u = Cod t ∧ Cod u = Dom t*
using *tu*
by (*metis* (*no-types*, *lifting*) *H.inverse-arrowsE H-composable-char*
V.not-ide-null obj-is-sta)
show *Src t = Map t* **and** *Trg t = Map t*
using *2 sta-char AB.ide-char*$_{ERTS}$ **by** *auto*
let *?T = Map t* **and** *?U = Map u*
interpret *T*: *simulation* ‹*Dom t*› ‹*Cod t*› *?T*
using *2 sta-char AB.ide-char*$_{ERTS}$ **by** *simp*
interpret *U*: *simulation* ‹*Cod t*› ‹*Dom t*› *?U*
using *2 3 sta-char BA.ide-char*$_{ERTS}$ **by** *simp*
have *inverse-simulations* (*Dom t*) (*Cod t*) *?U ?T*
proof
show *?T ∘ ?U = I* (*Cod t*)
by (*metis* (*no-types*, *lifting*) *2 H.ide-compE H.inverse-arrowsE*
Map-hcomp Map-simps(*2*) *V.ide-implies-arr tu*)
show *?U ∘ ?T = I* (*Dom t*)
by (*metis* (*no-types*, *lifting*) *2 H.ide-compE H.inverse-arrowsE*
Map-hcomp Map-simps(*1*) *V.ide-implies-arr tu*)
qed
thus *invertible-simulation* (*Dom t*) (*Cod t*) (*Map t*)
using *invertible-simulation-def'* **by** *blast*
qed
next
assume *t*: *arr t ∧ Src t = Map t ∧ Trg t = Map t ∧*
invertible-simulation (*Dom t*) (*Cod t*) (*Map t*)
interpret *A*: *extensional-rts* ‹*Dom t*›
using *t arr-char* **by** *blast*
interpret *A*: *small-rts* ‹*Dom t*›
using *t arr-char* **by** *blast*
interpret *B*: *extensional-rts* ‹*Cod t*›
using *t arr-char* **by** *blast*
interpret *B*: *small-rts* ‹*Cod t*›
using *t arr-char* **by** *blast*
interpret *AB*: *exponential-rts* ‹*Dom t*› ‹*Cod t*› **..**
interpret *BA*: *exponential-rts* ‹*Cod t*› ‹*Dom t*› **..**
interpret *AA*: *exponential-rts* ‹*Dom t*› ‹*Dom t*› **..**
interpret *BB*: *exponential-rts* ‹*Cod t*› ‹*Cod t*› **..**
interpret *C*: *COMP* ‹*Dom t*› ‹*Cod t*› ‹*Dom t*› **..**
interpret *C'*: *COMP* ‹*Cod t*› ‹*Dom t*› ‹*Cod t*› **..**
interpret *T*: *invertible-simulation* ‹*Dom t*› ‹*Cod t*› ‹*Map t*›
using *t* **by** *auto*
show *H.iso t*

```
proof −
  obtain U where U: inverse-simulations (Dom t) (Cod t) U (Map t)
    using T.invertible by blast
  interpret U: simulation ‹Cod t› ‹Dom t› U
    using U inverse-simulations-def by blast
  interpret U: simulation-as-transformation ‹Cod t› ‹Dom t› U ..
  interpret TU: inverse-simulations ‹Dom t› ‹Cod t› U ‹Map t›
    using U by blast
  let ?u = mksta (Cod t) (Dom t) U
  have u: V.ide ?u ∧ «?u : cod t → dom t»
    using t sta-mksta U.simulation-axioms A.small-rts-axioms
          A.extensional-rts-axioms B.small-rts-axioms
          B.extensional-rts-axioms dom-char cod-char
    by auto
  have seq: H.seq ?u t ∧ H.seq t ?u
    using t u H.seqI by auto
  have H.inverse-arrows t ?u
  proof
    show obj (hcomp ?u t)
    proof −
      have hcomp ?u t = dom t
      proof (intro arr-eqI)
        show mksta (Cod t) (Dom t) U ⋆ t ≠ Null
          using t U.transformation-axioms sta-mksta V.not-arr-null
                null-char seq
          by force
        show dom t ≠ Null
          using t arr-char by blast
        show Dom (mksta (Cod t) (Dom t) U ⋆ t) = Dom (dom t)
          using t u sta-mksta mkarr-def by simp
        show Cod (mksta (Cod t) (Dom t) U ⋆ t) = Cod (dom t)
          using t u sta-mksta mkarr-def by simp
        show Trn (mksta (Cod t) (Dom t) U ⋆ t) = Trn (dom t)
        proof −
          have Trn (mksta (Cod t) (Dom t) U ⋆ t) =
                C.map (BA.MkIde U, Trn t)
            using t u Trn-hcomp mkarr-def by auto
          also have ... = C'.Currying.A-BC.MkArr
                            (U ∘ Src t) (U ∘ Trg t) (U ∘ Map t)
            unfolding C.map-eq
            using t U.transformation-axioms arr-char by auto
          also have ... = Trn (dom t)
            using t U inverse-simulations.inv' dom-char mkobj-simps(3)
            by auto
          finally show ?thesis by blast
        qed
      qed
      thus ?thesis
        using t H.ide-dom by auto
```

```
      qed
      show obj (hcomp t ?u)
      proof −
        have hcomp t ?u = cod t
        proof (intro arr-eqI)
          show t ⋆ mksta (Cod t) (Dom t) U ≠ Null
            using t U.transformation-axioms sta-mksta V.not-arr-null
                  null-char seq
            by force
          show cod t ≠ Null
            using t arr-char by blast
          show Dom (t ⋆ mksta (Cod t) (Dom t) U) = Dom (cod t)
            using t u sta-mksta mkarr-def by simp
          show Cod (t ⋆ mksta (Cod t) (Dom t) U) = Cod (cod t)
            using t u sta-mksta mkarr-def by simp
          show Trn (t ⋆ mksta (Cod t) (Dom t) U) = Trn (cod t)
          proof −
            have Trn (t ⋆ mksta (Cod t) (Dom t) U) =
                 C'.map (Trn t, BA.MkIde U)
              using t u Trn-hcomp mkarr-def by auto
            also have ... = C.Currying.A-BC.MkArr
                              (Src t ∘ U) (Trg t ∘ U) (Map t ∘ U)
              unfolding C'.map-eq
              using t U.transformation-axioms arr-char by auto
            also have ... = Trn (cod t)
              using t U inverse-simulations.inv cod-char mkobj-simps(3)
              by auto
            finally show ?thesis by blast
          qed
        qed
        thus ?thesis
          using t H.ide-cod by auto
      qed
    qed
    thus H.iso t by blast
  qed
qed

end
```

4.3.1 Terminal Object

The object corresponding to the one-arrow RTS is a terminal object. We don't want too much clutter in *rtscatx*, so we prove everything in a separate locale and then transfer only what we want to *rtscatx*.

```
locale terminal-object-in-rtscat =
  rtscatx arr-type
for arr-type :: 'A itself
begin
```

sublocale *One*: *one-arr-rts arr-type* **..**
interpretation I_1: *identity-simulation One.resid* **..**

abbreviation *one* (**1**)
where *one* ≡ *mkobj One.resid*

lemma *obj-one*:
shows *obj* **1**
using *obj-mkobj One.is-extensional-rts One.small-rts-axioms* **by** *blast*

definition *trm*
where *trm a* ≡ *MkArr* (*Dom a*) *One.resid*
(*exponential-rts.MkIde*
(*constant-simulation.map* (*Dom a*) *One.resid One.the-arr*))

lemma *one-universality*:
assumes *obj a*
shows «*trm a* : *a* → **1**»
and ⋀*t*. «*t* : *a* → **1**» ⟹ *t* = *trm a*
and ∃!*t*. «*t* : *a* → **1**»
proof −
interpret *A*: *extensional-rts* ‹*Dom a*›
using *assms obj-char arr-char* **by** *blast*
interpret *A*: *small-rts* ‹*Dom a*›
using *assms obj-char arr-char* **by** *blast*
interpret *A-One*: *exponential-rts* ‹*Dom a*› *One.resid* **..**
interpret *Trm*: *constant-simulation* ‹*Dom a*› *One.resid One.the-arr*
using $One.ide\text{-}char_{1RTS}$
by *unfold-locales auto*
interpret *Trm*: *simulation-as-transformation* ‹*Dom a*› *One.resid Trm.map* **..**
have *Dom-trm*: *Dom* (*trm a*) = *Dom a*
using *trm-def* **by** *simp*
have *Cod-trm*: *Cod* (*trm a*) = *One.resid*
using *trm-def* **by** *auto*
show *1*: «*trm a* : *a* → **1**»
proof −
have *a*: *mksta* (*Dom a*) (*Dom a*) (*I* (*Dom a*)) = *a*
using *assms bij-mkobj*(*4*) [*of a*] *mkobj-def mkarr-def* **by** *auto*
have *t*: *arr* (*trm a*)
using *assms obj-char arr-char One.is-extensional-rts One.small-rts-axioms*
A-One.ide-MkIde A-One.ide-implies-arr Trm.transformation-axioms
by (*unfold trm-def*, *intro arr-MkArr*) *auto*
show *?thesis*
using *a t dom-char cod-char Dom-trm Cod-trm mkobj-def mkarr-def*
by (*intro H.in-homI*) *auto*
qed
show ⋀*t*. «*t* : *a* → **1**» ⟹ *t* = *trm a*
proof (*intro arr-eqI*)

```
    fix t
    assume t: «t : a → 1»
    show t ≠ Null
      using t arr-char [of t] by auto
    show trm a ≠ Null
      using 1 arr-char [of trm a] by auto
    show Dom t = Dom (trm a)
      using t 1 trm-def dom-char by auto
    show Cod t = Cod (trm a)
      using t 1 cod-char mkobj-def
      by (metis (no-types, lifting) Cod.simps(1) H.in-homE)
    show Trn t = Trn (trm a)
    proof (intro A-One.arr-eqI)
      have 2: ⋀F G. ⟦simulation (Dom a) One.resid F;
                    simulation (Dom a) One.resid G⟧
                      ⟹ F = G
        using A.rts-axioms One.universality by blast
      show 3: A-One.arr (Trn t)
        using assms t arr-char mkobj-def
        by (metis (no-types, lifting) H.ideD(1-2) H.in-homE
            H-arr-char cod-char dom-char arr.simps(1))
      show 4: A-One.arr (Trn (trm a))
        using 1 trm-def H.in-homE H-arr-char by auto
      show A-One.Dom (Trn t) = A-One.Dom (Trn (trm a))
        using 2 3 4 trm-def A-One.ide-MkIde A-One.ide-src A-One.src-simp
        by metis
      show A-One.Cod (Trn t) = A-One.Cod (Trn (trm a))
        using 2 3 4 trm-def A-One.arr-char transformation.axioms(4)
        by metis
      show ⋀x. A.ide x ⟹
                 A-One.Map (Trn t) x = A-One.Map (Trn (trm a)) x
        using 3 trm-def A-One.con-char One.arr-char One.con-char by force
    qed
  qed
  thus ∃!t. «t : a → 1»
    using 1 by blast
qed

lemma terminal-one:
shows H.terminal 1
  using one-universality H.terminal-def obj-one by blast

lemma trm-in-hom [intro, simp]:
assumes obj a
shows «trm a : a → 1»
  using assms one-universality by simp

lemma terminal-arrow-is-sta:
assumes «t : a → 1»
```

shows *sta t*
proof −
have *src t* = *t*
using *assms H.ide-dom one-universality*(*3*)
by (*metis* (*no-types*, *lifting*) *H.in-homE H.terminal-arr-unique*
cod-src dom-src src.preserves-arr terminal-one)
thus *?thesis*
using *assms*
by (*metis* (*no-types*, *lifting*) *H.arrI V.ide-iff-src-self arr-coincidence*)
qed

For any object a we have an RTS isomorphism $Dom\ a \cong HOM\ \mathbf{1}\ a$. Note that these are *not* at the same type.

abbreviation *UP* :: *'A arr* ⇒ *'A* ⇒ *'A arr*
where *UP a* ≡ $MkArr_{ext}$ (\$_1$) (*Dom a*) ∘ *exponential-by-One.Up* (*Dom a*)

abbreviation *DN* :: *'A arr* ⇒ *'A arr* ⇒ *'A*
where *DN a* ≡ *exponential-by-One.Dn* (*Dom a*) ∘ Trn_{ext} **1** *a*

lemma *inverse-simulations-DN-UP*:
assumes *obj a*
shows *inverse-simulations* (*Dom a*) (*HOM* **1** *a*) (*DN a*) (*UP a*)
and *isomorphic-rts* (*Dom a*) (*HOM* **1** *a*)
proof −
interpret *A*: *extensional-rts* ‹*Dom a*›
using *assms obj-char arr-char* **by** *blast*
interpret *A*: *small-rts* ‹*Dom a*›
using *assms obj-char arr-char* **by** *blast*
interpret *Exp*: *exponential-rts One.resid* ‹*Dom a*› **..**
interpret *HOM*: *sub-rts resid* ‹λ*t*. «*t* : **1** → *a*»›
using *assms sub-rts-HOM* **by** *blast*
interpret *exponential-by-One arr-type* ‹*Dom a*› **..**
interpret *Dom-Exp*: *inverse-simulations* ‹*Dom a*› *Exp.resid Dn Up*
using *inverse-simulations-Dn-Up* **by** *blast*
interpret *Trn-MkArr*: *inverse-simulations Exp.resid HOM.resid*
‹Trn_{ext} **1** *a*› ‹$MkArr_{ext}$ *One.resid* (*Dom a*)›
using *assms inverse-simulations-Trn-MkArr* [*of One.resid Dom a*]
bij-mkobj(*4*) [*of a*] *A.extensional-rts-axioms A.small-rts-axioms*
One.extensional-rts-axioms One.small-rts-axioms mkobj-def
apply *auto*[*1*]
by *metis*
show *inverse-simulations* (*Dom a*) *HOM.resid*
(*Dn* ∘ Trn_{ext} **1** *a*) ($MkArr_{ext}$ *One.resid* (*Dom a*) ∘ *Up*)
using *inverse-simulations-compose Dom-Exp.inverse-simulations-axioms*
Trn-MkArr.inverse-simulations-axioms
by *blast*
thus *isomorphic-rts* (*Dom a*) (*HOM* **1** *a*)
using *isomorphic-rts-def* **by** *blast*
qed

```
lemma terminal-char:
shows H.terminal x ⟷ obj x ∧ (∃!t. residuation.arr (Dom x) t)
proof (intro iffI conjI)

  assume x: H.terminal x
  show obj-x: obj x
    using x H.terminal-def by fastforce
  interpret X: extensional-rts ‹Dom x›
    using obj-x obj-char arr-char by blast
  have 1: H.isomorphic x 𝟭
    using x obj-char terminal-one H.terminal-objs-isomorphic by force
  obtain f where f: «f : x → 𝟭» ∧ H.iso f
    using 1 H.isomorphic-def by auto
  have ide-f: sta f
    using f iso-implies-sta by blast
  show ∃!t. residuation.arr (Dom x) t
  proof −
    have card (Collect (residuation.arr (Dom x))) = 1
    proof −
      have bij-betw (Map f) (Collect X.arr) (Collect One.arr)
      proof −
        have Dom f = Dom x and Cod f = One.resid
          using f dom-char cod-char mkobj-def by auto
        thus ?thesis
          by (metis (no-types, lifting) f
              invertible-simulation.is-bijection-betw-arr-sets iso-char)
      qed
      moreover have card (Collect One.arr) = 1
        by (simp add: Collect-cong One.arr-char)
      ultimately show ?thesis
        by (simp add: bij-betw-same-card)
    qed
    thus ?thesis
      by (metis CollectI Collect-empty-eq One-nat-def card-1-singleton-iff
          card-eq-0-iff singleton-iff zero-neq-one)
  qed
  next
  assume x: obj x ∧ (∃!t. residuation.arr (Dom x) t)
  interpret X: extensional-rts ‹Dom x›
    using x obj-char arr-char by blast
  interpret T: simulation ‹Dom x› One.resid ‹One.terminator (Dom x)›
    using x One.terminator-is-simulation obj-char arr-char small-rts-def
    by blast
  have bij-betw (One.terminator (Dom x)) (Collect X.arr) (Collect One.arr)
  proof (unfold bij-betw-def, intro conjI)
    show inj-on (One.terminator (Dom x)) (Collect X.arr)
      using x T.simulation-axioms
      by (intro inj-onI) auto
```

```
    show One.terminator (Dom x) ‘ Collect X.arr = Collect One.arr
    proof
      show One.terminator (Dom x) ‘ Collect X.arr ⊆ Collect One.arr
        by auto
      show Collect One.arr ⊆ One.terminator (Dom x) ‘ Collect X.arr
        using x T.simulation-axioms One.arr-char T.preserves-reflects-arr
        by (metis (no-types, lifting) CollectD CollectI image-iff subsetI)
    qed
  qed
  hence 2: invertible-simulation (Dom x) One.resid (One.terminator (Dom x))
    using invertible-simulation-iff
          [of Dom x One.resid One.terminator (Dom x)]
        One.con-implies-arr
    by (metis T.simulation-axioms X.arrE T.preserves-reflects-arr x)
  have 3: sta (mksta (Dom x) One.resid (One.terminator (Dom x)))
    using x T.simulation-axioms obj-char iso-char sta-mksta(1)
        arr-char One.small-rts-axioms One.extensional-rts-axioms
        invertible-simulation-def
    by blast
  have 4: H.iso (mksta (Dom x) One.resid (One.terminator (Dom x)))
    unfolding iso-char
    using 2 3 bij-mksta(3) sta-char mkarr-def
    by (metis Cod-mkarr Dom-mkarr Map-simps(4) Src-mkarr Trg-mkarr
        V.ide-implies-arr V.trg-ide)
  interpret T: simulation-as-transformation
                ‹Dom x› One.resid ‹One.terminator (Dom x)›
    ..
  have H.isomorphic x 1
    using x 4 obj-char arr-char mkarr-simps(1−2) One.small-rts-axioms
        One.extensional-rts-axioms T.transformation-axioms
        H.isomorphicI [of mksta (Dom x) (\1) (One.terminator (Dom x))]
    by (simp add: arr-mkarr(4−5))
  thus H.terminal x
    using H.isomorphic-symmetric H.isomorphic-to-terminal-is-terminal
        terminal-one
    by blast
qed

end
```

The above was all carried out in a separate locale. Here we transfer to *rtscatx* just the final definitions and facts that we want.

```
context rtscatx
begin

  sublocale One: one-arr-rts arr-type ..

  definition one  (1)
  where one ≡ terminal-object-in-rtscat.one
```

definition *trm*
where *trm* = *terminal-object-in-rtscat.trm*

interpretation *Trm*: *terminal-object-in-rtscat* **..**
no-notation *Trm.one* (**1**)

lemma *obj-one* [*intro*, *simp*]:
shows *obj one*
  unfolding *one-def*
  using *Trm.obj-one* **by** *blast*

lemma *trm-simps'* [*simp*]:
assumes *obj a*
shows *arr* (*trm a*) **and** *dom* (*trm a*) = *a* **and** *cod* (*trm a*) = **1**
and *src* (*trm a*) = *trm a* **and** *trg* (*trm a*) = *trm a*
and *sta* (*trm a*)
proof −
  show *arr* (*trm a*) **and** *dom* (*trm a*) = *a* **and** *cod* (*trm a*) = **1**
    unfolding *trm-def one-def*
    using *assms Trm.terminal-arrow-is-sta H.in-homE*
      by *auto blast*+
  show *src* (*trm a*) = *trm a* **and** *trg* (*trm a*) = *trm a* **and** *sta* (*trm a*)
    using *Trm.terminal-arrow-is-sta Trm.trm-in-hom V.src-ide*
        V.trg-ide assms trm-def
    by (*metis* (*no-types*, *lifting*))+
qed

sublocale *category-with-terminal-object hcomp*
  using *Trm.terminal-one H.terminal-def Trm.obj-one*
  by *unfold-locales auto*

sublocale *elementary-category-with-terminal-object hcomp one trm*
  using *Trm.obj-one Trm.trm-in-hom*
  by *unfold-locales*
    (*auto simp add*: *Trm.one-universality*(*2*) *one-def trm-def*)

lemma *is-elementary-category-with-terminal-object*:
shows *elementary-category-with-terminal-object hcomp one trm*
  ..

lemma *terminal-char*:
shows *H.terminal x* ⟷ *obj x* ∧ (∃!*t*. *residuation.arr* (*Dom x*) *t*)
  using *Trm.terminal-char* **by** *simp*

lemma *Map-trm*:
assumes *obj a*
shows *Map* (*trm a*) =
      constant-simulation.map (*Dom a*) *One.resid One.the-arr*

proof −
interpret *A*: *extensional-rts* ‹*Dom a*›
using *assms obj-char arr-char* **by** *blast*
interpret *A1*: *exponential-rts* ‹*Dom a*› *One.resid* **..**
show *?thesis*
using *assms trm-def Trm.trm-def*
by (*metis A1.Map.simps*(*1*) *Trn.simps*(*1*) *comp-apply*)
qed

lemma *inverse-simulations-DN-UP*:
assumes *obj a*
shows *inverse-simulations* (*Dom a*) (*HOM* **1** *a*) (*Trm.DN a*) (*Trm.UP a*)
and *isomorphic-rts* (*Dom a*) (*HOM* **1** *a*)
unfolding *one-def*
using *assms Trm.inverse-simulations-DN-UP* **by** *auto*

abbreviation UP_{rts} :: *'A arr* ⇒ *'A* ⇒ *'A arr*
where UP_{rts} *a* ≡ $MkArr_{ext}$ (\$\backslash_1$) (*Dom a*) ∘ *exponential-by-One.Up* (*Dom a*)

abbreviation DN_{rts} :: *'A arr* ⇒ *'A arr* ⇒ *'A*
where DN_{rts} *a* ≡ *exponential-by-One.Dn* (*Dom a*) ∘ Trn_{ext} **1** *a*

lemma *UP-DN-naturality*:
assumes *arr t*
shows DN_{rts} (*cod t*) ∘ *cov-HOM* **1** *t* = *Map t* ∘ DN_{rts} (*dom t*)
and UP_{rts} (*cod t*) ∘ *Map t* = *cov-HOM* **1** *t* ∘ UP_{rts} (*dom t*)
and *cov-HOM* **1** *t* = UP_{rts} (*cod t*) ∘ *Map t* ∘ DN_{rts} (*dom t*)
and *Map t* = DN_{rts} (*cod t*) ∘ *cov-HOM* **1** *t* ∘ UP_{rts} (*dom t*)
proof −
let *?a* = *dom t* **and** *?b* = *cod t*
let *?A* = *Dom t* **and** *?B* = *Cod t*
have *a*: *obj ?a* **and** *b*: *obj ?b*
using *assms* **by** *auto*
have *t*: «*t* : *?a* → *?b*»
using *assms* **by** *auto*
have *a-simp*: *mksta ?A ?A* (*I ?A*) = *?a*
using *assms a bij-mkobj*(*4*) *dom-char mkobj-def mkarr-def* **by** *simp*
have *b-simp*: *mksta ?B ?B* (*I ?B*) = *?b*
using *assms b bij-mkobj*(*4*) *cod-char mkobj-def mkarr-def* **by** *simp*
have *one-simp*: *mksta One.resid One.resid* (*I One.resid*) = *one*
unfolding *one-def mkarr-def*
by (*simp add*: *mkobj-def*)
interpret *A*: *extensional-rts ?A*
using *assms* **by** *blast*
interpret *A*: *small-rts ?A*
using *assms* **by** *blast*
interpret *B*: *extensional-rts ?B*
using *assms* **by** *blast*
interpret *B*: *small-rts ?B*

using *assms* **by** *blast*
interpret *OneA*: *exponential-by-One arr-type ?A* **..**
interpret *OneB*: *exponential-by-One arr-type ?B* **..**
interpret *HOM-1a*: *sub-rts resid* ‹$\lambda t.$ «t: **1** $\rightarrow$ *?a*»›
using *a sub-rts-HOM* **by** *blast*
interpret *HOM-1a*: *sub-rts-of-extensional-rts resid* ‹$\lambda t.$ «t: **1** $\rightarrow$ *?a*»› **..**
interpret *HOM-1b*: *sub-rts resid* ‹$\lambda t.$ «t: **1** $\rightarrow$ *?b*»›
using *b sub-rts-HOM* **by** *blast*
interpret *HOM-1b*: *sub-rts-of-extensional-rts resid* ‹$\lambda t.$ «t: **1** $\rightarrow$ *?b*»› **..**
interpret *Trn-MkArr-a*: *inverse-simulations OneA.resid* ‹*HOM* **1** *?a*›
‹Trn_{ext} **1** *?a*› ‹$MkArr_{ext}$ *One.resid ?A*›
proof −
show *inverse-simulations OneA.resid* (*HOM* **1** *?a*)
(*Trn*$_{ext}$ **1** *?a*) (*MkArr*$_{ext}$ *One.resid ?A*)
using *assms inverse-simulations-Trn-MkArr*(*1*) [*of One.resid ?A*]
unfolding *one-def mkobj-def*
apply *simp*
by (*metis A.extensional-rts-axioms A.small-rts-axioms*
One.is-extensional-rts One.small-rts-axioms a-simp mkarr-def)
qed
interpret *Trn-MkArr-b*: *inverse-simulations OneB.resid* ‹*HOM* **1** *?b*›
‹Trn_{ext} **1** *?b*› ‹$MkArr_{ext}$ *One.resid ?B*›
proof −
show *inverse-simulations OneB.resid* (*HOM* **1** *?b*)
(*Trn*$_{ext}$ **1** *?b*) (*MkArr*$_{ext}$ *One.resid ?B*)
using *assms inverse-simulations-Trn-MkArr*(*1*) [*of One.resid ?B*]
unfolding *one-def mkobj-def*
apply *simp*
by (*metis B.extensional-rts-axioms B.small-rts-axioms*
One.is-extensional-rts One.small-rts-axioms b-simp mkarr-def)
qed
have *UP-a*: UP_{rts} *?a* = $MkArr_{ext}$ ($\backslash_1$) *?A* ∘ *OneA.Up*
using *assms t Dom-dom* **by** *presburger*
have *UP-b*: UP_{rts} *?b* = $MkArr_{ext}$ ($\backslash_1$) *?B* ∘ *OneB.Up*
using *assms t Dom-cod* **by** *presburger*
have *DN-a*: DN_{rts} *?a* = *OneA.Dn* ∘ Trn_{ext} **1** *?a*
using *assms t Dom-dom* **by** *presburger*
have *DN-b*: DN_{rts} *?b* = *OneB.Dn* ∘ Trn_{ext} **1** *?b*
using *assms t Dom-cod* **by** *presburger*
interpret *UP-DN-a*: *inverse-simulations ?A HOM-1a.resid*
‹DN_{rts} *?a*› ‹UP_{rts} *?a*›
using *a t DN-a UP-a OneA.inverse-simulations-Dn-Up*
Trn-MkArr-a.inverse-simulations-axioms
inverse-simulations-compose
by *fastforce*
interpret *UP-DN-b*: *inverse-simulations ?B HOM-1b.resid*
‹DN_{rts} *?b*› ‹UP_{rts} *?b*›
using *b t DN-b UP-b OneB.inverse-simulations-Dn-Up*
Trn-MkArr-b.inverse-simulations-axioms

```
      inverse-simulations-compose
  by fastforce
interpret T: transformation ?A ?B ‹Src t› ‹Trg t› ‹Map t›
  using assms(1) arr-char [of t]
  by (simp add: A.rts-axioms A.weak-extensionality B.extensional-rts-axioms
      exponential-rts.arr-char exponential-rts.intro
      weakly-extensional-rts.intro weakly-extensional-rts-axioms.intro)
interpret T': transformation ‹HOM 𝟭 ?a› ‹HOM 𝟭 ?b›
               ‹cov-HOM 𝟭 (src t)› ‹cov-HOM 𝟭 (trg t)› ‹cov-HOM 𝟭 t›
  using assms(1) transformation-cov-HOM-arr [of 𝟭 t] obj-one by blast

interpret LHS: transformation ‹HOM 𝟭 ?a› ?B
                ‹DN_rts ?b ∘ cov-HOM 𝟭 (src t)›
                ‹DN_rts ?b ∘ cov-HOM 𝟭 (trg t)›
                ‹DN_rts ?b ∘ cov-HOM 𝟭 t›
  using assms transformation-whisker-left UP-DN-b.F.simulation-axioms
    T'.F.simulation-axioms T'.G.simulation-axioms T'.transformation-axioms
      B.weakly-extensional-rts-axioms DN-b
  by fastforce
interpret RHS: transformation ‹HOM 𝟭 ?a› ?B
                ‹Src t ∘ DN_rts ?a› ‹Trg t ∘ DN_rts ?a› ‹Map t ∘ DN_rts ?a›
  using assms
        transformation-whisker-right
          [of ?A ?B Src t Trg t Map t HOM-1a.resid DN_rts ?a]
        UP-DN-a.F.simulation-axioms T.transformation-axioms
        HOM-1a.weakly-extensional-rts-axioms DN-a
  by auto
show 1: DN_rts ?b ∘ cov-HOM 𝟭 t = Map t ∘ DN_rts ?a
proof
  fix x
  show (DN_rts ?b ∘ cov-HOM 𝟭 t) x = (Map t ∘ DN_rts ?a) x
  proof (cases HOM-1a.arr x)
    show ¬ HOM-1a.arr x ⟹ ?thesis
      using LHS.extensional RHS.extensional by auto
    assume x: HOM-1a.arr x
    have Trn-x: OneA.arr (Trn x)
      using Trn-MkArr-a.F.preserves-reflects-arr x by presburger
    have Trn-tx: OneB.arr (Trn (t ⋆ x))
      using x t T'.preserves-arr Trn-MkArr-b.F.preserves-reflects-arr
      by presburger
    show ?thesis
      using assms x Map-hcomp Trn-tx Dom-cod T'.preserves-arr
            HOM-1b.arr-char HOM-1b.inclusion T'.preserves-arr Trn-x
      by (auto simp add: one-def)
  qed
qed
show cov-HOM 𝟭 t = UP_rts ?b ∘ Map t ∘ DN_rts ?a
proof −
  have cov-HOM 𝟭 t = (UP_rts ?b ∘ DN_rts ?b) ∘ cov-HOM 𝟭 t
```

```
    using b t comp-identity-transformation [of HOM-1a.resid HOM-1b.resid]
          T'.transformation-axioms UP-DN-b.inv UP-b DN-b
    by force
  also have ... = UP_rts ?b ∘ (DN_rts ?b ∘ cov-HOM 1 t)
    by auto
  also have ... = UP_rts ?b ∘ (Map t ∘ DN_rts ?a)
    using 1 by simp
  also have ... = UP_rts ?b ∘ Map t ∘ DN_rts ?a
    by auto
  finally show ?thesis by blast
qed
show 2: UP_rts ?b ∘ Map t = cov-HOM 1 t ∘ UP_rts ?a

proof −
  have UP_rts ?b ∘ Map t = UP_rts ?b ∘ (Map t ∘ (DN_rts ?a ∘ UP_rts ?a))
    using a t T.transformation-axioms UP-a DN-a
          UP-DN-a.inverse-simulations-axioms
    by (simp add: comp-transformation-identity inverse-simulations.inv')
  also have ... = UP_rts ?b ∘ (DN_rts ?b ∘ cov-HOM 1 t ∘ UP_rts ?a)
    using 1 by auto
  also have ... = UP_rts ?b ∘ (DN_rts ?b ∘ (cov-HOM 1 t ∘ UP_rts ?a))
    using Fun.comp-assoc [of DN_rts ?b cov-HOM 1 t UP_rts ?a] by force
  also have ... = (UP_rts ?b ∘ DN_rts ?b) ∘ (cov-HOM 1 t ∘ UP_rts ?a)
    using Fun.comp-assoc [of UP_rts ?b DN_rts ?b cov-HOM 1 t ∘ UP_rts ?a]
    by force
  also have ... = I HOM-1b.resid ∘ (cov-HOM 1 t ∘ UP_rts ?a)
    using UP-DN-b.inv by force
  also have ... = I HOM-1b.resid ∘ cov-HOM 1 t ∘ UP_rts ?a
    using Fun.comp-assoc [of I HOM-1b.resid cov-HOM 1 t UP_rts ?a]
    by force
  also have ... = cov-HOM 1 t ∘ UP_rts ?a
    using comp-identity-transformation
          [of HOM-1a.resid HOM-1b.resid cov-HOM 1 (src t)
              cov-HOM 1 (trg t) cov-HOM 1 t]
          T'.transformation-axioms
    by fastforce
  finally show ?thesis by blast
qed
show Map t = DN_rts ?b ∘ cov-HOM 1 t ∘ UP_rts ?a
proof −
  have Map t = DN_rts ?b ∘ UP_rts ?b ∘ Map t
  proof −
    have Map t = I (Cod t) ∘ Map t
      using T.transformation-axioms
            comp-identity-transformation [of Dom t Cod t]
      by auto
    also have ... = DN_rts ?b ∘ UP_rts ?b ∘ Map t
      using b UP-b DN-b UP-DN-b.inverse-simulations-axioms
            inverse-simulations.inv'
```

```
      by (metis (no-types, lifting))
    finally show ?thesis by blast
  qed
  also have ... = DN_rts ?b ∘ (UP_rts ?b ∘ Map t)
    by auto
  also have ... = DN_rts ?b ∘ (cov-HOM 1 t ∘ UP_rts ?a)
    using 2 by simp
  also have ... = DN_rts ?b ∘ cov-HOM 1 t ∘ UP_rts ?a
    using comp-assoc [of DN_rts ?b cov-HOM 1 t UP_rts ?a] by metis
  finally show ?thesis by blast
 qed
qed
```

Equality of parallel arrows «$u : a \to b$» and «$v : a \to b$» is determined by their compositions with global transitions «$t : \mathbf{1} \to a$».

```
lemma arr-extensionality:
assumes «u : a → b» and «v : a → b» and src u = src v and trg u = trg v
shows u = v ⟷ (∀ t. «t : 1 → a» ⟶ u ⋆ t = v ⋆ t)
proof
  have a: obj a and b: obj b
    using assms(1) by auto
  have A: small-rts (Dom a) ∧ extensional-rts (Dom a)
  and B: small-rts (Dom b) ∧ extensional-rts (Dom b)
    using a b obj-char arr-char by blast+
  interpret A: extensional-rts ‹Dom a›
    using A by blast
  interpret B: extensional-rts ‹Dom b›
    using B by blast
  interpret AB: exponential-rts ‹Dom a› ‹Dom b› ..
  have Dom u = Dom a and Cod u = Dom b
    using assms(1) Dom-dom Dom-cod by auto
  have Dom v = Dom a and Cod v = Dom b
    using assms(2) Dom-dom Dom-cod by auto
  have Map (src u) = Src u and Map (trg u) = Trg u
    using assms(1) Map-simps(3-4) by fastforce+
  have Map (src v) = Src v and Map (trg v) = Trg v
    using assms(2) Map-simps(3-4) by fastforce+
  interpret U: transformation ‹Dom a› ‹Dom b›
                ‹Map (src u)› ‹Map (trg u)› ‹Map u›
    using assms(1) arr-char [of u] AB.arr-char [of Trn u]
          ‹Dom u = Dom a› ‹Cod u = Dom b›
          ‹Map (src u) = Src u› ‹Map (trg u) = Trg u›
    by auto
  interpret V: transformation ‹Dom a› ‹Dom b›
                ‹Map (src v)› ‹Map (trg v)› ‹Map v›
    using assms(2) arr-char [of v] AB.arr-char [of Trn v]
          ‹Dom v = Dom a› ‹Cod v = Dom b›
          ‹Map (src v) = Src v› ‹Map (trg v) = Trg v›
    by auto
```

```
show u = v ⟹ ∀ t. «t : 1 → a» ⟶ u ⋆ t = v ⋆ t
  by blast
show ∀ t. «t : one → a» ⟶ u ⋆ t = v ⋆ t ⟹ u = v
proof (intro arr-eqI)
  assume 1: ∀ t. «t : 1 → a» ⟶ u ⋆ t = v ⋆ t
  show u ≠ Null
    using assms(1) arr-char by fastforce
  show v ≠ Null
    using assms(2) arr-char by fastforce
  show Dom u = Dom v
    using ‹Dom u = Dom a› ‹Dom v = Dom a› by auto
  show Cod u = Cod v
    using ‹Cod u = Dom b› ‹Cod v = Dom b› by presburger
  show Trn u = Trn v
  proof (intro AB.arr-eqI)
    show AB.arr (Trn u)
      using assms(1) arr-char [of u] ‹Dom u = Dom a› ‹Cod u = Dom b›
      by auto
    show AB.arr (Trn v)
      using assms(2) arr-char [of v] ‹Dom v = Dom a› ‹Cod v = Dom b›
      by auto
    show AB.Dom (Trn u) = AB.Dom (Trn v)
      using assms(3) arr-char arr-char ‹Map (src u) = Src u›
            ‹Map (src v) = Src v›
      by auto
    show AB.Cod (Trn u) = AB.Cod (Trn v)
      using assms(4) arr-char arr-char ‹Map (trg u) = Trg u›
            ‹Map (trg v) = Trg v›
      by auto
    have Map u = Map v
    proof −
      have ⋀Q R T. transformation One.resid (Dom a) Q R T
                     ⟹ Map u ∘ T = Map v ∘ T
      proof −
        fix Q R T
        assume 2: transformation One.resid (Dom a) Q R T
        interpret T: transformation One.resid ‹Dom a› Q R T
          using 2 by blast
        let ?t = mkarr One.resid (Dom a) Q R T
        have t: «?t : 1 → a»
          by (metis (no-types, lifting) 2 A H.ideD(2) H.ideD(3) H.ide-in-hom
            H.in-homI One.is-extensional-rts One.small-rts-axioms Trm.obj-one
            Trm.one-universality(2) a arr-coincidence arr-mkarr(1) arr-mkarr(5)
              mkarr-simps(1) mkobj-Dom trm-def trm-simps(3))
        show Map u ∘ T = Map v ∘ T
          by (metis (no-types, lifting) 1 AB.Map.simps(1) H.seqI' mkarr-def
              Map-hcomp Trn.simps(1) assms(1) comp-def t)
      qed
      thus Map u = Map v
```

```
        using assms(3-4)
        One.eq-transformation-iff U.transformation-axioms V.transformation-axioms
            A.weakly-extensional-rts-axioms B.weakly-extensional-rts-axioms
            ‹Map (src u) = Src u› ‹Map (trg u) = Trg u›
            ‹Map (src v) = Src v› ‹Map (trg v) = Trg v›
        by metis
    qed
    thus ⋀a. A.ide a ⟹ AB.Map (Trn u) a = AB.Map (Trn v) a by simp
  qed
 qed
qed

lemma sta-extensionality:
assumes «f : a →sta b» and «g : a →sta b»
shows f = g ⟷ (∀t. «t : 𝟭 → a» ⟶ f ⋆ t = g ⋆ t)
proof
 have a: obj a and b: obj b
  using assms(1) by auto
 have A: small-rts (Dom a) ∧ extensional-rts (Dom a)
 and B: small-rts (Dom b) ∧ extensional-rts (Dom b)
  using a b obj-char arr-char by blast+
 interpret A: extensional-rts ‹Dom a›
  using A by blast
 interpret B: extensional-rts ‹Dom b›
  using B by blast
 interpret AB: exponential-rts ‹Dom a› ‹Dom b› ..
 have Dom f = Dom a and Cod f = Dom b
  using assms(1) Dom-dom Dom-cod by auto
 have Dom g = Dom a and Cod g = Dom b
  using assms(2) Dom-dom Dom-cod by auto
 interpret F: simulation ‹Dom a› ‹Dom b› ‹Map f›
  using assms(1) sta-char [of f] AB.ide-char_ERTS [of Trn f]
      ‹Dom f = Dom a› ‹Cod f = Dom b›
  by simp
 interpret G: simulation ‹Dom a› ‹Dom b› ‹Map g›
  using assms(2) sta-char [of g] AB.ide-char_ERTS [of Trn g]
      ‹Dom g = Dom a› ‹Cod g = Dom b›
  by simp
 show f = g ⟹ ∀t. «t : 𝟭 → a» ⟶ f ⋆ t = g ⋆ t
  by blast
 show ∀t. «t : 𝟭 → a» ⟶ f ⋆ t = g ⋆ t ⟹ f = g
 proof −
  assume 1: ∀t. «t : 𝟭 → a» ⟶ f ⋆ t = g ⋆ t
  have ⋀Q R T. transformation One.resid (Dom a) Q R T
                ⟹ Map f ∘ T = Map g ∘ T
  proof −
   fix Q R T
   assume 2: transformation One.resid (Dom a) Q R T
   interpret T: transformation One.resid ‹Dom a› Q R T
```

```
      using 2 by blast
    let ?t = mkarr One.resid (Dom a) Q R T
    have t: «?t : 1 → a»
      by (metis (no-types, lifting) 2 A H.ideD(2) H.ideD(3) H.ide-in-hom
          H.in-homI One.is-extensional-rts One.small-rts-axioms Trm.obj-one
          Trm.one-universality(2) a arr-coincidence arr-mkarr(1) arr-mkarr(5)
          mkarr-simps(1) mkobj-Dom trm-def trm-simps(3))
    show Map f ∘ T = Map g ∘ T
      by (metis (no-types, lifting) 1 AB.Map.simps(1) H.seqI' mkarr-def
          Map-hcomp Trn.simps(1) assms(1) comp-apply t)
  qed
  hence 2: Map f = Map g
    using One.eq-simulation-iff F.simulation-axioms G.simulation-axioms
        A.weakly-extensional-rts-axioms B.weakly-extensional-rts-axioms
    by blast
  have f = mksta (Dom a) (Dom b) (Map f)
    using assms(1) a b A B obj-char arr-char bij-mksta(4) by fastforce
  also have ... = mksta (Dom a) (Dom b) (Map g)
    using 2 by simp
  also have ... = g
    using assms(2) a b A B obj-char arr-char bij-mksta(4) by fastforce
  finally show f = g by auto
 qed
qed
```

The mapping *HOM* **1**, like *Dom*, takes each object to a corresponding RTS, but unlike *Dom* it stays at type *'A arr*, rather than decreasing the type from *'A arr* to *'A*.

```
lemma HOM1-mapsto:
shows HOM 1 ∈ Collect obj → Collect extensional-rts ∩ Collect small-rts
proof
  fix a
  assume a: a ∈ Collect obj
  have A: extensional-rts (HOM 1 a)
    using a extensional-rts-HOM by blast
  interpret HOM-1A: sub-rts resid ‹λt. «t : 1 → a»›
    using a sub-rts-HOM by simp
  have small-rts HOM-1A.resid
  proof −
    have Collect HOM-1A.arr ⊆ H.hom 1 a
      using HOM-1A.arr-char by blast
    moreover have small (H.hom 1 a)
      using a small-homs by blast
    ultimately show ?thesis
      using smaller-than-small small-rts-def HOM-1A.rts-axioms
          small-rts-axioms-def
      by blast
  qed
  moreover have extensional-rts HOM-1A.resid
```

```
    using A by blast
  ultimately show HOM 1 a ∈ Collect extensional-rts ∩ Collect small-rts
    by blast
qed
```

The mapping *HOM* **1** is not necessarily injective, but it is essentially so.

```
lemma HOM1-reflects-isomorphic:
assumes obj a and obj b and isomorphic-rts (HOM 1 a) (HOM 1 b)
shows H.isomorphic a b
proof −
  have 1: isomorphic-rts (Dom a) (Dom b)
  proof −
    have isomorphic-rts (Dom a) (HOM 1 a)
      using assms(1) inverse-simulations-DN-UP(2) by blast
    also have isomorphic-rts ... (HOM 1 b)
      using assms(3) by blast
    also have isomorphic-rts ... (Dom b)
      using assms(2) inverse-simulations-DN-UP(2) isomorphic-rts-symmetric
      by blast
    finally show ?thesis by blast
  qed
  obtain F G where FG: inverse-simulations (Dom b) (Dom a) F G
    using 1 isomorphic-rts-def isomorphic-rts-symmetric by blast
  interpret FG: inverse-simulations ‹Dom b› ‹Dom a› F G
    using FG by blast
  let ?f = mksta (Dom a) (Dom b) F
  let ?g = mksta (Dom b) (Dom a) G
  have f: «?f : a →sta b» and g: «?g : b →sta a» ∧ sta ?g
    using assms(1−2) FG.F.simulation-axioms FG.G.simulation-axioms
          bij-mksta(1) obj-char [of a] obj-char [of b]
          obj-is-sta sta-char sta-mksta(1−3)
    by auto
  have H.inverse-arrows ?f ?g
  proof
    show obj (?g ⋆ ?f)
    proof −
      have ?g ⋆ ?f = a
      proof −
        have gf: «?g ⋆ ?f : a →sta a»
         using f g Cod.simps(1) Dom.simps(1) H.cod-comp H.dom-comp H-seqI
               V.ide-implies-arr sta-hcomp
          by blast
        have ?g ⋆ ?f = mksta (Dom a) (Dom a) (Map (?g ⋆ ?f))
          using f g gf
          by (metis (no-types, lifting) Cod-mkarr Dom-mkarr Int-Collect
              Src-mkarr Trg-mkarr V.ide-implies-arr assms(1−2) bij-mksta(3)
              inf-idem mkarr-comp objE transformation-Map-arr)
        also have ... = mksta (Dom a) (Dom a) (Map ?g ∘ Map ?f)
          using gf H.arrI Map-hcomp by force
```

```
      also have ... = mksta (Dom a) (Dom a) (G ∘ F)
        using assms obj-char arr-char FG.F.simulation-axioms
              FG.G.simulation-axioms bij-mksta(3)
        by auto
      also have ... = mksta (Dom a) (Dom a) (I (Dom a))
        using FG.inv by simp
      also have ... = a
        using assms obj-char mkobj-def mkarr-def by simp
      finally show ?thesis by blast
    qed
    thus obj (?g ⋆ ?f)
      using assms by simp
  qed
  show obj (?f ⋆ ?g)
  proof −
    have ?f ⋆ ?g = b
    proof −
      have fg: «?f ⋆ ?g : b →sta b»
       using f g Cod.simps(1) Dom.simps(1) H.cod-comp H.dom-comp H-seqI
             V.ide-implies-arr sta-hcomp
        by blast
      have ?f ⋆ ?g = mksta (Dom b) (Dom b) (Map (?f ⋆ ?g))
        using f g fg
        by (metis (no-types, lifting) Cod-mkarr Dom-mkarr Int-Collect
            Src-mkarr Trg-mkarr V.ide-implies-arr assms(1−2) bij-mksta(3)
            inf-idem mkarr-comp objE transformation-Map-arr)
      also have ... = mksta (Dom b) (Dom b) (Map ?f ∘ Map ?g)
        using fg H.arrI Map-hcomp by force
      also have ... = mksta (Dom b) (Dom b) (F ∘ G)
        using assms obj-char arr-char FG.F.simulation-axioms
              FG.G.simulation-axioms bij-mksta(3)
        by auto
      also have ... = mksta (Dom b) (Dom b) (I (Dom b))
        using FG.inv' by simp
      also have ... = b
        using assms obj-char mkobj-def mkarr-def by simp
      finally show ?thesis by blast
    qed
    thus obj (?f ⋆ ?g)
      using assms by simp
  qed
qed
hence «?f : a → b» ∧ H.iso ?f
  using f by blast
thus ?thesis
  using H.isomorphic-def by blast
qed

end
```

4.3.2 Products

In this section we show that the category $\mathbf{RTS}^{\dagger}$ has products. A product of objects a and b is obtained by constructing the product $Dom\ a \times Dom\ b$ of their underlying RTS's and then showing that there exists an object $a \otimes b$ such that $Dom\ (a \otimes b)$ is isomorphic to $Dom\ a \times Dom\ b$. Since $Dom\ (a \otimes b)$ will have arrow type $'A$, but $Dom\ a \otimes Dom\ b$ has arrow type $'A \times 'A$, we need a way to reduce the arrow type of $Dom\ a \otimes Dom\ b$ from $'A \times 'A$ to $'A$. This is done by using the assumption that the type $'A$ admits pairing to obtain an injective map from $'A \times 'A$ to $'A$, and then applying the injective image construction to obtain an RTS with arrow type $'A$ that is isomorphic to $Dom\ a \otimes Dom\ b$.

```
locale product-in-rtscat =
  rtscatx arr-type
for arr-type :: 'A itself
and a
and b +
assumes obj-a: obj a
and obj-b: obj b
begin

  notation hcomp  (infixr ⋆ 53)

  interpretation A: extensional-rts ‹Dom a›
    using obj-a bij-mkobj obj-char by blast
  interpretation A: small-rts ‹Dom a›
    using obj-a bij-mkobj obj-char by blast
  interpretation B: extensional-rts ‹Dom b›
    using obj-b bij-mkobj obj-char by blast
  interpretation B: small-rts ‹Dom b›
    using obj-b bij-mkobj obj-char by blast
  interpretation AB: exponential-rts ‹Dom a› ‹Dom b› ..

  sublocale PROD: product-rts ‹Dom a› ‹Dom b› ..
  sublocale PROD: product-of-extensional-rts ‹Dom a› ‹Dom b› ..
  sublocale PROD: product-of-small-rts ‹Dom a› ‹Dom b› ..

  sublocale Prod: inj-image-rts pairing.some-pair PROD.resid
    by (metis (no-types, opaque-lifting) PROD.rts-axioms
        inj-image-rts-axioms-def inj-image-rts-def inj-on-subset
        inj-some-pair top-greatest)
  sublocale Prod: small-rts Prod.resid
    using PROD.small-rts-axioms Prod.preserves-reflects-small-rts
    by unfold-locales (simp add: small-rts.small)
  sublocale Prod: extensional-rts Prod.resid
    using PROD.extensional-rts-axioms Prod.preserves-extensional-rts
    by unfold-locales (simp add: extensional-rts.extensional)
```

The injective image construction on RTS's gives us invertible simulations

between *Prod.resid* and *PROD.resid*.

abbreviation *Pack* :: $'A \times 'A \Rightarrow 'A$
where $Pack \equiv Prod.map_{ext}$

abbreviation *Unpack* :: $'A \Rightarrow 'A \times 'A$
where $Unpack \equiv Prod.map'_{ext}$

interpretation P_1: *composite-simulation Prod.resid PROD.resid ‹Dom a› Unpack PROD.*P_1
..
interpretation P_0: *composite-simulation Prod.resid PROD.resid ‹Dom b› Unpack PROD.*P_0
..

abbreviation *prod* :: *'A arr*
where *prod* ≡ *mkobj Prod.resid*

lemma *obj-prod*:
shows *obj prod*
using *obj-mkobj Prod.extensional-rts-axioms Prod.small-rts-axioms* **by** *blast*

lemma *Dom-prod* [*simp*]:
shows *Dom prod = Prod.resid*
by (*simp add*: *Prod.extensional-rts-axioms Prod.small-rts-axioms*)

definition p_0 :: *'A arr*
where $p_0 \equiv$ *mksta Prod.resid (Dom b)* P_0*.map*

definition p_1 :: *'A arr*
where $p_1 \equiv$ *mksta Prod.resid (Dom a)* P_1*.map*

lemma p_0*-simps* [*simp*]:
shows *sta* p_0 **and** *dom* p_0 = *prod* **and** *cod* p_0 = *b*
and *Dom* p_0 = *Prod.resid* **and** *Cod* p_0 = *Dom b*
and *Trn* p_0 = *exponential-rts.MkIde* P_0*.map*
using p_0*-def obj-b B.extensional-rts-axioms B.small-rts-axioms* P_0*.simulation-axioms Prod.extensional-rts-axioms Prod.small-rts-axioms sta-mksta*(*1*) *H.dom-eqI H.cod-eqI H-seqI obj-char obj-prod mkarr-def*
by *auto*

lemma p_1*-simps* [*simp*]:
shows *sta* p_1 **and** *dom* p_1 = *prod* **and** *cod* p_1 = *a*
and *Dom* p_1 = *Prod.resid* **and** *Cod* p_1 = *Dom a*
and *Trn* p_1 = *exponential-rts.MkIde* P_1*.map*
using p_1*-def obj-a A.extensional-rts-axioms A.small-rts-axioms* P_1*.simulation-axioms Prod.extensional-rts-axioms Prod.small-rts-axioms sta-mksta*(*1*) *H.dom-eqI H.cod-eqI H-seqI obj-char obj-prod mkarr-def*

by *auto*

lemma p_0*-in-hom* [*intro*]:
shows «p_0 : *prod* → *b*»
by *auto*

lemma p_1*-in-hom* [*intro*]:
shows «p_1 : *prod* → *a*»
by *auto*

It should be noted that the length of the proof of the following result is partly due to the fact that it is proving something rather stronger than one might expect at first blush. The category we are working with here is analogous to a 2-category in the sense that there are essentially two classes of arrows: *states*, which correspond to simulations between RTS's, and *transitions*, which correspond to transformations. The class of states is included in the class of transitions. The universality result below shows the universality of the product for the full class of arrows, so it is in that sense analogous to showing that the category has 2-products, rather than just ordinary products.

lemma *universality*:
assumes «*h* : *x* → *a*» **and** «*k* : *x* → *b*»
shows ∃!*m*. p_1 ⋆ *m* = *h* ∧ p_0 ⋆ *m* = *k*
proof
interpret *X*: *extensional-rts* ‹*Dom x*›
using *assms*(*1*) *H.in-homE H-arr-char dom-char* **by** *auto*
interpret *X*: *small-rts* ‹*Dom x*›
using *assms*(*1*) *H.in-homE H-arr-char dom-char* **by** *auto*
interpret *A*: *extensional-rts* ‹*Dom a*›
using *assms*(*1*) *H.in-homE H-arr-char cod-char* **by** *auto*
interpret *A*: *small-rts* ‹*Dom a*›
using *assms*(*1*) *H.in-homE H-arr-char cod-char* **by** *auto*
interpret *B*: *extensional-rts* ‹*Dom b*›
using *assms*(*2*) *H.in-homE H-arr-char cod-char* **by** *auto*
interpret *B*: *small-rts* ‹*Dom b*›
using *assms*(*2*) *H.in-homE H-arr-char cod-char* **by** *auto*
interpret *XA*: *exponential-rts* ‹*Dom x*› ‹*Dom a*› **..**
interpret *XB*: *exponential-rts* ‹*Dom x*› ‹*Dom b*› **..**
have ∗: *Dom h* = *Dom x* ∧ *Cod h* = *Dom a* ∧
Dom k = *Dom x* ∧ *Cod k* = *Dom b*
using *assms*(*1*−*2*) *dom-char cod-char* **by** *auto*
interpret H_0: *simulation* ‹*Dom x*› ‹*Dom a*› ‹*Map* (*src h*)›
by (*metis* (*mono-tags*, *lifting*) ∗ *H.arrI H-arr-char Trn.simps*(*1*)
$XA.ide\text{-}char_{ERTS}$ *XA.ide-src arr-char assms*(*1*) *comp-apply src-char*)
interpret H_1: *simulation* ‹*Dom x*› ‹*Dom a*› ‹*Map* (*trg h*)›
by (*metis* (*mono-tags*, *lifting*) ∗ *H.arrI H-arr-char Trn.simps*(*1*)
$XA.ide\text{-}char_{ERTS}$ *XA.ide-trg arr-char assms*(*1*) *comp-apply trg-char*)
interpret K_0: *simulation* ‹*Dom x*› ‹*Dom b*› ‹*Map* (*src k*)›

by (*metis* (*mono-tags*, *lifting*) ∗ *H.arrI H-arr-char Map-simps*(*3*) *XB.arrE arr-char assms*(*2*) *comp-apply transformation-def*)
interpret K_1: *simulation* ‹*Dom x*› ‹*Dom b*› ‹*Map* (*trg k*)›
by (*metis* (*mono-tags*, *lifting*) ∗ *H.arrI H-arr-char Map-simps*(*4*) *XB.arrE arr-char assms*(*2*) *comp-apply transformation-def*)
interpret *H*: *transformation* ‹*Dom x*› ‹*Dom a*›
‹*Map* (*src h*)› ‹*Map* (*trg h*)› ‹*Map h*›
using ∗ *Map-simps*(*3*) *Map-simps*(*4*) *XA.arr-char arr-char assms*(*1*)
by (*metis H.arrI H-arr-char transformation-Map-arr*)
interpret *K*: *transformation* ‹*Dom x*› ‹*Dom b*›
‹*Map* (*src k*)› ‹*Map* (*trg k*)› ‹*Map k*›
using ∗ *Map-simps*(*3*) *Map-simps*(*4*) *XB.arr-char arr-char assms*(*2*) *H.arrI*
by *force*

interpret HK_0: *simulation* ‹*Dom x*› *PROD.resid*
‹⟨⟨*Map* (*src h*), *Map* (*src k*)⟩⟩›
using *assms PROD.universality*(*1*) [*of Dom h Map* (*src h*) *Map* (*src k*)] H_0.*simulation-axioms* K_0.*simulation-axioms*
by *blast*
interpret HK_1: *simulation* ‹*Dom x*› *PROD.resid*
‹⟨⟨*Map* (*trg h*), *Map* (*trg k*)⟩⟩›
using *assms PROD.universality*(*1*) [*of Dom h Map* (*trg h*) *Map* (*trg k*)] H_1.*simulation-axioms* K_1.*simulation-axioms*
by *blast*

interpret $PROD.P_1$: *simulation-as-transformation PROD.resid* ‹*Dom a*› $PROD.P_1$
..
interpret $PROD.P_0$: *simulation-as-transformation PROD.resid* ‹*Dom b*› $PROD.P_0$
..
interpret P_1: *simulation-as-transformation Prod.resid* ‹*Dom a*› P_1.*map* **..**
interpret P_0: *simulation-as-transformation Prod.resid* ‹*Dom b*› P_0.*map* **..**

interpret $P_0 oHK_0$: *composite-simulation* ‹*Dom x*› *PROD.resid* ‹*Dom b*›
‹⟨⟨*Map* (*src h*), *Map* (*src k*)⟩⟩› $PROD.P_0$
..
interpret $P_0 oHK_1$: *composite-simulation* ‹*Dom x*› *PROD.resid* ‹*Dom b*›
‹⟨⟨*Map* (*trg h*), *Map* (*trg k*)⟩⟩› $PROD.P_0$
..
interpret $P_1 oHK_0$: *composite-simulation* ‹*Dom x*› *PROD.resid* ‹*Dom a*›
‹⟨⟨*Map* (*src h*), *Map* (*src k*)⟩⟩› $PROD.P_1$
..
interpret $P_1 oHK_1$: *composite-simulation* ‹*Dom x*› *PROD.resid* ‹*Dom a*›
‹⟨⟨*Map* (*trg h*), *Map* (*trg k*)⟩⟩› $PROD.P_1$
..
interpret *HK*: *transformation* ‹*Dom x*› *PROD.resid*
‹⟨⟨*Map* (*src h*), *Map* (*src k*)⟩⟩›

```
                ‹⟨⟨Map (trg h), Map (trg k)⟩⟩›
                ‹⟨⟨Map h, Map k⟩⟩›
  using assms HK0.simulation-axioms HK1.simulation-axioms
        H.transformation-axioms K.transformation-axioms
  by (metis H0.simulation-axioms H1.simulation-axioms
      K0.simulation-axioms K1.simulation-axioms PROD.proj-tuple(1)
      PROD.universality2(1) PROD.universality(3))
interpret Pack-o-HK: transformation ‹Dom h› Prod.resid
                    ‹Pack ∘ ⟨⟨Map (src h), Map (src k)⟩⟩›
                    ‹Pack ∘ ⟨⟨Map (trg h), Map (trg k)⟩⟩›
                    ‹Pack ∘ ⟨⟨Map h, Map k⟩⟩›
  using assms transformation-whisker-left
        Prod.weakly-extensional-rts-axioms HK.transformation-axioms
        Prod.Map.simulation-axioms dom-char
  by fastforce

let ?hk = mkarr (Dom h) Prod.resid
          (Pack ∘ ⟨⟨Map (src h), Map (src k)⟩⟩)
          (Pack ∘ ⟨⟨Map (trg h), Map (trg k)⟩⟩)
          (Pack ∘ ⟨⟨Map h, Map k⟩⟩)
have hk: «?hk : dom h → prod»
  using assms arr-mkarr arr-char Pack-o-HK.transformation-axioms
        X.extensional-rts-axioms X.small-rts-axioms
        Prod.extensional-rts-axioms Prod.small-rts-axioms
        dom-char cod-char
  by auto
show p1 ⋆ ?hk = h ∧ p0 ⋆ ?hk = k
proof
  have seq0: H.seq p0 ?hk
    using hk by blast
  have seq1: H.seq p1 ?hk
    using hk by blast
  show p1 ⋆ ?hk = h
  proof (intro arr-eqI)
    show p1 ⋆ ?hk ≠ Null
      using seq1 arr-char by auto
    show h ≠ Null
      using assms arr-char [of h] by auto
    show Dom: Dom (p1 ⋆ ?hk) = Dom h
      using seq1 H-seq-char mkarr-def by fastforce
    show Cod: Cod (p1 ⋆ ?hk) = Cod h
      using ∗ H.arrI hk mkarr-def by auto
    show Trn (p1 ⋆ ?hk) = Trn h
    proof −
      interpret C: COMP ‹Dom x› Prod.resid ‹Dom a› ..
      have Trn (p1 ⋆ ?hk) =
           COMP.map (Dom ?hk) (Cod ?hk) (Cod p1) (Trn p1, Trn ?hk)
        using assms seq1 Trn-hcomp hk H.seqE mkarr-def by auto
      also have ... =
```

```
          COMP.map (Dom x) Prod.resid (Dom a) (Trn p₁, Trn ?hk)
    using assms hk dom-char mkarr-def by auto
  also have ... =
          C.BC.MkArr
           ((P₁.map ∘ Pack) ∘
             ⟨⟨C.BC.Map (Trn (src h)), C.BC.Map (Trn (src k))⟩⟩)
           ((P₁.map ∘ Pack) ∘
             ⟨⟨C.BC.Map (Trn (trg h)), C.BC.Map (Trn (trg k))⟩⟩)
           ((P₁.map ∘ Pack) ∘
             ⟨⟨C.BC.Map (Trn h), C.BC.Map (Trn k)⟩⟩)
    unfolding p₁-def C.map-eq
    using assms hk C.map-eq P₁.transformation-axioms
        Pack-o-HK.transformation-axioms dom-char cod-char mkarr-def
    by auto
  also have ... =
          C.BC.MkArr
           (PROD.P₁ ∘
             ⟨⟨C.BC.Map (Trn (src h)), C.BC.Map (Trn (src k))⟩⟩)
           (PROD.P₁ ∘
             ⟨⟨C.BC.Map (Trn (trg h)), C.BC.Map (Trn (trg k))⟩⟩)
           (PROD.P₁ ∘
             ⟨⟨C.BC.Map (Trn h), C.BC.Map (Trn k)⟩⟩)
  proof −
    have P₁.map ∘ Pack = PROD.P₁
      using PROD.P₁.extensional Prod.map-null Prod.null-char by auto
    thus ?thesis by simp
  qed
  also have ... = C.BC.MkArr
                   (C.BC.Map (Trn (src h))) (C.BC.Map (Trn (trg h)))
                   (C.BC.Map (Trn h))
    using PROD.proj-tuple2(1−2) PROD.proj-tuple(1−2)
        H.transformation-axioms K.transformation-axioms
        H₀.simulation-axioms H₁.simulation-axioms
        K₀.simulation-axioms K₁.simulation-axioms
    by auto
  also have ... = Trn h
   using assms C.BC.MkArr-Map ∗ Map-simps(3−4) XA.arr-char arr-char
    by (metis (no-types, lifting) H.arrI H-arr-char comp-def)
  finally show ?thesis by blast
 qed
qed
show p₀ ⋆ ?hk = k
proof (intro arr-eqI)
 show p₀ ⋆ ?hk ≠ Null
   using seq0 arr-char by auto
 show k ≠ Null
   using assms arr-char [of k] by auto
 show Dom: Dom (p₀ ⋆ ?hk) = Dom k
   using ∗ H-seq-char seq1 mkarr-def by auto
```

```
show Cod: Cod (p₀ ⋆ ?hk) = Cod k
  using ∗ H-seq-char seq0 by auto
show Trn (p₀ ⋆ ?hk) = Trn k
proof −
  interpret C: COMP ‹Dom x› Prod.resid ‹Dom b› ..
  have Trn (p₀ ⋆ ?hk) =
       COMP.map (Dom ?hk) (Cod ?hk) (Cod p₀) (Trn p₀, Trn ?hk)
    using assms seq0 Trn-hcomp [of p₀ ?hk] hk H.seqE mkarr-def by auto
  also have ... =
            COMP.map (Dom x) Prod.resid (Dom b) (Trn p₀, Trn ?hk)
    using assms hk dom-char mkarr-def by auto
  also have ... =
            C.BC.MkArr
              ((P₀.map ∘ Pack) ∘
                 ⟨⟨C.BC.Map (Trn (src h)), C.BC.Map (Trn (src k))⟩⟩)
              ((P₀.map ∘ Pack) ∘
                 ⟨⟨C.BC.Map (Trn (trg h)), C.BC.Map (Trn (trg k))⟩⟩)
              ((P₀.map ∘ Pack) ∘
                 ⟨⟨C.BC.Map (Trn h), C.BC.Map (Trn k)⟩⟩)
    unfolding p₀-def C.map-eq
    using assms hk C.map-eq P₀.transformation-axioms
          Pack-o-HK.transformation-axioms dom-char cod-char mkarr-def
    by auto
  also have ... =
            C.BC.MkArr
              (PROD.P₀ ∘
                 ⟨⟨C.BC.Map (Trn (src h)), C.BC.Map (Trn (src k))⟩⟩)
              (PROD.P₀ ∘
                 ⟨⟨C.BC.Map (Trn (trg h)), C.BC.Map (Trn (trg k))⟩⟩)
              (PROD.P₀ ∘
                 ⟨⟨C.BC.Map (Trn h), C.BC.Map (Trn k)⟩⟩)
  proof −
    have P₀.map ∘ Pack = PROD.P₀
      using PROD.P₀.extensional Prod.map-null Prod.null-char by auto
    thus ?thesis by simp
  qed
  also have ... = C.BC.MkArr
                    (C.BC.Map (Trn (src k))) (C.BC.Map (Trn (trg k)))
                    (C.BC.Map (Trn k))
    using PROD.proj-tuple2(1−2) PROD.proj-tuple(1−2)
          H.transformation-axioms K.transformation-axioms
          H₀.simulation-axioms H₁.simulation-axioms
          K₀.simulation-axioms K₁.simulation-axioms
    by auto
  also have ... = Trn k
    using assms C.BC.MkArr-Map [of Trn k] ∗ Map-simps(3−4)
          XB.arr-char arr-char
    by (metis (no-types, lifting) H.arrI H-arr-char comp-apply)
  finally show ?thesis by blast
```

```
    qed
  qed
qed
fix m
assume m: p₁ ⋆ m = h ∧ p₀ ⋆ m = k
have arr-m: arr m
  using assms m by fastforce
have Dom-m: Dom m = Dom x
  using assms m dom-char by fastforce
have Cod-m: Cod m = Prod.resid
  using assms m cod-char
  using H-seq-char by auto
interpret X-Prod: exponential-rts ‹Dom x› Prod.resid ..
interpret M: transformation ‹Dom x› Prod.resid
              ‹Map (src m)› ‹Map (trg m)› ‹Map m›
  using Cod-m Dom-m Map-simps(3) Map-simps(4) arr-char arr-m by auto
interpret UnpackoM: transformation ‹Dom h› PROD.resid
                      ‹Unpack ∘ Map (src m)›
                      ‹Unpack ∘ Map (trg m)›
                      ‹Unpack ∘ Map m›
  using ∗ M.transformation-axioms PROD.weakly-extensional-rts-axioms
        Prod.Map'.simulation-axioms transformation-whisker-left
  by fastforce
show m = ?hk
proof (intro arr-eqI)
  show m ≠ Null
    using assms m null-char by fastforce
  show ?hk ≠ Null
    using hk mkarr-def by auto
  show 2: Dom m = Dom ?hk
    using assms m hk cod-char ∗ Dom.simps(1) Dom-m mkarr-def
    by presburger
  show 3: Cod m = Cod ?hk
    using assms m hk cod-char mkarr-def
    by (simp add: Cod-m)
  show Trn m = Trn ?hk
  proof (intro X-Prod.arr-eqI)
    interpret COMPa: COMP ‹Dom x› Prod.resid ‹Dom a› ..
    interpret COMPb: COMP ‹Dom x› Prod.resid ‹Dom b› ..
    show 4: X-Prod.arr (Trn m)
      using assms arr-m Dom-m Cod-m arr-char by simp
    show 5: X-Prod.arr (Trn ?hk)
      using 2 Dom-m H-arr-char hk mkarr-def by force
    show 6: X-Prod.Dom (Trn m) = X-Prod.Dom (Trn ?hk)
    proof −
      have PROD.P₁ ∘ (Unpack ∘ X-Prod.Dom (Trn ?hk)) =
           PROD.P₁ ∘ (Unpack ∘ X-Prod.Dom (Trn m))
      proof −
        have PROD.P₁ ∘ (Unpack ∘ X-Prod.Dom (Trn ?hk)) =
```

```
      PROD.P₁ ∘ (Unpack ∘ Pack) ∘
       ⟨⟨COMPa.BC.Map (Trn (src h)),
         COMPa.BC.Map (Trn (src k))⟩⟩
  using mkarr-def by auto
also have ... = COMPa.BC.Map (Trn (src h))
proof
  fix x
  show (PROD.P₁ ∘ (Unpack ∘ Pack) ∘
          ⟨⟨COMPa.BC.Map (Trn (src h)),
            COMPa.BC.Map (Trn (src k))⟩⟩) x =
       COMPa.BC.Map (Trn (src h)) x
    using PROD.P₁-def
    apply (auto simp add: pointwise-tuple-def)[1]
     apply (metis A.not-arr-null PROD.null-char
      Prod.null-char first-conv)
    apply (metis (no-types, opaque-lifting) H₀.extensional
      H₀.simulation-axioms comp-apply
      simulation.preserves-reflects-arr)
   apply (metis B.not-arr-null PROD.null-char Prod.null-char
      second-conv)
  by (metis (no-types, opaque-lifting) A.not-arr-null
      H₀.extensional P₁oHK₀.preserves-reflects-arr comp-def
      pointwise-tuple-def)
qed
also have ... = COMPa.BC.Map (Trn (src (p₁ ⋆ m)))
  using m by blast
also have ... = COMPa.BC.Map (Trn (p₁ ⋆ src m))
  using assms m by auto
also have ... =
          COMPa.BC.Map (COMPa.map (Trn p₁, Trn (src m)))
  using arr-m Dom-m Cod-m Trn-hcomp by simp
also have ... =
          COMPa.BC.Map (Trn p₁) ∘ COMPa.BC.Map (Trn (src m))
proof −
  have COMPa.BCxAB.arr (COMPa.BC.MkIde P₁.map, Trn m)
    using assms arr-m Dom-m Cod-m arr-char arr-char p₁-simps(1)
        P₁.transformation-axioms
    by auto
  thus ?thesis
    unfolding COMPa.map-eq
    using assms m arr-m Dom-m Cod-m arr-char [of src m] by simp
qed
also have ... =
          (PROD.P₁ ∘ Unpack) ∘ COMPa.BC.Map (Trn (src m))
  by simp
also have ... = PROD.P₁ ∘ (Unpack ∘ X-Prod.Dom (Trn m))
  by (auto simp add: 4 Cod-m Dom-m arr-m src-char)
finally show ?thesis by blast
qed
```

```
moreover have PROD.P₀ ∘ (Unpack ∘ X-Prod.Dom (Trn ?hk)) =
                PROD.P₀ ∘ (Unpack ∘ X-Prod.Dom (Trn m))
proof −
  have PROD.P₀ ∘ (Unpack ∘ X-Prod.Dom (Trn ?hk)) =
       PROD.P₀ ∘ (Unpack ∘ Pack) ∘
       ⟨⟨COMPb.BC.Map (Trn (src h)),
         COMPb.BC.Map (Trn (src k))⟩⟩
    using mkarr-def by auto
  also have ... = COMPb.BC.Map (Trn (src k))
  proof
    fix x
    show (PROD.P₀ ∘ (Unpack ∘ Pack) ∘
            ⟨⟨COMPb.BC.Map (Trn (src h)),
              COMPb.BC.Map (Trn (src k))⟩⟩) x =
         COMPb.BC.Map (Trn (src k)) x
      using PROD.P₀-def H₀.preserves-reflects-arr K₀.extensional
            PROD.null-char Prod.null-char
      apply (auto simp add: pointwise-tuple-def)[1]
       apply (metis second-conv)
      apply (metis A.not-arr-null PROD.null-char first-conv)
     by (metis (no-types, opaque-lifting) B.not-arr-null
         P₀oHK₀.preserves-reflects-arr comp-def pointwise-tuple-def)
  qed
  also have ... = COMPb.BC.Map (Trn (src (p₀ ⋆ m)))
    using m by blast
  also have ... = COMPb.BC.Map (Trn (p₀ ⋆ src m))
    using assms m by auto
 also have ... = COMPb.BC.Map (COMPb.map (Trn p₀, Trn (src m)))
    using arr-m Dom-m Cod-m Trn-hcomp by simp
  also have ... = COMPb.BC.Map (Trn p₀) ∘
                  COMPb.BC.Map (Trn (src m))
  proof −
    have COMPb.BCxAB.arr (COMPb.BC.MkIde P₀.map, Trn m)
      using assms arr-m Dom-m Cod-m arr-char arr-char p₀-simps(1)
            P₀.transformation-axioms
      by auto
    thus ?thesis
      unfolding COMPb.map-eq
      using assms m arr-m Dom-m Cod-m arr-char [of src m]
      by simp
  qed
  also have ... =
            (PROD.P₀ ∘ Unpack) ∘ COMPb.BC.Map (Trn (src m))
    by simp
  also have ... = PROD.P₀ ∘ (Unpack ∘ X-Prod.Dom (Trn m))
    by (auto simp add: 4 Cod-m Dom-m arr-m src-char)
  finally show ?thesis by blast
qed
moreover have simulation (Dom x) PROD.resid
```

```
                (Unpack ∘ X-Prod.Dom (Trn ?hk))
    using hk arr-char 2 Pack-o-HK.F.simulation-axioms Dom-m
        Prod.Map'.simulation-axioms simulation-comp mkarr-def
    by auto
  moreover have simulation (Dom x) PROD.resid
                (Unpack ∘ X-Prod.Dom (Trn m))
    using 4 X-Prod.ide-src Prod.Map'.simulation-axioms
        simulation-comp
    by auto
  ultimately have Unpack ∘ X-Prod.Dom (Trn ?hk) =
                  Unpack ∘ X-Prod.Dom (Trn m)
    using PROD.proj-joint-monic by blast
 moreover have simulation (Dom x) Prod.resid (X-Prod.Dom (Trn ?hk))
    using hk arr-char X-Prod.ide-src 2 Pack-o-HK.F.simulation-axioms
        Dom-m mkarr-def
    by fastforce
 moreover have simulation (Dom x) Prod.resid (X-Prod.Dom (Trn m))
    using 4 X-Prod.ide-src by auto
 ultimately show ?thesis
    using invertible-simulation-cancel-left
    by (metis (no-types, lifting) Prod.invertible-simulation-map')
qed
show 7: X-Prod.Cod (Trn m) = X-Prod.Cod (Trn ?hk)
proof −
  have PROD.P₁ ∘ (Unpack ∘ X-Prod.Cod (Trn ?hk)) =
       PROD.P₁ ∘ (Unpack ∘ X-Prod.Cod (Trn m))
  proof −
    have PROD.P₁ ∘ (Unpack ∘ X-Prod.Cod (Trn ?hk)) =
         PROD.P₁ ∘ (Unpack ∘ Pack) ∘
           ⟨⟨COMPa.BC.Map (Trn (trg h)),
             COMPa.BC.Map (Trn (trg k))⟩⟩
      using mkarr-def by auto
    also have ... = COMPa.BC.Map (Trn (trg h))
    proof
      fix x
      show (PROD.P₁ ∘ (Unpack ∘ Pack) ∘
              ⟨⟨COMPa.BC.Map (Trn (trg h)),
                COMPa.BC.Map (Trn (trg k))⟩⟩) x =
           COMPa.BC.Map (Trn (trg h)) x
        using PROD.P₁-def
        apply (auto simp add: pointwise-tuple-def)[1]
        subgoal by (metis A.not-arr-null PROD.null-char Prod.null-char
            first-conv)
        subgoal using H₁.extensional H₁.preserves-reflects-arr by auto
        subgoal by (metis B.not-arr-null PROD.null-char Prod.null-char
            second-conv)
        subgoal by (metis (mono-tags, lifting) H₁.simulation-axioms
            K₁.simulation-axioms comp-def simulation.extensional
            simulation.preserves-reflects-arr)
```

```
    done
  qed
  also have ... = COMPa.BC.Map (Trn (trg (p₁ ⋆ m)))
    using m by blast
  also have ... = COMPa.BC.Map (Trn (p₁ ⋆ trg m))
    using assms m by auto
 also have ... = COMPa.BC.Map (COMPa.map (Trn p₁, Trn (trg m)))
    using arr-m Dom-m Cod-m by simp
  also have ... =
            COMPa.BC.Map (Trn p₁) ∘ COMPa.BC.Map (Trn (trg m))
  proof −
    have COMPa.BCxAB.arr (COMPa.BC.MkIde P₁.map, Trn m)
      using assms arr-m Dom-m Cod-m arr-char arr-char p₁-simps(1)
            P₁.transformation-axioms
      by auto
    thus ?thesis
      unfolding COMPa.map-eq
      using assms m arr-m Dom-m Cod-m arr-char [of trg m] by simp
  qed
  also have ... =
            (PROD.P₁ ∘ Unpack) ∘ COMPa.BC.Map (Trn (trg m))
    by simp
  also have ... = PROD.P₁ ∘ (Unpack ∘ X-Prod.Cod (Trn m))
    by (auto simp add: 4 Cod-m Dom-m arr-m trg-char)
  finally show ?thesis by blast
qed
moreover have PROD.P₀ ∘ (Unpack ∘ X-Prod.Cod (Trn ?hk)) =
              PROD.P₀ ∘ (Unpack ∘ X-Prod.Cod (Trn m))
proof −
  have PROD.P₀ ∘ (Unpack ∘ X-Prod.Cod (Trn ?hk)) =
       PROD.P₀ ∘ (Unpack ∘ Pack) ∘
         ⟨⟨COMPb.BC.Map (Trn (trg h)),
           COMPb.BC.Map (Trn (trg k))⟩⟩
    using mkarr-def by auto
  also have ... = COMPb.BC.Map (Trn (trg k))
  proof
    fix x
    show (PROD.P₀ ∘ (Unpack ∘ Pack) ∘
            ⟨⟨COMPb.BC.Map (Trn (trg h)),
              COMPb.BC.Map (Trn (trg k))⟩⟩) x =
         COMPb.BC.Map (Trn (trg k)) x
      using PROD.P₀-def H₀.preserves-reflects-arr K₀.extensional
            K₁.extensional PROD.null-char Prod.null-char
            H₁.preserves-reflects-arr second-conv
      apply (auto simp add: pointwise-tuple-def)[1]
        apply metis
       apply (metis B.not-arr-null)
      using K₁.extensional K₁.preserves-reflects-arr by fastforce
  qed
```

```
    also have ... = COMPb.BC.Map (Trn (trg (p₀ ⋆ m)))
      using m by blast
    also have ... = COMPb.BC.Map (Trn (p₀ ⋆ trg m))
      using assms m by auto
    also have ... =
              COMPb.BC.Map (COMPb.map (Trn p₀, Trn (trg m)))
      using arr-m Dom-m Cod-m by simp
    also have ... =
              COMPb.BC.Map (Trn p₀) ∘ COMPb.BC.Map (Trn (trg m))
    proof −
      have COMPb.BCxAB.arr (COMPb.BC.MkIde P₀.map, Trn m)
        using assms arr-m Dom-m Cod-m arr-char arr-char p₀-simps(1)
              P₀.transformation-axioms
        by auto
      thus ?thesis
        unfolding COMPb.map-eq
        using assms m arr-m Dom-m Cod-m arr-char [of trg m]
        by simp
    qed
     also have ... = (PROD.P₀ ∘ Unpack) ∘ COMPb.BC.Map (Trn (trg
m))
      by simp
    also have ... = PROD.P₀ ∘ (Unpack ∘ X-Prod.Cod (Trn m))
      by (auto simp add: 4 Cod-m Dom-m arr-m trg-char)
    finally show ?thesis by blast
  qed
  moreover have simulation (Dom x) PROD.resid
                 (Unpack ∘ X-Prod.Cod (Trn ?hk))
    using hk arr-char 2 Pack-o-HK.G.simulation-axioms Dom-m
          Prod.Map'.simulation-axioms simulation-comp
    using mkarr-def by auto
  moreover have simulation (Dom x) PROD.resid
                 (Unpack ∘ X-Prod.Cod (Trn m))
    using X-Prod.ide-trg ‹X-Prod.arr (Trn m)›
          Prod.Map'.simulation-axioms simulation-comp
    by auto
  ultimately have Unpack ∘ X-Prod.Cod (Trn ?hk) =
                  Unpack ∘ X-Prod.Cod (Trn m)
    using PROD.proj-joint-monic by blast
  moreover have simulation (Dom x) Prod.resid
                 (X-Prod.Cod (Trn ?hk))
    using hk 2 arr-char X-Prod.ide-trg Dom-m mkarr-def
          Pack-o-HK.F.simulation-axioms Pack-o-HK.G.simulation-axioms
    by force
  moreover have simulation (Dom x) Prod.resid
                 (X-Prod.Cod (Trn m))
    using 4 X-Prod.ide-trg by auto
  ultimately show ?thesis
    using invertible-simulation-cancel-left
```

```
    by (metis (no-types, lifting) Prod.invertible-simulation-map')
 qed
show ⋀x. X.ide x ⟹ X-Prod.Map (Trn m) x = X-Prod.Map (Trn ?hk) x
 proof −
  have PROD.P₁ ∘ (Unpack ∘ X-Prod.Map (Trn ?hk)) =
       PROD.P₁ ∘ (Unpack ∘ X-Prod.Map (Trn m))
  proof −
    have PROD.P₁ ∘ (Unpack ∘ X-Prod.Map (Trn ?hk)) =
         PROD.P₁ ∘ (Unpack ∘ Pack) ∘
           ⟨⟨COMPa.BC.Map (Trn h), COMPa.BC.Map (Trn k)⟩⟩
      using mkarr-def by auto
    also have ... = COMPa.BC.Map (Trn h)
    proof
      fix x
      show (PROD.P₁ ∘ (Unpack ∘ Pack) ∘
            ⟨⟨COMPa.BC.Map (Trn h),
              COMPa.BC.Map (Trn k)⟩⟩) x =
           COMPa.BC.Map (Trn h) x
        using PROD.P₁-def
        apply (auto simp add: pointwise-tuple-def)[1]
          apply (metis A.not-arr-null PROD.null-char Prod.null-char
            first-conv)
         apply (metis (mono-tags, lifting) H.transformation-axioms
           K.transformation-axioms PROD.P₁.extensional
           PROD.P₁.preserves-arr PROD.proj-tuple2(1) comp-apply)
        apply (metis B.not-arr-null PROD.null-char Prod.null-char
          second-conv)
       by (metis H.extensional K.preserves-arr comp-apply)
    qed
    also have ... = COMPa.BC.Map (Trn (p₁ ⋆ m))
      using m by blast
    also have ... = COMPa.BC.Map (COMPa.map (Trn p₁, Trn m))
      using arr-m Dom-m Cod-m Trn-hcomp by simp
   also have ... = COMPa.BC.Map (Trn p₁) ∘ COMPa.BC.Map (Trn m)
    proof −
      have COMPa.BCxAB.arr (COMPa.BC.MkIde P₁.map, Trn m)
        using assms arr-m Dom-m Cod-m arr-char arr-char p₁-simps(1)
              P₁.transformation-axioms
        by auto
      thus ?thesis
        unfolding COMPa.map-eq
        using assms m arr-m Dom-m Cod-m by simp
    qed
    also have ... = (PROD.P₁ ∘ Unpack) ∘ COMPa.BC.Map (Trn m)
      by simp
    also have ... = PROD.P₁ ∘ (Unpack ∘ X-Prod.Map (Trn m))
      by (auto simp add: 4 Cod-m Dom-m arr-m src-char)
    finally show ?thesis by blast
  qed
```

```
moreover have PROD.P₀ ∘ (Unpack ∘ X-Prod.Map (Trn ?hk)) =
                PROD.P₀ ∘ (Unpack ∘ X-Prod.Map (Trn m))
proof −
  have PROD.P₀ ∘ (Unpack ∘ X-Prod.Map (Trn ?hk)) =
       PROD.P₀ ∘ (Unpack ∘ Pack) ∘
         ⟨⟨COMPb.BC.Map (Trn h),
           COMPb.BC.Map (Trn k)⟩⟩
    using mkarr-def by auto
  also have ... = COMPb.BC.Map (Trn k)
  proof
    fix x
    show (PROD.P₀ ∘ (Unpack ∘ Pack) ∘
            ⟨⟨COMPb.BC.Map (Trn h), COMPb.BC.Map (Trn k)⟩⟩) x =
         COMPb.BC.Map (Trn k) x
      using PROD.P₀-def H₀.preserves-reflects-arr K₀.extensional
            PROD.null-char Prod.null-char
      apply (auto simp add: pointwise-tuple-def)[1]
        apply (metis B.not-arr-null second-conv)
       apply (metis H.preserves-arr K.extensional comp-apply)
      apply (metis A.not-arr-null PROD.null-char first-conv)
      by (metis K.extensional K.preserves-arr comp-apply)
  qed
  also have ... = COMPb.BC.Map (Trn (p₀ ⋆ m))
    using m by blast
  also have ... = COMPb.BC.Map (COMPb.map (Trn p₀, Trn m))
    using arr-m Dom-m Cod-m Trn-hcomp by simp
  also have ... =
            COMPb.BC.Map (Trn p₀) ∘ COMPb.BC.Map (Trn m)
  proof −
    have COMPb.BCxAB.arr (COMPb.BC.MkIde P₀.map, Trn m)
      using assms arr-m Dom-m Cod-m arr-char p₀-simps(1)
            P₀.transformation-axioms
      by auto
    thus ?thesis
      unfolding COMPb.map-eq
      using assms m arr-m Dom-m Cod-m by simp
  qed
  also have ... =
            (PROD.P₀ ∘ Unpack) ∘ COMPb.BC.Map (Trn m)
    by simp
  also have ... = PROD.P₀ ∘ (Unpack ∘ X-Prod.Map (Trn m))
    by (auto simp add: 4 Cod-m Dom-m arr-m src-char)
  finally show ?thesis by blast
qed
moreover have transformation (Dom x) PROD.resid
                (Unpack ∘ Map (src m)) (Unpack ∘ Map (trg m))
                (Unpack ∘ X-Prod.Map (Trn m))
  by (metis ∗ UnpackoM.transformation-axioms comp-def)
moreover have transformation (Dom x) PROD.resid
```

```
                    (Unpack ∘ Map (src m)) (Unpack ∘ Map (trg m))
                    (Unpack ∘ X-Prod.Map (Trn ?hk))
        proof −
        have Map (src m) = X-Prod.Dom (Trn m) ∧ Map (trg m) = X-Prod.Cod
(Trn m)
          by (metis Map-simps(3−4) arr-m comp-def)
           hence Map (src m) = X-Prod.Dom (Trn ?hk) ∧ Map (trg m) =
X-Prod.Cod (Trn ?hk)
          using 6 7 by simp
        thus ?thesis
          using 5 Prod.Map'.simulation-axioms X-Prod.arr-char [of Trn ?hk]
                PROD.weakly-extensional-rts-axioms
                transformation-whisker-left
                  [of Dom x Prod.resid Map (src m) Map (trg m)
                      X-Prod.Map (Trn ?hk) PROD.resid Unpack]
          by metis
        qed
        ultimately have Unpack ∘ X-Prod.Map (Trn ?hk) =
                        Unpack ∘ X-Prod.Map (Trn m)
          using 4 5 X-Prod.arr-char
                PROD.proj-joint-monic2
                  [of Dom x Unpack ∘ Map (src m) Unpack ∘ Map (trg m)
                     Unpack ∘ X-Prod.Map (Trn ?hk) Unpack ∘ X-Prod.Map (Trn
m)]
          by metis
        moreover have Pack ∘ ⟨⟨Map (src h), Map (src k)⟩⟩ = Map (src m)
          by (simp add: 4 Cod-m Dom-m
              ‹COMPb.BC.Dom (Trn m) = COMPb.BC.Dom (Trn ?hk)›
              arr-m src-char mkarr-def)
        moreover have Pack ∘ ⟨⟨Map (trg h), Map (trg k)⟩⟩ = Map (trg m)
          by (simp add: 4 Cod-m Dom-m
              ‹COMPb.BC.Cod (Trn m) = COMPb.BC.Cod (Trn ?hk)›
              arr-m trg-char mkarr-def)
        ultimately have X-Prod.Map (Trn ?hk) = X-Prod.Map (Trn m)
          using assms 2 Dom-m Prod.invertible-simulation-map'
                invertible-simulation-cancel-left'
                M.transformation-axioms Pack-o-HK.transformation-axioms
                mkarr-def
          by simp
        thus ⋀x. X.ide x ⟹
                    X-Prod.Map (Trn m) x = X-Prod.Map (Trn ?hk) x
          by simp
      qed
    qed
  qed
qed

lemma has-as-binary-product:
shows H.has-as-binary-product a b p₁ p₀
```

proof
  show *H.span* p_1 p_0 **and** *cod* p_1 = *a* **and***cod* p_0 = *b*
    by *auto*
  fix *x f g*
  assume *f*: «*f* : *x* → *a*» **and** *g*: «*g* : *x* → *b*»
  have *1*: ∃!*h*. p_1 ⋆ *h* = *f* ∧ p_0 ⋆ *h* = *g*
    using *f g universality* **by** *blast*
  show ∃!*h*. «*h* : *x* → *dom* p_1» ∧ p_1 ⋆ *h* = *f* ∧ p_0 ⋆ *h* = *g*
    using *1 f* **by** *blast*
qed

sublocale *binary-product hcomp a b* p_1 p_0
  using *has-as-binary-product*
  by *unfold-locales blast*

lemma *preserves-extensional-rts*:
assumes *extensional-rts* (*Dom a*) **and** *extensional-rts* (*Dom b*)
shows *extensional-rts Prod.resid*
  using *PROD.preserves-extensional-rts*
      Prod.preserves-extensional-rts assms(*1*−*2*)
  by *fastforce*

lemma *preserves-small-rts*:
assumes *small-rts* (*Dom a*) **and** *small-rts* (*Dom b*)
shows *small-rts Prod.resid*
  using *PROD.preserves-small-rts*
      Prod.preserves-reflects-small-rts assms(*1*−*2*)
  by *fastforce*

lemma *sta-tuple*:
assumes *H.span t u* **and** *cod t* = *a* **and** *cod u* = *b* **and** *sta t* **and** *sta u*
shows *sta* (*tuple t u*)
proof −
  have *0*: «*tuple t u* : *dom t* → *prod*»
    using *assms tuple-props*(*1*)
    by (*simp add*: *product-def*)
  have *src* (*tuple t u*) = *tuple t u*
    by (*metis* (*no-types*, *lifting*) *H.arr-iff-in-hom V.src-ide assms*(*1*−*5*)
        p_0-*simps*(*1*) p_1-*simps*(*1*) *src-hcomp tuple-props*(*6*) *universality*)
  thus *?thesis*
    using *0 V.ide-iff-src-self* **by** *auto*
qed

lemma *Map-*p_0:
shows *Map* p_0 = *PROD.*P_0 ∘ *Unpack*
  by *simp*

lemma *Map-*p_1:
shows *Map* p_1 = *PROD.*P_1 ∘ *Unpack*

by *simp*

lemma *Map-tuple*:
assumes «$t : x \to a$» **and** «$u : x \to b$»
shows $Map\ (tuple\ t\ u) = Pack \circ \langle\langle Map\ t,\ Map\ u\rangle\rangle$
proof −
have ∗: $Dom\ t = Dom\ x \wedge Cod\ t = Dom\ a \wedge$
$Dom\ u = Dom\ x \wedge Cod\ u = Dom\ b$
using *assms dom-char cod-char* **by** *auto*
interpret *X*: *extensional-rts* ‹*Dom x*›
using *assms*(*1*) *arr-char* [*of t*] *dom-char* **by** *auto*
interpret *XA*: *exponential-rts* ‹*Dom x*› ‹*Dom a*› **..**
interpret *XB*: *exponential-rts* ‹*Dom x*› ‹*Dom b*› **..**
interpret *AB*: *exponential-rts* ‹*Dom a*› ‹*Dom b*› **..**
interpret *aXb*: *extensional-rts* ‹*Dom prod*›
using *obj-prod obj-char arr-char* [*of prod*] **by** *blast*
interpret *X-aXb*: *exponential-rts* ‹*Dom x*› *Prod.resid* **..**
interpret *aXb-A*: *exponential-rts Prod.resid* ‹*Dom a*› **..**
interpret *aXb-B*: *exponential-rts Prod.resid* ‹*Dom b*› **..**
interpret $COMP_1$: *COMP* ‹*Dom x*› *Prod.resid* ‹*Dom a*› **..**
interpret $COMP_0$: *COMP* ‹*Dom x*› *Prod.resid* ‹*Dom b*› **..**
have *span*: *H.span t u*
using *assms* **by** *blast*
have *1*: *arr* (*tuple t u*)
using *assms span tuple-props* [*of t u*]
by (*elim H.in-homE*) *auto*
have *2*: *Dom* (*tuple t u*) = *Dom x*
by (*metis* (*no-types*, *lifting*) *1 Dom-dom H.in-homE*
assms(*1*−*2*) *tuple-props*(*2*))
have *3*: *Cod* (*tuple t u*) = *Prod.resid*
by (*metis* (*no-types*, *lifting*) *H.in-homE H-seq-char*
assms(*1*−*2*) p_1-*simps*(*4*) *tuple-props*(*4*))
have *4*: $COMP_1$.*BCxAB.arr* (*Trn* p_1, *Trn* (*tuple t u*))
by (*metis* (*no-types*, *lifting*) ∗ *1 2 3*
$COMP_1$.*extensional H.in-homE H-arr-char Trn-hcomp*
V.ide-implies-arr XA.not-arr-null assms(*1*−*2*)
p_1-*simps*(*1*) p_1-*simps*(*4*) p_1-*simps*(*5*) *tuple-props*(*4*))
interpret P_1: *simulation-as-transformation Prod.resid* ‹*Dom a*› P_1.*map* **..**
interpret P_0: *simulation-as-transformation Prod.resid* ‹*Dom b*› P_0.*map* **..**

interpret *T*: *transformation* ‹*Dom x*› ‹*Dom a*›
‹*XA.Dom* (*Trn t*)› ‹*XA.Cod* (*Trn t*)› ‹*XA.Map* (*Trn t*)›
using *assms*(*1*) ∗ *arr-char* [*of t*] *XA.arr-char* [*of Trn t*] **by** *auto*
interpret *U*: *transformation* ‹*Dom x*› ‹*Dom b*›
‹*XB.Dom* (*Trn u*)› ‹*XB.Cod* (*Trn u*)› ‹*XB.Map* (*Trn u*)›
using *assms*(*2*) ∗ *arr-char* [*of u*] *XB.arr-char* [*of Trn u*] *H.arrI*
by *auto*
have $Map\ t = PROD.P_1 \circ (Unpack \circ X\text{-}aXb.Map\ (Trn\ (tuple\ t\ u)))$
by (*metis* (*no-types*, *lifting*) *H.in-homE Map-hcomp Map-*p_1 *assms*(*1*−*2*)

```
      comp-assoc o-apply tuple-props(4))
  moreover
  have Map u = PROD.P₀ ∘ (Unpack ∘ X-aXb.Map (Trn (tuple t u)))
    by (metis (no-types, lifting) H.in-homE Map-hcomp Map-p₀ assms(1−2)
      comp-assoc comp-def tuple-props(5))
  moreover have Map t = PROD.P₁ ∘ ⟨⟨Map t, Map u⟩⟩
    using T.transformation-axioms U.transformation-axioms
      PROD.proj-tuple2 [of Dom x]
    by simp
  moreover have Map u = PROD.P₀ ∘ ⟨⟨Map t, Map u⟩⟩
    using T.transformation-axioms U.transformation-axioms
      PROD.proj-tuple2 [of Dom x]
    by simp
  ultimately
  have 5: Unpack ∘ X-aXb.Map (Trn (tuple t u)) = ⟨⟨Map t, Map u⟩⟩
    by (metis (no-types, lifting) PROD.tuple-proj
      Prod.Map'.simulation-axioms comp-assoc comp-pointwise-tuple)
  show ?thesis
  proof −
    have X-aXb.Map (Trn (tuple t u)) =
      Pack ∘ Unpack ∘ X-aXb.Map (Trn (tuple t u))
      using 1 2 3 arr-char [of tuple t u]
        X-aXb.arr-char [of Trn (tuple t u)]
        Prod.inv' comp-identity-transformation
      by fastforce
    also have ... = Pack ∘ (Unpack ∘ X-aXb.Map (Trn (tuple t u)))
      by auto
    also have ... = Pack ∘ ⟨⟨Map t, Map u⟩⟩
      using 5 by simp
    finally show ?thesis by simp
  qed
qed

end
```

Now we transfer to *rtscatx* just the definitions and facts we want from *product-in-rtscat*, generalized to all pairs of objects rather than a fixed pair.

```
context rtscatx
begin

  definition p₀
  where p₀ ≡ product-in-rtscat.p₀

  definition p₁
  where p₁ ≡ product-in-rtscat.p₁

  lemma sta-p₀:
  assumes obj a and obj b
  shows sta (p₀ a b)
```

by (*simp add*: *assms*(*1*−*2*) p_0*-def product-in-rtscat.*p_0*-simps*(*1*)
product-in-rtscat-axioms.intro product-in-rtscat-def
rtscatx.intro universe-axioms)

lemma *sta-*p_1:
assumes *obj a* **and** *obj b*
shows *sta* (p_1 *a b*)
by (*simp add*: *assms*(*1*−*2*) p_1*-def product-in-rtscat.*p_1*-simps*(*1*)
product-in-rtscat-axioms.intro product-in-rtscat-def
rtscatx.intro universe-axioms)

lemma *has-binary-products*:
assumes *obj a* **and** *obj b*
shows *H.has-as-binary-product a b* (p_1 *a b*) (p_0 *a b*)
by (*simp add*: *assms*(*1*−*2*) p_0*-def* p_1*-def*
product-in-rtscat.has-as-binary-product product-in-rtscat-axioms-def
product-in-rtscat-def rtscatx.intro universe-axioms)

interpretation *category-with-binary-products hcomp*
using *H.has-binary-products-def has-binary-products*
by *unfold-locales auto*

proposition *is-category-with-binary-products*:
shows *category-with-binary-products hcomp*
..

lemma *extends-to-elementary-category-with-binary-products*:
shows *elementary-category-with-binary-products hcomp* p_0 p_1
proof
fix *a b*
assume *a*: *obj a* **and** *b*: *obj b*
interpret *axb*: *product-in-rtscat arr-type a b*
using *a b* **by** *unfold-locales*
show *H.span* (p_1 *a b*) (p_0 *a b*) **and** *cod* (p_1 *a b*) = *a* **and** *cod* (p_0 *a b*) = *b*
unfolding p_0*-def* p_1*-def*
using *a b* **by** *auto*
next
fix *t u*
assume *tu*: *H.span t u*
interpret *axb*: *product-in-rtscat arr-type* ‹*cod t*› ‹*cod u*›
using *tu H.ide-cod*
by *unfold-locales auto*
show ∃!*l*. p_1 (*cod t*) (*cod u*) ⋆ *l* = *t* ∧
p_0 (*cod t*) (*cod u*) ⋆ *l* = *u*
unfolding p_0*-def* p_1*-def*
using *tu axb.universality* **by** *blast*
qed

interpretation *elementary-category-with-binary-products hcomp* p_0 p_1

using *extends-to-elementary-category-with-binary-products* **by** *blast*

notation p_0 ($\mathfrak{p}_0[\text{-}, \text{-}]$)
notation p_1 ($\mathfrak{p}_1[\text{-}, \text{-}]$)
notation *tuple* ($\langle\text{-}, \text{-}\rangle$)
notation *prod* (**infixr** $\otimes$ *51*)

lemma *prod-eq*:
assumes *obj a* **and** *obj b*
shows $a \otimes b$ = *product-in-rtscat.prod a b*
proof −
  interpret *axb*: *product-in-rtscat arr-type a b*
    using *assms* **by** *unfold-locales auto*
  show $a \otimes b$ = *axb.prod*
    using *assms*
    by (*metis* (*no-types*, *lifting*) *axb.*p_1*-simps*(*2*) p_1*-def pr-simps*(*5*))
qed

lemma *sta-tuple* [*simp*]:
assumes *H.span t u* **and** *sta t* **and** *sta u*
shows *sta* $\langle t, u\rangle$
proof −
  let *?a* = *cod t* **and** *?b* = *cod u*
  have *a*: *obj ?a* **and** *b*: *obj ?b*
    using *assms* **by** *auto*
  interpret *axb*: *product-in-rtscat arr-type ?a ?b*
    using *assms*
    by *unfold-locales auto*
  have $\langle t, u\rangle$ = *axb.tuple t u*
    using *assms*(*1*−*2*) *a b H.in-homE axb.tuple-props*(*4*−*5*)
        tuple-pr-arr p_0*-def* p_1*-def*
    by (*metis* (*no-types*, *lifting*))
  thus *?thesis*
    using *assms axb.sta-tuple* **by** *auto*
qed

lemma *sta-prod*:
assumes *sta t* **and** *sta u*
shows *sta* ($t \otimes u$)
proof −
  have *H.span* ($t \star p_1$ (*dom t*) (*dom u*)) ($u \star p_0$ (*dom t*) (*dom u*))
    using *H.seqI assms*(*1*−*2*) *pr-simps*(*1*,*4*) **by** *force*
  moreover have *V.ide* ($t \star p_1$ (*dom t*) (*dom u*))
    using *assms pr-in-hom sta-*p_1 *H-seq-char calculation* **by** *auto*
  moreover have *V.ide* ($u \star p_0$ (*dom t*) (*dom u*))
    using *assms pr-in-hom sta-*p_0 *H-seq-char calculation* **by** *auto*
  ultimately show *?thesis*

unfolding *prod-def*
using *assms sta-tuple* **by** *blast*
qed

definition *Pack* :: *'A arr* ⇒ *'A arr* ⇒ *'A* × *'A* ⇒ *'A*
where *Pack* ≡ *product-in-rtscat.Pack*

definition *Unpack* :: *'A arr* ⇒ *'A arr* ⇒ *'A* ⇒ *'A* × *'A*
where *Unpack* ≡ *product-in-rtscat.Unpack*

lemma *inverse-simulations-Pack-Unpack*:
assumes *obj a* **and** *obj b*
shows *inverse-simulations* (*Dom* (*a* ⊗ *b*)) (*product-rts.resid* (*Dom a*) (*Dom b*))
(*Pack a b*) (*Unpack a b*)
proof −
interpret *axb*: *product-in-rtscat arr-type a b*
using *assms* **by** *unfold-locales auto*
show *?thesis*
unfolding *Pack-def Unpack-def*
using p_0*-def* p_1*-def*
by (*metis* (*no-types*, *lifting*) *axb.Dom-prod*
axb.Prod.inverse-simulations-axioms axb.obj-a axb.obj-b
$axb.p_0$*-simps*(*2*) *pr-simps*(*2*))
qed

lemma *simulation-Pack*:
assumes *obj a* **and** *obj b*
shows *simulation* (*product-rts.resid* (*Dom a*) (*Dom b*)) (*Dom* (*a* ⊗ *b*))
(*Pack a b*)
using *assms inverse-simulations-Pack-Unpack* [*of a b*]
inverse-simulations-def
by *fast*

lemma *simulation-Unpack*:
assumes *obj a* **and** *obj b*
shows *simulation* (*Dom* (*a* ⊗ *b*)) (*product-rts.resid* (*Dom a*) (*Dom b*))
(*Unpack a b*)
using *assms inverse-simulations-Pack-Unpack* [*of a b*]
inverse-simulations-def
by *fast*

lemma *Pack-o-Unpack*:
assumes *obj a* **and** *obj b*
shows *Pack a b* ∘ *Unpack a b* = *I* (*Dom* (*a* ⊗ *b*))
proof −
interpret *PU*: *inverse-simulations*
‹*Dom* (*a* ⊗ *b*)› ‹*product-rts.resid* (*Dom a*) (*Dom b*)›
‹*Pack a b*› ‹*Unpack a b*›
using *assms inverse-simulations-Pack-Unpack* **by** *blast*

show *?thesis*
using *assms PU.inv'* **by** *simp*
qed

lemma *Unpack-o-Pack*:
assumes *obj a* **and** *obj b*
shows *Unpack a b ∘ Pack a b = I (product-rts.resid (Dom a) (Dom b))*
proof −
interpret *PU*: *inverse-simulations*
‹*Dom (a ⊗ b)*› ‹*product-rts.resid (Dom a) (Dom b)*›
‹*Pack a b*› ‹*Unpack a b*›
using *assms inverse-simulations-Pack-Unpack* **by** *blast*
show *?thesis*
using *assms PU.inv* **by** *simp*
qed

lemma *Pack-Unpack* [*simp*]:
assumes *obj a* **and** *obj b*
and *residuation.arr (Dom (a ⊗ b)) t*
shows *Pack a b (Unpack a b t) = t*
by (*meson assms*(*1*−*3*) *inverse-simulations.inv'-simp*
inverse-simulations-Pack-Unpack)

lemma *Unpack-Pack* [*simp*]:
assumes *obj a* **and** *obj b*
and *residuation.arr (product-rts.resid (Dom a) (Dom b)) t*
shows *Unpack a b (Pack a b t) = t*
by (*metis* (*no-types*, *lifting*) *Unpack-o-Pack assms*(*1*−*3*) *o-apply*)

lemma *src-tuple* [*simp*]:
assumes *H.span t u*
shows *src ⟨t, u⟩ = ⟨src t, src u⟩*
using *assms sta-p_0 sta-p_1 tuple-simps*(*1*) *src-hcomp H.seqI*
src-hcomp [*of* p_0 (*cod t*) (*cod u*) *tuple t u*]
src-hcomp [*of* p_1 (*cod t*) (*cod u*) *tuple t u*]
by (*intro tuple-eqI*) *auto*

lemma *trg-tuple* [*simp*]:
assumes *H.span t u*
shows *trg ⟨t, u⟩ = ⟨trg t, trg u⟩*
using *assms sta-p_0 sta-p_1 tuple-simps*(*1*) *src-hcomp H.seqI*
trg-hcomp [*of* p_0 (*cod t*) (*cod u*) *tuple t u*]
trg-hcomp [*of* p_1 (*cod t*) (*cod u*) *tuple t u*]
by (*intro tuple-eqI*) *auto*

lemma *Map-p_0*:
assumes *obj a* **and** *obj b*
shows *Map* $\mathfrak{p}_0[a, b]$ = *product-rts.*P_0 (*Dom a*) (*Dom b*) ∘ *Unpack a b*
proof −

```
  interpret axb: product-in-rtscat arr-type a b
    using assms by unfold-locales auto
  show ?thesis
    unfolding Unpack-def p₀-def
    using axb.Map-p₀ by blast
qed

lemma Map-p₁:
assumes obj a and obj b
shows Map 𝔭₁[a, b] = product-rts.P₁ (Dom a) (Dom b) ∘ Unpack a b
proof −
  interpret axb: product-in-rtscat arr-type a b
    using assms by unfold-locales auto
  show ?thesis
    unfolding Unpack-def p₁-def
    using axb.Map-p₁ by blast
qed

lemma Map-tuple:
assumes «t : x → a» and «u : x → b»
shows Map ⟨t, u⟩ = Pack a b ∘ ⟨⟨Map t, Map u⟩⟩
proof −
  interpret axb: product-in-rtscat arr-type a b
    using assms by unfold-locales auto
  show ?thesis
    unfolding Pack-def
    using assms axb.Map-tuple [of t x u] p₀-def p₁-def
    by (metis (no-types, lifting) H.in-homE axb.tuple-props(6) pr-tuple(1−2))
qed

lemma assoc-expansion:
assumes obj a and obj b and obj c
shows assoc a b c =
      ⟨𝔭₁[a, b] ⋆ 𝔭₁[a ⊗ b, c], ⟨𝔭₀[a, b] ⋆ 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩ ⟩
  using assms assoc-def by blast

end
```

4.3.3 Exponentials

In this section we show that the category $\mathbf{RTS}^\dagger$ has exponentials. The strategy is the same as for products: given objects b and c, construct the exponential RTS $[Dom\ b,\ Dom\ c]$, apply an injective map on the arrows to obtain an isomorphic RTS with arrow type $'A$, then let $exp\ b\ c$ be the object corresponding to this RTS. In order for the type-reducing injection to exist, we use the assumption that the type $'A$ admits exponentiation, but this is also where we use the assumption that the RTS's $Dom\ b$ and $Dom\ c$ are small, so that the exponential RTS $[Dom\ b,\ Dom\ c]$ is also small.

```
context rtscatx
begin

  definition inj-exp :: ('A, 'A) exponential-rts.arr ⇒ 'A
  where inj-exp ≡ λ exponential-rts.MkArr F G T ⇒
                          lifting.some-lift
                            (Some (pairing.some-pair
                                    (exponentiation.some-inj F,
                                     pairing.some-pair
                                       (exponentiation.some-inj G,
                                        exponentiation.some-inj T))))
                      | exponential-rts.Null ⇒ lifting.some-lift None

  lemma inj-inj-exp:
  assumes small-rts A and extensional-rts A
  and small-rts B and extensional-rts B
  shows inj-on inj-exp
      (Collect (residuation.arr (exponential-rts.resid A B)) ∪ {exponential-rts.Null})
  proof
    interpret A: small-rts A
      using assms by blast
    interpret A: extensional-rts A
      using assms by blast
    interpret B: small-rts B
      using assms by blast
    interpret B: extensional-rts B
      using assms by blast
    interpret AB: exponential-rts A B ..
    fix x y
    assume x: x ∈ Collect AB.arr ∪ {AB.Null}
    and y: y ∈ Collect AB.arr ∪ {AB.Null}
    assume eq: inj-exp x = inj-exp y
    show x = y
    proof (cases x; cases y)
      show ⟦x = AB.Null; y = AB.Null⟧ ⟹ x = y
        by blast
      show ⋀F' G' T'. ⟦x = AB.Null; y = AB.MkArr F' G' T'⟧ ⟹ x = y
        using x y eq inj-some-lift
        unfolding inj-exp-def inj-def
        by simp blast
      show ⋀F G T. ⟦x = AB.MkArr F G T; y = AB.Null⟧ ⟹ x = y
        using x y eq inj-some-lift
        unfolding inj-exp-def inj-def
        by simp blast
      fix F G T F' G' T'
      show ⟦x = AB.MkArr F G T; y = AB.MkArr F' G' T'⟧ ⟹ x = y
      proof −
        assume x-eq: x = AB.MkArr F G T and y-eq: y = AB.MkArr F' G' T'
        have some-lift
```

```
            (Some
              (some-pair
                (some-inj F, some-pair (some-inj G, some-inj T)))) =
          some-lift
            (Some
              (some-pair
                (some-inj F', some-pair (some-inj G', some-inj T'))))
        using eq x-eq y-eq
        unfolding inj-exp-def
        by simp
      hence Some
            (some-pair
              (some-inj F, some-pair (some-inj G, some-inj T))) =
          Some
            (some-pair
              (some-inj F', some-pair (some-inj G', some-inj T')))
        using inj-some-lift inj-def by metis
      hence some-pair (some-inj F, some-pair (some-inj G, some-inj T)) =
          some-pair (some-inj F', some-pair (some-inj G', some-inj T'))
        by simp
      hence some-inj F = some-inj F' ∧
          some-pair (some-inj G, some-inj T) =
          some-pair (some-inj G', some-inj T')
        using inj-some-pair inj-def [of some-pair] by blast
      hence 1: some-inj F = some-inj F' ∧ some-inj G = some-inj G' ∧
          some-inj T = some-inj T'
        using inj-some-pair inj-def [of some-pair] by blast
      have F = F' ∧ G = G' ∧ T = T'
      proof −
        have small-function F ∧ small-function F' ∧
            small-function G ∧ small-function G'
          using x-eq x y-eq y AB.arr-char small-function-simulation
              transformation-def
          by (metis A.small-rts-axioms AB.arr.simps(2)
            AB.arr-MkArr B.small-rts-axioms UnE mem-Collect-eq singletonD)
        moreover have small-function T ∧ small-function T'
          using assms x-eq x y-eq y AB.arr-char small-function-transformation
          by fast
        ultimately show ?thesis
          using 1 inj-some-inj inj-on-def [of some-inj] by auto
      qed
      thus x = y
        using x-eq y-eq by auto
    qed
  qed
qed

end
```

```
locale exponential-in-rtscat =
  rtscatx arr-type
for arr-type :: 'A itself
and b :: 'A rtscatx.arr
and c :: 'A rtscatx.arr +
assumes obj-b: obj b
and obj-c: obj c
begin

  sublocale elementary-category-with-binary-products hcomp p₀ p₁
    using extends-to-elementary-category-with-binary-products by blast

  notation hcomp  (infixr ⋆ 53)
  notation p₀     (𝔭₀[-, -])
  notation p₁     (𝔭₁[-, -])
  notation tuple  (⟨-, -⟩)
  notation prod   (infixr ⊗ 51)

  sublocale B: extensional-rts ‹Dom b›
    using obj-b bij-mkobj obj-char by blast
  sublocale B: small-rts ‹Dom b›
    using obj-b bij-mkobj obj-char by blast
  sublocale C: extensional-rts ‹Dom c›
    using obj-c bij-mkobj obj-char by blast
  sublocale C: small-rts ‹Dom c›
    using obj-c bij-mkobj obj-char by blast

  sublocale EXP: exponential-rts ‹Dom b› ‹Dom c› ..
  sublocale EXP: exponential-of-small-rts ‹Dom b› ‹Dom c› ..

  lemma small-function-Map:
  assumes EXP.arr t
  shows small-function (EXP.Dom t) and small-function (EXP.Cod t)
  and small-function (EXP.Map t)
    using assms small-function-simulation small-function-transformation
          transformation-def B.small-rts-axioms C.small-rts-axioms
          EXP.con-arr-src(2) EXP.con-char
    by metis+
```

Sublocale *Exp* refers to the isomorphic image of the RTS *EXP* under the type-reducing injective map *inj-exp*. These are connected by simulation *Func*, which maps *Exp* to *EXP*, and its inverse *Unfunc*, which maps *EXP* to *Exp*.

```
  sublocale Exp: inj-image-rts inj-exp EXP.resid
    using inj-inj-exp [of Dom b Dom c] EXP.null-char
          B.small-rts-axioms B.extensional-rts-axioms
          C.small-rts-axioms C.extensional-rts-axioms
    by unfold-locales argo
  sublocale Exp: extensional-rts Exp.resid
```

using *EXP.is-extensional-rts Exp.preserves-extensional-rts* **by** *blast*
sublocale *Exp*: *small-rts Exp.resid*
using *EXP.small-rts-axioms Exp.preserves-reflects-small-rts* **by** *blast*

lemma *is-extensional-rts*:
shows *extensional-rts Exp.resid*
..

lemma *is-small-rts*:
shows *small-rts Exp.resid*
..

abbreviation *Func* :: $'A \Rightarrow ('A, 'A)$ *EXP.arr*
where *Func* $\equiv$ *Exp.map*$'_{ext}$

abbreviation *Unfunc* :: $('A, 'A)$ *EXP.arr* $\Rightarrow 'A$
where *Unfunc* $\equiv$ *Exp.map*$_{ext}$

We define *exp* to be the object of the category $\mathbf{RTS}^\dagger$ having *Exp* as its underlying RTS.

definition *exp*
where *exp* $\equiv$ *mkobj Exp.resid*

lemma *obj-exp*:
shows *obj exp*
using *exp-def Exp.rts-axioms is-small-rts is-extensional-rts bij-mkobj* **by** *auto*

The fact that *Dom exp* and *Exp.resid* are equal, but not identical, poses a minor inconvenience for the moment.

lemma *Dom-exp* [*simp*]:
shows *Dom exp* = *Exp.resid*
unfolding *exp-def* **by** *fastforce*

sublocale *EXPxB*: *product-rts EXP.resid* ‹*Dom b*› **..**
sublocale *ExpxB*: *product-rts Exp.resid* ‹*Dom b*› **..**

sublocale *B*: *identity-simulation* ‹*Dom b*› **..**
sublocale *B*: *simulation-as-transformation* ‹*Dom b*› ‹*Dom b*› *B.map* **..**
sublocale *B*: *transformation-to-extensional-rts*
‹*Dom b*› ‹*Dom b*› *B.map B.map B.map* **..**
sublocale *UnfuncxB*: *product-simulation*
EXP.resid ‹*Dom b*› *Exp.resid* ‹*Dom b*› *Unfunc B.map* **..**
sublocale *FuncxB*: *product-simulation*
Exp.resid ‹*Dom b*› *EXP.resid* ‹*Dom b*› *Func B.map* **..**

sublocale *inverse-simulations EXPxB.resid ExpxB.resid*
FuncxB.map UnfuncxB.map
using *UnfuncxB.map-def FuncxB.map-def Exp.arr-char Exp.not-arr-null*
by *unfold-locales force+*

lemma *obj-expxb*:
shows *obj* (*exp* ⊗ *b*)
using *obj-exp obj-b* **by** *blast*

We now have a simulation *FuncxB-o-Unpack*, which refers to the result of composing the isomorphism *Unpack exp b* from *Dom expxb* to *ExpxB*, with the isomorphism *FuncxB* from *ExpxB* to *EXPxB*. This composite essentially "unpacks" the RTS *Dom expxb*, which underlies the product object *expxb*, to expose its construction as an application of the exponential RTS construction, followed by an application of the product RTS construction.

sublocale *FuncxB-o-Unpack*: *composite-simulation*
‹*Dom* (*exp* ⊗ *b*)› *ExpxB.resid EXPxB.resid*
‹*Unpack exp b*› *FuncxB.map*
proof −
have *product-rts.resid* (*Dom exp*) (*Dom b*) = *ExpxB.resid*
by *simp*
thus *composite-simulation* (*Dom* (*exp* ⊗ *b*)) *ExpxB.resid EXPxB.resid*
(*Unpack exp b*) *FuncxB.map*
using *FuncxB.simulation-axioms simulation-Unpack* [*of exp b*]
simulation-comp
by (*simp add*: *composite-simulation-def obj-b obj-exp*)
qed

We construct the evaluation map associated with *ExpxB* by composing the evaluation map *Eval.map* from *EXPxB* to *C*, derived from the exponential RTS construction, with the isomorphism *FuncxB-o-Unpack* from *Dom expxb.prod* to *EXPxB* and then obtain the corresponding arrow of the category.

sublocale *Eval*: *evaluation-map* ‹*Dom b*› ‹*Dom c*› **..**
sublocale *Eval*: *evaluation-map-between-extensional-rts* ‹*Dom b*› ‹*Dom c*› **..**
sublocale *Eval-o-FuncxB-o-Unpack*:
composite-simulation
‹*Dom* (*exp* ⊗ *b*)› *EXPxB.resid* ‹*Dom c*›
FuncxB-o-Unpack.map Eval.map
using *Eval.simulation-axioms FuncxB-o-Unpack.simulation-axioms*
composite-simulation-def
by *fastforce*

lemma *EvaloFuncxB-o-Unpack-is-simulation*:
shows *simulation* (*Dom* (*exp* ⊗ *b*)) (*Dom c*) *Eval-o-FuncxB-o-Unpack.map*
using *Eval-o-FuncxB-o-Unpack.simulation-axioms* **by** *blast*

definition *eval*
where *eval* ≡ *mksta* (*Dom* (*exp* ⊗ *b*)) (*Dom c*) *Eval-o-FuncxB-o-Unpack.map*

lemma *eval-simps* [*simp*]:
shows *sta eval* **and** *dom eval* = *exp* ⊗ *b* **and** *cod eval* = *c*

```
  and Dom eval = Dom (exp ⊗ b) and Cod eval = Dom c
  and Trn eval = exponential-rts.MkIde Eval-o-FuncxB-o-Unpack.map
  proof −
    show 1: sta eval
      unfolding eval-def
      using sta-mksta [of Dom (exp ⊗ b) Dom c Eval-o-FuncxB-o-Unpack.map]
        obj-b obj-c obj-exp obj-char arr-char Eval-o-FuncxB-o-Unpack.is-simulation
      by blast
    show 2: Dom eval = Dom (exp ⊗ b) and 3: Cod eval = Dom c
      unfolding eval-def mkarr-def by auto
    show Trn eval = exponential-rts.MkIde Eval-o-FuncxB-o-Unpack.map
      unfolding eval-def mkarr-def by auto
    have 4: (λa. if FuncxB-o-Unpack.F.A.arr a then a
                 else ResiduatedTransitionSystem.partial-magma.null
                        (Dom eval)) =
            I (Dom (exp ⊗ b))
      using 2 by presburger
    have 5: (λt. if C.arr t then t
                 else ResiduatedTransitionSystem.partial-magma.null
                        (Cod eval)) =
            I (Dom c)
      using 3 by presburger
    show dom eval = exp ⊗ b
      using 1 2 4 dom-char obj-char obj-expxb by auto
    show cod eval = c
      using 1 3 5 cod-char obj-char obj-c by auto
  qed

  lemma eval-in-hom [intro]:
  shows «eval : exp ⊗ b → c»
    using eval-simps by auto

  lemma Map-eval:
  shows Map eval = Eval.map ∘ (FuncxB.map ∘ Unpack exp b)
    unfolding eval-def mkarr-def by simp

end
```

Now we transfer the definitions and facts we want to *rtscatx*.

```
context rtscatx
begin

  interpretation elementary-category-with-binary-products hcomp p0 p1
    using extends-to-elementary-category-with-binary-products by blast

  notation prod    (infixr ⊗ 51)

  definition exp
  where exp b c ≡ exponential-in-rtscat.exp b c
```

lemma *obj-exp*:
assumes *obj b* **and** *obj c*
shows *obj* (*exp b c*)
proof −
  interpret *bc*: *exponential-in-rtscat arr-type b c*
    using *assms* **by** *unfold-locales auto*
  show *?thesis*
    unfolding *exp-def*
    using *bc.obj-exp* **by** *blast*
qed

definition *eval*
where *eval b c* ≡ *exponential-in-rtscat.eval b c*

lemma *eval-simps* [*simp*]:
assumes *obj b* **and** *obj c*
shows *sta* (*eval b c*)
and *dom* (*eval b c*) = *exp b c* ⊗ *b*
and *cod* (*eval b c*) = *c*
proof −
  interpret *bc*: *exponential-in-rtscat arr-type b c*
    using *assms* **by** *unfold-locales auto*
  show *sta* (*eval b c*)
  and *dom* (*eval b c*) = *exp b c* ⊗ *b*
  and *cod* (*eval b c*) = *c*
    unfolding *eval-def exp-def*
    using *bc.eval-simps* **by** *auto*
qed

lemma *eval-in-hom*$_{RCR}$ [*intro*]:
assumes *obj b* **and** *obj c*
shows «*eval b c* : *exp b c* ⊗ *b* → *c*»
  using *assms eval-simps* **by** *auto*

definition *Func* :: *'a arr* ⇒ *'a arr* ⇒ *'a* ⇒ (*'a*, *'a*) *exponential-rts.arr*
where *Func* ≡ *exponential-in-rtscat.Func*

definition *Unfunc* :: *'a arr* ⇒ *'a arr* ⇒ (*'a*, *'a*) *exponential-rts.arr* ⇒ *'a*
where *Unfunc* ≡ *exponential-in-rtscat.Unfunc*

lemma *inverse-simulations-Func-Unfunc*:
assumes *obj b* **and** *obj c*
shows *inverse-simulations*
        (*exponential-rts.resid* (*Dom b*) (*Dom c*)) (*Dom* (*exp b c*))
        (*Func b c*) (*Unfunc b c*)
proof −
  interpret *bc*: *exponential-in-rtscat arr-type b c*

using *assms* **by** *unfold-locales blast*
have *bc.Exp.resid* = *Dom* (*exp b c*)
using *assms exp-def bc.Dom-exp* **by** *metis*
thus *?thesis*
unfolding *Func-def Unfunc-def*
using *bc.Exp.inverse-simulations-axioms inverse-simulations-sym* **by** *auto*
qed

lemma *simulation-Func*:
assumes *obj b* **and** *obj c*
shows *simulation* (*Dom* (*exp b c*)) (*exponential-rts.resid* (*Dom b*) (*Dom c*))
(*Func b c*)
using *assms inverse-simulations-Func-Unfunc* [*of b c*]
inverse-simulations-def
by *fast*

lemma *simulation-Unfunc*:
assumes *obj b* **and** *obj c*
shows *simulation* (*exponential-rts.resid* (*Dom b*) (*Dom c*)) (*Dom* (*exp b c*))
(*Unfunc b c*)
using *assms inverse-simulations-Func-Unfunc* [*of b c*]
inverse-simulations-def
by *fast*

lemma *Func-o-Unfunc*:
assumes *obj b* **and** *obj c*
shows *Func b c* ∘ *Unfunc b c* = *I* (*exponential-rts.resid* (*Dom b*) (*Dom c*))
proof −
interpret *FU*: *inverse-simulations*
‹*exponential-rts.resid* (*Dom b*) (*Dom c*)› ‹*Dom* (*exp b c*)›
‹*Func b c*› ‹*Unfunc b c*›
using *assms inverse-simulations-Func-Unfunc* **by** *blast*
show *?thesis*
using *assms FU.inv′* **by** *simp*
qed

lemma *Unfunc-o-Func*:
assumes *obj b* **and** *obj c*
shows *Unfunc b c* ∘ *Func b c* = *I* (*Dom* (*exp b c*))
proof −
interpret *FU*: *inverse-simulations*
‹*exponential-rts.resid* (*Dom b*) (*Dom c*)› ‹*Dom* (*exp b c*)›
‹*Func b c*› ‹*Unfunc b c*›
using *assms inverse-simulations-Func-Unfunc* **by** *blast*
show *?thesis*
using *assms FU.inv* **by** *simp*
qed

lemma *Func-Unfunc* [*simp*]:

```
  assumes obj b and obj c
  and residuation.arr (exponential-rts.resid (Dom b) (Dom c)) t
  shows Func b c (Unfunc b c t) = t
    by (meson assms(1−3) inverse-simulations.inv'-simp
        inverse-simulations-Func-Unfunc)

  lemma Unfunc-Func [simp]:
  assumes obj b and obj c
  and residuation.arr (Dom (exp b c)) t
  shows Unfunc b c (Func b c t) = t
  proof −
    have Unfunc b c (Func b c t) = (Unfunc b c ∘ Func b c) t
      using assms by simp
    also have ... = t
      using assms Unfunc-o-Func [of b c] by auto
    finally show ?thesis by auto
  qed

  lemma Map-eval:
  assumes obj b and obj c
  shows Map (eval b c) =
        evaluation-map.map (Dom b) (Dom c) ∘
          (product-simulation.map
             (Dom (exp b c)) (Dom b) (Func b c) (I (Dom b)) ∘
             Unpack (exp b c) b)
  proof −
    interpret bc: exponential-in-rtscat arr-type b c
      using assms by unfold-locales blast
    have Map (eval b c) = bc.Eval.map ∘ (bc.FuncxB.map ∘ Unpack (exp b c) b)
      using assms bc.Map-eval comp-assoc exp-def local.eval-def
      by (simp add: exp-def eval-def)
    thus ?thesis
      unfolding Func-def
      by (simp add: exp-def)
  qed

end

locale currying-in-rtscat =
  exponential-in-rtscat arr-type b c
for arr-type :: 'A itself
and a :: 'A rtscatx.arr
and b :: 'A rtscatx.arr
and c :: 'A rtscatx.arr +
assumes obj-a: obj a
begin

  sublocale A: extensional-rts ‹Dom a›
    using obj-a obj-char arr-char by blast
```

sublocale *A*: *small-rts* ‹*Dom a*›
  using *obj-a obj-char arr-char* **by** *blast*
sublocale *B*: *extensional-rts* ‹*Dom b*›
  using *obj-b obj-char arr-char* **by** *blast*
sublocale *B*: *small-rts* ‹*Dom b*›
  using *obj-b obj-char arr-char* **by** *blast*

sublocale *AxB*: *product-of-extensional-rts* ‹*Dom a*› ‹*Dom b*› **..**
sublocale *A-Exp*: *exponential-rts* ‹*Dom a*› *Exp.resid* **..**

sublocale *aXb*: *extensional-rts* ‹*Dom* (*a* ⊗ *b*)›
  using *obj-a obj-b obj-char arr-char* **by** *blast*
sublocale *aXb*: *small-rts* ‹*Dom* (*a* ⊗ *b*)›
  using *obj-a obj-b obj-char arr-char* **by** *blast*
sublocale *expXb*: *exponential-rts Exp.resid* ‹*Dom b*› **..**
sublocale *aXb-C*: *exponential-rts* ‹*Dom* (*a* ⊗ *b*)› ‹*Dom c*› **..**

sublocale *Currying* ‹*Dom a*› ‹*Dom b*› ‹*Dom c*› **..**

definition *curry* :: *'A arr* ⇒ *'A arr*
where *curry f* = *mkarr* (*Dom a*) *Exp.resid*
                (*Unfunc* ∘ *Curry3* (*aXb-C.Dom* (*Trn f*) ∘ *Pack a b*))
                (*Unfunc* ∘ *Curry3* (*aXb-C.Cod* (*Trn f*) ∘ *Pack a b*))
                (*Unfunc* ∘ *Curry* (*aXb-C.Dom* (*Trn f*) ∘ *Pack a b*)
                              (*aXb-C.Cod* (*Trn f*) ∘ *Pack a b*)
                              (*aXb-C.Map* (*Trn f*) ∘ *Pack a b*))

lemma *curry-in-hom* [*intro*]:
assumes «*f* : *a* ⊗ *b* → *c*»
shows «*curry f* : *a* → *exp*»
proof −
  have *Dom*: *Dom f* = *Dom* (*a* ⊗ *b*) **and** *Cod*: *Cod f* = *Dom c*
    using *assms*
    apply (*metis* (*no-types*, *lifting*) *Dom-dom H.in-homE H-arr-char arr-char*)
    using *cod-char assms* **by** *fastforce*
  interpret *F*: *transformation* ‹*Dom* (*a* ⊗ *b*)› ‹*Dom c*›
              ‹*aXb-C.Dom* (*Trn f*)› ‹*aXb-C.Cod* (*Trn f*)› ‹*aXb-C.Map* (*Trn f*)›
    using *assms arr-char aXb-C.arr-char dom-char cod-char H.in-homE Dom*
    by *auto*

  let *?Src* = *Unfunc* ∘ *Curry3* (*aXb-C.Dom* (*Trn f*) ∘ *Pack a b*)
  let *?Trg* = *Unfunc* ∘ *Curry3* (*aXb-C.Cod* (*Trn f*) ∘ *Pack a b*)
  let *?Map* = *Unfunc* ∘ *Curry* (*aXb-C.Dom* (*Trn f*) ∘ *Pack a b*)
                          (*aXb-C.Cod* (*Trn f*) ∘ *Pack a b*)
                          (*aXb-C.Map* (*Trn f*) ∘ *Pack a b*)

  interpret *Src*: *simulation* ‹*Dom a*› *Exp.resid ?Src*
    using *obj-a obj-b simulation-Pack F.F.simulation-axioms*

```
      Exp.Map.simulation-axioms
  by (intro simulation-comp) auto
 interpret Trg: simulation ‹Dom a› Exp.resid ?Trg
  using obj-a obj-b simulation-Pack F.G.simulation-axioms
      Exp.Map.simulation-axioms
  by (intro simulation-comp) auto
 interpret Map: transformation ‹Dom a› Exp.resid ?Src ?Trg ?Map
 proof −
  interpret FoMap: transformation AxB.resid ‹Dom c›
                   ‹aXb-C.Dom (Trn f) ∘ Pack a b›
                   ‹aXb-C.Cod (Trn f) ∘ Pack a b›
                   ‹aXb-C.Map (Trn f) ∘ Pack a b›
    using obj-a obj-b F.transformation-axioms simulation-Pack
        transformation-whisker-right
          [of Dom (a ⊗ b) Dom c
              aXb-C.Dom (Trn f) aXb-C.Cod (Trn f)
              aXb-C.Map (Trn f) AxB.resid Pack a b]
        AxB.weakly-extensional-rts-axioms
    by blast
  have transformation (Dom a) EXP.resid
        (Eval.coext (Dom a) (aXb-C.Dom (Trn f) ∘ Pack a b))
        (Eval.coext (Dom a) (aXb-C.Cod (Trn f) ∘ Pack a b))
        (Curry (aXb-C.Dom (Trn f) ∘ Pack a b)
               (aXb-C.Cod (Trn f) ∘ Pack a b)
               (aXb-C.Map (Trn f) ∘ Pack a b))
    using Curry-preserves-transformations FoMap.transformation-axioms
    by blast
  thus transformation (Dom a) Exp.resid ?Src ?Trg ?Map
    using Exp.Map.simulation-axioms Exp.weakly-extensional-rts-axioms
        transformation-whisker-left
    by fastforce
 qed
 show ?thesis
  unfolding curry-def exp-def
  using obj-a obj-exp obj-char arr-char Map.transformation-axioms arr-mkarr
  by (intro H.in-homI) auto
qed

lemma curry-simps [simp]:
assumes «t : a ⊗ b → c»
shows arr (curry t) and dom (curry t) = a and cod (curry t) = exp
and Dom (curry t) = Dom a and Cod (curry t) = Exp.resid
and src (curry t) = curry (src t) and trg (curry t) = curry (trg t)
and Map (curry t) =
    (Unfunc ∘ Curry (aXb-C.Dom (Trn t) ∘ Pack a b)
                    (aXb-C.Cod (Trn t) ∘ Pack a b)
                    (aXb-C.Map (Trn t) ∘ Pack a b))
proof −
 let ?Src = Unfunc ∘ Curry3 (aXb-C.Dom (Trn t) ∘ Pack a b)
```

```
let ?Trg = Unfunc ∘ Curry3 (aXb-C.Cod (Trn t) ∘ Pack a b)
let ?Map = Unfunc ∘ Curry (aXb-C.Dom (Trn t) ∘ Pack a b)
                          (aXb-C.Cod (Trn t) ∘ Pack a b)
                          (aXb-C.Map (Trn t) ∘ Pack a b)
show arr (curry t) and dom (curry t) = a and cod (curry t) = exp
and Dom (curry t) = Dom a and Cod (curry t) = Exp.resid
and Map (curry t) = ?Map
  using assms obj-a curry-in-hom sta-mksta H.in-homE H-arr-char arr-char
        A.extensional-rts-axioms A.small-rts-axioms
        Exp.extensional-rts-axioms Exp.small-rts-axioms
  apply (auto simp add: curry-def mkarr-def)[6]
  by (metis (no-types, lifting) A-Exp.arr-MkArr
      Cod.simps(1) Dom.simps(1) Trn.simps(1))
have 1: transformation (Dom a) Exp.resid ?Src ?Trg ?Map
  using ‹arr (curry t)› A-Exp.src-char curry-def curry-in-hom
        arr-char mkarr-def
  by simp
show src (curry t) = curry (src t)
proof −
  have src (curry t) =
       MkArr (Dom a) (Exp.resid) (A-Exp.src (Trn (curry t)))
    unfolding src-char
    using assms ‹arr (curry t)› ‹Dom (curry t) = Dom a›
          ‹Cod (curry t) = Exp.resid›
    by simp
  also have ... = mksta (Dom a) Exp.resid ?Src
    using 1 A-Exp.src-char curry-def curry-in-hom arr-char
          aXb-C.arr-char mkarr-def
    by auto
  also have ... = curry (src t)
    unfolding src-char curry-def mkarr-def
    using assms
    apply auto[1]
     apply (metis (no-types, lifting) Dom-cod Dom-dom H.in-homE
       H-arr-char aXb-C.Dom.simps(1) aXb-C.src-simp)
    apply (metis (no-types, lifting) Cod-cod Cod-dom H.ide-char
       H.in-homE H-arr-char aXb-C.MkIde-Dom expXb.Cod.simps(1)
       ide-prod obj-a obj-b obj-c obj-char)
    by (metis (no-types, lifting) Cod-cod Cod-dom H.ide-char
       H.in-homE H-arr-char aXb-C.Map-src aXb-C.src-simp
       expXb.Cod.simps(1) expXb.Dom.simps(1) ide-prod
       obj-a obj-b obj-c obj-char)
  finally show ?thesis by blast
qed
show trg (curry t) = curry (trg t)
proof −
  have trg (curry t) =
       MkArr (Dom a) (Exp.resid) (A-Exp.trg (Trn (curry t)))
    unfolding trg-char
```

```
    using assms ‹arr (curry t)› ‹Dom (curry t) = Dom a›
          ‹Cod (curry t) = Exp.resid›
    by simp
  also have ... = mksta (Dom a) Exp.resid ?Trg
    using 1 A-Exp.trg-char curry-def curry-in-hom arr-char
          aXb-C.arr-char mkarr-def
    by auto
  also have ... = curry (trg t)
  proof −
    have arr t
      using assms by auto
    moreover have aXb-C.Dom (Trn (trg t)) = aXb-C.Map (Trn (trg t)) ∧
                  aXb-C.Cod (Trn (trg t)) = aXb-C.Map (Trn (trg t))
      by (metis (no-types, lifting) Cod-trg Dom-cod Dom-dom Dom-trg
          H.in-homE aXb-C.ide-char_ERTS assms calculation ide-trg staE)
    moreover have aXb-C.Cod (Trn t) =
                  aXb-C.Map
                    (residuation.trg
                       (exponential-rts.resid (Dom t) (Cod t)) (Trn t))
      by (metis (no-types, lifting) Dom-cod Dom-dom H.in-homE H-arr-char
          aXb-C.Map-trg arr-char assms)
    ultimately show ?thesis
      unfolding curry-def
      using assms trg-char by simp
  qed
  finally show ?thesis by blast
 qed
qed

lemma sta-curry:
assumes «f : a ⊗ b → c» and sta f
shows sta (curry f)
 using assms V.ide-iff-src-self [of curry f] by auto

definition uncurry :: 'A arr ⇒ 'A arr
where uncurry g = mkarr (Dom (a ⊗ b)) (Dom c)
                  (Uncurry (Func ∘ exponential-rts.Dom (Trn g)) ∘ Unpack a b)
                  (Uncurry (Func ∘ exponential-rts.Cod (Trn g)) ∘ Unpack a b)
                  (Uncurry (Func ∘ exponential-rts.Map (Trn g)) ∘ Unpack a b)

lemma uncurry-in-hom [intro]:
assumes «g : a → exp»
shows «uncurry g : a ⊗ b → c»
proof −
 interpret G: transformation ‹Dom a› Exp.resid
              ‹A-Exp.Dom (Trn g)› ‹A-Exp.Cod (Trn g)› ‹A-Exp.Map (Trn g)›
   using assms arr-char exp-def A-Exp.arr-char dom-char cod-char
   by (metis (no-types, lifting) H.in-homE H-arr-char
       mkobj-simps(2) obj-a obj-char)
```

```
  interpret Cmp'oG: transformation ‹Dom a› EXP.resid
                    ‹Func ∘ A-Exp.Dom (Trn g)›
                    ‹Func ∘ A-Exp.Cod (Trn g)›
                    ‹Func ∘ A-Exp.Map (Trn g)›
    using Exp.Map'.simulation-axioms G.transformation-axioms
          EXP.weakly-extensional-rts-axioms transformation-whisker-left
    by simp
  have transformation AxB.resid (Dom c)
         (Uncurry (Func ∘ A-Exp.Dom (Trn g)))
         (Uncurry (Func ∘ A-Exp.Cod (Trn g)))
         (Uncurry (Func ∘ A-Exp.Map (Trn g)))
    using Cmp'oG.transformation-axioms Uncurry-preserves-transformations
    by blast
  hence transformation (Dom (a ⊗ b)) (Dom c)
          (Uncurry (Func ∘ A-Exp.Dom (Trn g)) ∘ Unpack a b)
          (Uncurry (Func ∘ A-Exp.Cod (Trn g)) ∘ Unpack a b)
          (Uncurry (Func ∘ A-Exp.Map (Trn g)) ∘ Unpack a b)
    using obj-a obj-b simulation-Unpack [of a b]
          aXb.weakly-extensional-rts-axioms transformation-whisker-right
    by auto
  thus ?thesis
    unfolding uncurry-def
    using obj-c obj-char arr-char aXb.extensional-rts-axioms
          aXb.small-rts-axioms arr-mkarr
    apply (intro H.in-homI)
     apply auto[3]
    using obj-a obj-b by blast
qed

lemma uncurry-simps [simp]:
assumes «u : a → exp»
shows arr (uncurry u)
and dom (uncurry u) = a ⊗ b and cod (uncurry u) = c
and Dom (uncurry u) = Dom (a ⊗ b) and Cod (uncurry u) = Dom c
and Map (uncurry u) =
    Uncurry (Func ∘ exponential-rts.Map (Trn u)) ∘ Unpack a b
and src (uncurry u) = uncurry (src u)
and trg (uncurry u) = uncurry (trg u)
proof −
  show 0: arr (uncurry u)
  and dom (uncurry u) = a ⊗ b and cod (uncurry u) = c
    using assms uncurry-in-hom [of u] by auto
  show Dom (uncurry u) = Dom (a ⊗ b) and Cod (uncurry u) = Dom c
    using 0 ‹dom (uncurry u) = a ⊗ b› ‹cod (uncurry u) = c›
    by (metis Dom-dom Dom-cod)+
  show Map (uncurry u) =
       Uncurry (Func ∘ exponential-rts.Map (Trn u)) ∘ Unpack a b
    unfolding uncurry-def mkarr-def by simp
  have 1: transformation (Dom (a ⊗ b)) (Dom c)
```

```
          (Uncurry (Func ∘ aXb-C.Dom (Trn u)) ∘ Unpack a b)
          (Uncurry (Func ∘ aXb-C.Cod (Trn u)) ∘ Unpack a b)
          (Uncurry (Func ∘ aXb-C.Map (Trn u)) ∘ Unpack a b)
    using 0 A-Exp.src-char uncurry-def uncurry-in-hom arr-char mkarr-def
    by simp
  show src (uncurry u) = uncurry (src u)
  proof −
    have src (uncurry u) =
        MkArr (Dom (a ⊗ b)) (Dom c) (aXb-C.src (Trn (uncurry u)))
      unfolding src-char
      using assms 0 ‹Dom (uncurry u) = Dom (a ⊗ b)›
          ‹Cod (uncurry u) = Dom c›
      by simp
    also have ... = uncurry (src u)
      unfolding uncurry-def mkarr-def
      using assms 1 src-char aXb-C.src-char
      apply auto[1]
       apply (metis (no-types, lifting) A-Exp.src-simp Dom-cod Dom-dom
         Dom-exp H.in-homE arrE expXb.Dom.simps(1))
      apply (metis (no-types, lifting) A-Exp.src-simp Dom-cod Dom-dom
         Dom-exp H.in-homE arrE expXb.Cod.simps(1))
      by (metis (no-types, lifting) A-Exp.Map-src Dom-cod Dom-dom
         Dom-exp H.in-homE arrE)
    finally show ?thesis by blast
  qed
  show trg (uncurry u) = uncurry (trg u)
  proof −
    have trg (uncurry u) =
        MkArr (Dom (a ⊗ b)) (Dom c) (aXb-C.trg (Trn (uncurry u)))
      unfolding trg-char
      using assms ‹arr (uncurry u)› ‹Dom (uncurry u) = Dom (a ⊗ b)›
          ‹Cod (uncurry u) = Dom c›
      by simp
    also have ... = uncurry (trg u)
      unfolding uncurry-def mkarr-def trg-char
      using assms 1 trg-char aXb-C.trg-char
      apply auto[1]
       apply (metis (no-types, lifting) A-Exp.trg-char Dom-cod Dom-dom
         Dom-exp H.in-homE arrE expXb.Dom.simps(1))
      apply (metis (no-types, lifting) A-Exp.trg-char Dom-cod Dom-dom
         Dom-exp H.in-homE arrE expXb.Cod.simps(1))
      by (metis (no-types, lifting) A-Exp.Map-trg Dom-cod Dom-dom
         Dom-exp H.in-homE arrE)
    finally show ?thesis by blast
  qed
qed

lemma sta-uncurry:
assumes «g : a → exp» and sta g
```

```
shows sta (uncurry g)
  using assms V.ide-iff-src-self [of uncurry g] by auto

lemma uncurry-curry:
assumes obj a and obj b
and «t : a ⊗ b → c»
shows uncurry (curry t) = t
proof −
  have mkarr (Dom (a ⊗ b)) (Dom c)
         (Uncurry
           (Func ∘
             (Unfunc ∘
               Curry3 (aXb-C.Dom (Trn t) ∘ Pack a b))) ∘
            Unpack a b)
         (Uncurry
           (Func ∘
             (Unfunc ∘
                Curry3 (aXb-C.Cod (Trn t) ∘ Pack a b))) ∘
            Unpack a b)
         (Uncurry
           (Func ∘
             (Unfunc ∘
               Curry (aXb-C.Dom (Trn t) ∘ Pack a b)
                     (aXb-C.Cod (Trn t) ∘ Pack a b)
                     (aXb-C.Map (Trn t) ∘ Pack a b))) ∘
            Unpack a b) =
    t
  (is mkarr (Dom (a ⊗ b)) (Dom c) ?Src ?Trg ?Map = t)
  proof −
    interpret Dom: simulation ‹Dom (a ⊗ b)› ‹Dom c› ‹aXb-C.Dom (Trn t)›
      using assms(3) arr-char aXb-C.arr-char Dom-dom Dom-cod
            transformation-def
      by (metis (no-types, lifting) H.in-homE arr-coincidence)
    interpret Cod: simulation ‹Dom (a ⊗ b)› ‹Dom c› ‹aXb-C.Cod (Trn t)›
      using assms(3) arr-char aXb-C.arr-char Dom-dom Dom-cod
            transformation-def
      by (metis (no-types, lifting) H.in-homE arr-coincidence)
    interpret T: transformation ‹Dom (a ⊗ b)› ‹Dom c›
                 ‹aXb-C.Dom (Trn t)› ‹aXb-C.Cod (Trn t)›
                 ‹aXb-C.Map (Trn t)›
      using assms(3) arr-char aXb-C.arr-char Dom-dom Dom-cod
      by (metis (no-types, lifting) H.in-homE arr-coincidence)
    interpret Dom-o-Pack: composite-simulation
                            AxB.resid ‹Dom (a ⊗ b)› ‹Dom c›
                            ‹Pack a b› ‹aXb-C.Dom (Trn t)›
      by intro-locales
         (simp add: obj-a obj-b simulation.axioms(3) simulation-Pack)
    interpret Dom-o-Pack: simulation-as-transformation
                            AxB.resid ‹Dom c› Dom-o-Pack.map
```

```
  ..
interpret Cod-o-Pack: composite-simulation
                      AxB.resid ‹Dom (a ⊗ b)› ‹Dom c›
                      ‹Pack a b› ‹aXb-C.Cod (Trn t)›
  ..
interpret Cod-o-Pack: simulation-as-transformation
                       AxB.resid ‹Dom c› Cod-o-Pack.map
  ..
interpret T-o-Pack: transformation AxB.resid ‹Dom c›
                    Dom-o-Pack.map Cod-o-Pack.map
                    ‹aXb-C.Map (Trn t) ∘ Pack a b›
  using obj-a obj-b T.transformation-axioms simulation-Pack
        AxB.weakly-extensional-rts-axioms
        transformation-whisker-right
          [of Dom (a ⊗ b) Dom c aXb-C.Dom (Trn t)
              aXb-C.Cod (Trn t) aXb-C.Map (Trn t)
              AxB.resid Pack a b]
  by auto
interpret Curry-T-o-Pack: transformation ‹Dom a› EXP.resid
                          ‹Curry3 Dom-o-Pack.map›
                          ‹Curry3 Cod-o-Pack.map›
                          ‹Curry
                             Dom-o-Pack.map
                             Cod-o-Pack.map
                             (aXb-C.Map (Trn t) ∘ Pack a b)›
  using T-o-Pack.transformation-axioms Curry-preserves-transformations
  by blast
have ?Src = aXb-C.Dom (Trn t)
proof −
  have ?Src =
        Uncurry
          ((Func ∘ Unfunc) ∘
                Curry3 (aXb-C.Dom (Trn t) ∘ Pack a b)) ∘
             Unpack a b
    using comp-assoc by metis
  also have ... =
             Uncurry
               (Curry3 (aXb-C.Dom (Trn t) ∘ Pack a b)) ∘
                 Unpack a b
    using Exp.inv Curry-T-o-Pack.transformation-axioms
          comp-identity-simulation
            [of Dom a EXP.resid Curry3 Dom-o-Pack.map]
    by (auto simp add: transformation-def)
  also have ... = aXb-C.Dom (Trn t) ∘ (Pack a b ∘ Unpack a b)
    using Dom-o-Pack.transformation-axioms Uncurry-Curry by auto
  also have ... = aXb-C.Dom (Trn t) ∘ I (Dom (a ⊗ b))
    using assms Pack-o-Unpack by simp
  also have ... = aXb-C.Dom (Trn t)
    using assms Dom.simulation-axioms comp-simulation-identity
```

```
    by auto
  finally show ?thesis by auto
qed
moreover
have ?Trg = aXb-C.Cod (Trn t)
proof −
  have ?Trg =
       Uncurry
        ((Func ∘ Exp.map_ext) ∘
             Curry3 (aXb-C.Cod (Trn t) ∘ Pack a b)) ∘
          Unpack a b
    using comp-assoc by metis
  also have ... =
       Uncurry
        (Curry3 (aXb-C.Cod (Trn t) ∘ Pack a b)) ∘
          Unpack a b
    using Exp.inv Curry-T-o-Pack.transformation-axioms
          comp-identity-simulation
          [of Dom a EXP.resid Curry Cod-o-Pack.map
              Cod-o-Pack.map Cod-o-Pack.map]
    by (auto simp add: transformation-def)
  also have ... = aXb-C.Cod (Trn t) ∘ (Pack a b ∘ Unpack a b)
    using Cod-o-Pack.transformation-axioms Uncurry-Curry by auto
  also have ... = aXb-C.Cod (Trn t) ∘ I (Dom (a ⊗ b))
    using assms Pack-o-Unpack by simp
  also have ... = aXb-C.Cod (Trn t)
    using assms Cod.simulation-axioms comp-simulation-identity
    by auto
  finally show ?thesis by auto
qed
moreover
have ?Map = aXb-C.Map (Trn t)
proof −
  have ?Map =
       Uncurry
        ((Func ∘ Unfunc) ∘
             Curry (aXb-C.Dom (Trn t) ∘ Pack a b)
                   (aXb-C.Cod (Trn t) ∘ Pack a b)
                   (aXb-C.Map (Trn t) ∘ Pack a b)) ∘
          Unpack a b
    using comp-assoc by metis
  also have ... =
       Uncurry
        (Curry (aXb-C.Dom (Trn t) ∘ Pack a b)
               (aXb-C.Cod (Trn t) ∘ Pack a b)
               (aXb-C.Map (Trn t) ∘ Pack a b)) ∘
          Unpack a b
    using Exp.inv Curry-T-o-Pack.transformation-axioms
          comp-identity-transformation [of Dom a EXP.resid]
```

```
      by (auto simp add: transformation-def)
    also have ... = aXb-C.Map (Trn t) ∘ (Pack a b ∘ Unpack a b)
      using T-o-Pack.transformation-axioms Uncurry-Curry by auto
    also have ... = aXb-C.Map (Trn t) ∘ I (Dom (a ⊗ b))
      using assms Pack-o-Unpack by simp
    also have ... = aXb-C.Map (Trn t)
      using assms T.transformation-axioms comp-transformation-identity
      by blast
    finally show ?thesis by auto
  qed
  ultimately have mkarr (Dom (a ⊗ b)) (Dom c) ?Src ?Trg ?Map =
                  mkarr (Dom (a ⊗ b)) (Dom c)
                    (aXb-C.Dom (Trn t)) (aXb-C.Cod (Trn t))
                    (aXb-C.Map (Trn t))
    by simp
  also have ... = t
    by (metis (no-types, lifting) Dom-cod Dom-dom H.in-homE
        H-arr-char mkarr-def MkArr-Trn aXb-C.arrE aXb-C.null-char
        arr-char assms(3) expXb.MkArr-Map)
  finally show ?thesis
    using curry-def uncurry-def by simp
 qed
 thus ?thesis
   using assms curry-def uncurry-def mkarr-def by simp
qed

lemma curry-uncurry:
assumes «u : a → exp»
shows curry (uncurry u) = u
proof −
 have mkarr (Dom a) Exp.resid
        (Exp.map_ext ∘
          Curry3
           ((Uncurry (Func ∘ A-Exp.Dom (Trn u)) ∘ Unpack a b) ∘ Pack a b))
        (Exp.map_ext ∘
          Curry3
           ((Uncurry (Func ∘ A-Exp.Cod (Trn u)) ∘ Unpack a b) ∘ Pack a b))
        (Exp.map_ext ∘
          Curry
           ((Uncurry (Func ∘ A-Exp.Dom (Trn u)) ∘ Unpack a b) ∘ Pack a b)
            ((Uncurry (Func ∘ A-Exp.Cod (Trn u)) ∘ Unpack a b) ∘ Pack a b)
           ((Uncurry (Func ∘ A-Exp.Map (Trn u)) ∘ Unpack a b) ∘ Pack a b))
         = u
  (is ?LHS = u)
 proof −
   interpret Dom: simulation ‹Dom a› Exp.resid ‹A-Exp.Dom (Trn u)›
     using assms(1) arr-char A-Exp.arr-char transformation-def
     by (metis Dom-cod Dom-dom Dom-exp H.in-homE arr-coincidence)
   interpret Cod: simulation ‹Dom a› Exp.resid ‹A-Exp.Cod (Trn u)›
```

using *assms*(*1*) *arr-char A-Exp.arr-char transformation-def*
by (*metis* (*mono-tags*, *lifting*) *Dom-cod Dom-dom Dom-exp*
H.in-homE arr-coincidence)
interpret *U*: *transformation* ‹*Dom a*› *Exp.resid*
‹*A-Exp.Dom* (*Trn u*)› ‹*A-Exp.Cod* (*Trn u*)›
‹*A-Exp.Map* (*Trn u*)›
using *assms*(*1*) *arr-char A-Exp.arr-char H.in-homE dom-char cod-char*
by (*metis* (*no-types*, *lifting*) *Dom-cod Dom-dom Dom-exp arr-coincidence*)
interpret *FuncoDom*: *composite-simulation*
‹*Dom a*› *Exp.resid EXP.resid*
‹*A-Exp.Dom* (*Trn u*)› *Func*
..
interpret *FuncoDom*: *simulation-as-transformation*
‹*Dom a*› *EXP.resid* ‹*Func* ∘ *A-Exp.Dom* (*Trn u*)›
..
interpret *FuncoCod*: *composite-simulation*
‹*Dom a*› *Exp.resid EXP.resid*
‹*A-Exp.Cod* (*Trn u*)› *Func*
..
interpret *FuncoCod*: *simulation-as-transformation*
‹*Dom a*› *EXP.resid* ‹*Func* ∘ *A-Exp.Cod* (*Trn u*)›
..
interpret *FuncoU*: *transformation* ‹*Dom a*› *EXP.resid*
FuncoDom.map FuncoCod.map
‹*Func* ∘ *A-Exp.Map* (*Trn u*)›
using *U.transformation-axioms Exp.Map'.simulation-axioms*
EXP.weakly-extensional-rts-axioms transformation-whisker-left
by *blast*
have *1*: *transformation AxB.resid* (*Dom c*)
(*Uncurry FuncoDom.map*) (*Uncurry FuncoCod.map*)
(*Uncurry* (*Func* ∘ *aXb-C.Map* (*Trn u*)))
using *Uncurry-preserves-transformations FuncoU.transformation-axioms*
by *simp*
have *2*: (*Uncurry* (*Func* ∘ *A-Exp.Dom* (*Trn u*)) ∘ *Unpack a b*) ∘
Pack a b =
Uncurry (*Func* ∘ *A-Exp.Dom* (*Trn u*))
proof −
have (*Uncurry* (*Func* ∘ *A-Exp.Dom* (*Trn u*)) ∘ *Unpack a b*) ∘ *Pack a b* =
Uncurry (*Func* ∘ *A-Exp.Dom* (*Trn u*)) ∘ (*Unpack a b* ∘ *Pack a b*)
using *comp-assoc* **by** *metis*
also have ... = *Uncurry* (*Func* ∘ *A-Exp.Dom* (*Trn u*)) ∘ *I AxB.resid*
using *obj-a obj-b Unpack-o-Pack* **by** *auto*
also have ... = *Uncurry* (*Func* ∘ *A-Exp.Dom* (*Trn u*))
using *1 transformation-def comp-simulation-identity* **by** *blast*
finally show *?thesis* **by** *simp*
qed
have *3*: (*Uncurry* (*Func* ∘ *A-Exp.Cod* (*Trn u*)) ∘ *Unpack a b*) ∘ *Pack a b* =
Uncurry (*Func* ∘ *A-Exp.Cod* (*Trn u*))
proof −

```
  have (Uncurry (Func ∘ A-Exp.Cod (Trn u)) ∘ Unpack a b) ∘ Pack a b =
        Uncurry (Func ∘ A-Exp.Cod (Trn u)) ∘ (Unpack a b ∘ Pack a b)
    using comp-assoc by metis
  also have ... = Uncurry (Func ∘ A-Exp.Cod (Trn u)) ∘ I AxB.resid
    using obj-a obj-b Unpack-o-Pack by auto
  also have ... = Uncurry (Func ∘ A-Exp.Cod (Trn u))
    using 1 transformation-def comp-simulation-identity by blast
  finally show ?thesis by simp
qed
have 4: (Uncurry (Func ∘ A-Exp.Map (Trn u)) ∘ Unpack a b) ∘ Pack a b =
         Uncurry (Func ∘ A-Exp.Map (Trn u))
proof −
  have (Uncurry (Func ∘ A-Exp.Map (Trn u)) ∘ Unpack a b) ∘ Pack a b =
        Uncurry (Func ∘ A-Exp.Map (Trn u)) ∘ (Unpack a b ∘ Pack a b)
    using comp-assoc by metis
  also have ... = Uncurry (Func ∘ A-Exp.Map (Trn u)) ∘ I AxB.resid
    using obj-a obj-b Unpack-o-Pack by auto
  also have ... = Uncurry (Func ∘ A-Exp.Map (Trn u))
    using 1 transformation-def comp-transformation-identity by blast
  finally show ?thesis by simp
qed
have ?LHS = mkarr (Dom a) Exp.resid
              (Exp.map_ext ∘ Exp.map'_ext ∘ A-Exp.Dom (Trn u))
              (Exp.map_ext ∘ Exp.map'_ext ∘ A-Exp.Cod (Trn u))
              (Exp.map_ext ∘ Exp.map'_ext ∘ A-Exp.Map (Trn u))
  using 2 3 4 FuncoDom.transformation-axioms
        FuncoCod.transformation-axioms FuncoU.transformation-axioms
        Curry-Uncurry mkarr-def
  by auto
also have ... = mkarr (Dom a) Exp.resid
                  (A-Exp.Dom (Trn u)) (A-Exp.Cod (Trn u))
                  (A-Exp.Map (Trn u))
  using Dom.simulation-axioms Cod.simulation-axioms
        U.transformation-axioms comp-identity-transformation
        comp-identity-simulation [of Dom a Exp.resid]
        Exp.inv' mkarr-def
  by simp
also have ... = u
proof −
  have Exp.resid = Cod u
    using assms Dom-exp cod-char
    by (metis (no-types, lifting) Dom-cod H.in-homE
        H-arr-char arr-char)
  moreover have Trn u =
                  A-Exp.MkArr
                    (A-Exp.Dom (Trn u)) (A-Exp.Cod (Trn u))
                    (A-Exp.Map (Trn u))
    using assms arr-char [of u] A-Exp.MkArr-Map
    apply auto[1]
```

```
      by (metis (no-types, lifting) A.weakly-extensional-rts-axioms
          Dom-dom H.in-homE exponential-rts.arr-char
          exponential-rts.intro)
    ultimately show ?thesis
      using assms U.transformation-axioms null-char arr-char
            A-Exp.arr-char A-Exp.null-char dom-char
            cod-char mkarr-def
      by (intro arr-eqI) auto
  qed
  finally show ?thesis by auto
 qed
 thus ?thesis
  unfolding curry-def uncurry-def mkarr-def
  by simp
qed
```

We are not yet quite where we want to go, because to establish the naturality of the curry/uncurry bijection we have to show how uncurry relates to evaluation.

```
lemma uncurry-expansion:
assumes «u : a → exp»
shows uncurry u = eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩
proof (intro arr-eqI)
  interpret AxB: identity-simulation AxB.resid ..
  interpret P₀: simulation-as-transformation AxB.resid ‹Dom b› AxB.P₀ ..
  interpret P₁: simulation-as-transformation AxB.resid ‹Dom a› AxB.P₁ ..
  interpret U: transformation ‹Dom a› ‹Dom exp›
                ‹A-Exp.Dom (Trn u)› ‹A-Exp.Cod (Trn u)› ‹A-Exp.Map (Trn u)›
    using assms Dom-dom Dom-cod [of u] arr-char A-Exp.arr-char
    by (metis (no-types, lifting) Dom-exp H.in-homE arr-coincidence)
  have a: obj a
    using assms H.ide-dom by blast
  have src-u: «src u : a →_sta exp»
    using assms by fastforce
  have 0: «eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩ : a ⊗ b → c»
    using assms obj-a obj-b by auto
  have 00: «eval ⋆ ⟨src u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩ : a ⊗ b → c»
    using src-u obj-a obj-b by auto
  have Dom: Dom (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩) = Dom (a ⊗ b)
    using 0 Dom-dom [of eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩] by auto
  have Cod: Cod (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩) = Dom c
    using 0 Cod-dom [of eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩]
    by (metis (no-types, lifting) Dom-cod H.in-homE H-arr-char arr-char)
  show uncurry u ≠ Null
    using assms uncurry-simps(1) V.not-arr-null null-char by metis
  show eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩ ≠ Null
    using assms eval-in-hom null-char tuple-in-hom pr-in-hom
    by (metis (no-types, lifting) H.comp-in-homI H.seqI' V.not-arr-null
        arr-coincidence obj-a obj-b)
```

```
show 1: Dom (uncurry u) = Dom (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)
  by (simp add: Dom assms)
show 2: Cod (uncurry u) = Cod (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)
  by (simp add: Cod assms)
show Trn (uncurry u) = Trn (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)
proof (intro aXb-C.arr-eqI)
  show aXb-C.arr (Trn (uncurry u))
    using assms arr-char [of uncurry u] by auto
  show aXb-C.arr (Trn (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩))
    using assms(1) 0 1 2 arr-char [of eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩]
          Dom-dom Dom-cod
    by auto
  show aXb-C.Dom (Trn (uncurry u)) =
       aXb-C.Dom (Trn (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩))
  proof −
    have 4: arr (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)
      using 0 by blast
    have 5: sta ⟨src u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩
      using assms 00 sta-tuple sta-p₀ sta-p₁
      by (metis (no-types, lifting) H.dom-comp H.in-homE
          H.seqI cod-pr1 obj-a obj-b pr-simps(1−2,4−5)
          src-u sta-hcomp)
    have aXb-C.Dom (Trn (uncurry u)) =
          Eval.map ∘
            product-simulation.map (Dom a) (Dom b)
              (Func ∘ Map (src u)) B.map ∘
            Unpack a b
    proof −
      have aXb-C.Dom (Trn (uncurry u)) =
           aXb-C.Map (aXb-C.src (Trn (uncurry u)))
        by (simp add: ‹aXb-C.arr (Trn (uncurry u))›)
      also have ... = Map (src (uncurry u))
        using assms(1) src-char by force
      also have ... = Map (uncurry (src u))
        using assms(1) uncurry-simps by simp
      also have ... = Uncurry (Func ∘ Map (src u)) ∘ Unpack a b
        unfolding uncurry-def mkarr-def by simp
      also have ... = Eval.map ∘
                      product-simulation.map (Dom a) (Dom b)
                        (Func ∘ Map (src u)) B.map ∘
                        Unpack a b
      proof −
       have simulation (Dom a) EXP.resid (Func ∘ A-Exp.Map (Trn (src u)))
         using assms Exp.Map′.simulation-axioms sta-char
               A-Exp.ide-char_ERTS simulation-comp
         by (metis (no-types, lifting) Cod-src Dom-cod Dom-dom
             Dom-exp Dom-src H.in-homE V.ide-src arr-coincidence)
       thus ?thesis
         using Eval.Uncurry-simulation-expansion
```

```
          [of Dom a Exp.map'_ext ∘ A-Exp.Map (Trn (src u))]
        A.weakly-extensional-rts-axioms
    by auto
  qed
  finally show ?thesis by simp
qed
moreover
have aXb-C.Dom (Trn (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)) =
     Eval.map ∘
       product-simulation.map (Dom a) (Dom b)
         (Func ∘ Map (src u)) B.map ∘
       Unpack a b
proof −
  have aXb-C.Dom (Trn (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)) =
       aXb-C.Map (aXb-C.src (Trn (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)))
    using assms(1) 0 4 Dom Cod arr-char by auto
  also have ... = aXb-C.Map (Trn (src (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)))
    using 4 src-char Cod Dom Trn.simps(1) by presburger
  also have ... = Map (eval ⋆ ⟨src u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)
  proof −
    have H.seq eval ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩
      by (simp add: 4)
    moreover have H.span (u ⋆ 𝔭₁[a, b]) 𝔭₀[a, b]
      by (metis (no-types, lifting) H.not-arr-null H-seq-char
          arr-coincidence calculation tuple-ext)
    ultimately show ?thesis
      using obj-a obj-b sta-p₀ sta-p₁ by auto
  qed
  also have ... = Map eval ∘ Map ⟨src u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩
    using 00 Map-hcomp by blast
  also have ... = Map eval ∘
                  (Pack exp b ∘
                    ⟨⟨Map (src u ⋆ 𝔭₁[a, b]), AxB.P₀ ∘ Unpack a b⟩⟩)
    using assms(1) obj-a obj-b Map-p₀
          Map-tuple [of src u ⋆ 𝔭₁[a, b] a ⊗ b exp 𝔭₀[a, b] b]
    by (metis (no-types, lifting) H.comp-in-homI pr-in-hom(1−2) src-u)
  also have ... = Map eval ∘
                  (Pack exp b ∘
                    ⟨⟨Map (src u) ∘ (AxB.P₁ ∘ Unpack a b),
                      AxB.P₀ ∘ Unpack a b⟩⟩)
    using H.seqI Map-hcomp Map-p₁ assms obj-b pr-simps(4) by auto
  also have ... = Map eval ∘
                  Pack exp b ∘
                    ⟨⟨Map (src u) ∘ (AxB.P₁ ∘ Unpack a b),
                      AxB.P₀ ∘ Unpack a b⟩⟩
    using comp-assoc by metis
  also have ... = (Eval.map ∘
                   (FuncxB.map ∘
                     (Unpack exp b ∘ Pack exp b)) ∘
```

$\langle\langle Map\ (src\ u) \circ AxB.P_1,\ AxB.P_0\rangle\rangle) \circ$
$Unpack\ a\ b$
using *Map-eval*
comp-pointwise-tuple
[*of Map* (*src u*) ∘ $AxB.P_1$ $AxB.P_0$ *Unpack a b*]
by (*simp add*: *comp-assoc*)
also have ... = (*Eval.map* ∘
FuncxB.map ∘
$\langle\langle Map\ (src\ u) \circ AxB.P_1,\ AxB.P_0\rangle\rangle) \circ$
Unpack a b
using *obj-b obj-exp Unpack-o-Pack Dom-exp*
FuncxB.simulation-axioms comp-simulation-identity
[*of ExpxB.resid EXPxB.resid FuncxB.map*]
by *presburger*
also have ... = *Eval.map* ∘
(*FuncxB.map* ∘
$\langle\langle Map\ (src\ u) \circ AxB.P_1,\ AxB.P_0\rangle\rangle) \circ$
Unpack a b
by *auto*
also have ... = *Eval.map* ∘
$\langle\langle Func \circ Map\ (src\ u) \circ AxB.P_1,$
$B.map \circ AxB.P_0\rangle\rangle \circ$
Unpack a b
proof −
have *1*: *Src u* = *Map* (*src u*)
using *assms Map-simps*(*3*) **by** *fastforce*
interpret $src\text{-}uoP_1$: *simulation AxB.resid Exp.resid*
‹*Map* (*src u*) ∘ $AxB.P_1$›
using *1* $AxB.P_1.simulation\text{-}axioms$ $U.F.simulation\text{-}axioms$
simulation-comp
by *auto*
interpret $src\text{-}uoP_1$: *simulation-as-transformation*
AxB.resid Exp.resid
‹*Map* (*src u*) ∘ $AxB.P_1$›
..
show *?thesis*
using $src\text{-}uoP_1.transformation\text{-}axioms$ *B.simulation-axioms*
$Exp.Map'.simulation\text{-}axioms$ $P_0.transformation\text{-}axioms$
comp-product-simulation-tuple2
[*of Exp.resid EXP.resid Func Dom b Dom b B.map*
AxB.resid - - Map (*src u*) ∘ $AxB.P_1$ - - $AxB.P_0$]
by (*simp add*: *comp-assoc*)
qed
also have ... = *Eval.map* ∘
(*product-simulation.map* (*Dom a*) (*Dom b*)
(*Func* ∘ *Map* (*src u*)) *B.map* ∘
$\langle\langle AxB.P_1,\ AxB.P_0\rangle\rangle) \circ$
Unpack a b
proof −

```
    have simulation (Dom a) EXP.resid (Func ∘ Map (src u))
      using Exp.Map'.simulation-axioms U.F.simulation-axioms
            simulation-comp Dom-exp H.arrI Map-simps(3) assms
      by auto
    thus ?thesis
      using B.simulation-axioms P₀.transformation-axioms
            P₁.transformation-axioms
            comp-product-simulation-tuple2
              [of Dom a EXP.resid Func ∘ Map (src u)
                  Dom b Dom b B.map AxB.resid
                  AxB.P₁ AxB.P₁ AxB.P₁ AxB.P₀ AxB.P₀ AxB.P₀]
            simulation-as-transformation.intro
            AxB.weakly-extensional-rts-axioms
            A.weakly-extensional-rts-axioms
            B.weakly-extensional-rts-axioms
      by simp
  qed
  also have ... = Eval.map ∘
                  product-simulation.map (Dom a) (Dom b)
                    (Func ∘ Map (src u)) B.map ∘
                    (⟨⟨AxB.P₁, AxB.P₀⟩⟩ ∘ Unpack a b)
    by auto
  also have ... = Eval.map ∘
                  (product-simulation.map (Dom a) (Dom b)
                     (Func ∘ Map (src u)) B.map) ∘
                  Unpack a b
  proof −
    interpret AxB: identity-simulation AxB.resid ..
    have ⟨⟨AxB.P₁, AxB.P₀⟩⟩ = I AxB.resid
      using AxB.tuple-proj [of AxB.resid I AxB.resid]
            AxB.simulation-axioms
            comp-simulation-identity [of AxB.resid Dom b AxB.P₀]
            comp-simulation-identity [of AxB.resid Dom a AxB.P₁]
            AxB.P₀.simulation-axioms AxB.P₁.simulation-axioms
      by simp
    thus ?thesis
      using obj-a obj-b simulation-Unpack
            comp-identity-simulation
              [of Dom (a ⊗ b) AxB.resid Unpack a b]
      by auto
  qed
  finally show ?thesis by blast
qed
ultimately show ?thesis by simp
qed
show aXb-C.Cod (Trn (uncurry u)) =
     aXb-C.Cod (Trn (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩))
proof −
  have aXb-C.Cod (Trn (uncurry u)) =
```

```
      aXb-C.Map (aXb-C.trg (Trn (uncurry u)))
  by (simp add: ‹aXb-C.arr (Trn (uncurry u))›)
also have ... = Map (trg (uncurry u))
  using assms(1) trg-char by force
also have ... = Map (uncurry (trg u))
  using assms(1) uncurry-simps by simp
also have ... = Uncurry (Func ∘ Map (trg u)) ∘ Unpack a b
  unfolding uncurry-def mkarr-def by simp
also have ... = Eval.map ∘
                  product-simulation.map (Dom a) (Dom b)
                    (Func ∘ Map (trg u)) B.map ∘
                    Unpack a b
proof −
  have simulation (Dom a) EXP.resid (Func ∘ A-Exp.Map (Trn (trg u)))
    using assms Exp.Map'.simulation-axioms sta-char A-Exp.ide-char_ERTS
          simulation-comp
            [of Dom a Exp.resid A-Exp.Map (Trn (trg u))
                EXP.resid Func]
    by (metis (no-types, lifting) Cod-trg Dom-cod Dom-exp Dom-trg
        H.in-homE H.seqI H-seq-char cod-pr1 ide-trg obj-a pr-simps(4))
  thus ?thesis
    using Eval.Uncurry-simulation-expansion
            [of Dom a Exp.map'_ext ∘ A-Exp.Map (Trn (trg u))]
          A.weakly-extensional-rts-axioms
    by auto
qed
also have ... = Eval.map ∘
                  product-simulation.map (Dom a) (Dom b)
                    (Func ∘ Map (trg u)) B.map ∘
                      (⟨⟨AxB.P_1, AxB.P_0⟩⟩ ∘ Unpack a b)
  by (metis (no-types, lifting) AxB.tuple-proj
      comp-pointwise-tuple obj-a obj-b simulation-Unpack)
also have ... = Eval.map ∘
                  (product-simulation.map (Dom a) (Dom b)
                    (Func ∘ Map (trg u)) B.map ∘
                      ⟨⟨AxB.P_1, AxB.P_0⟩⟩) ∘
                    Unpack a b
  by auto
also have ... = Eval.map ∘
                  ⟨⟨Func ∘ Map (trg u) ∘ AxB.P_1, B.map ∘ AxB.P_0⟩⟩ ∘
                    Unpack a b
proof −
  have simulation (Dom a) EXP.resid (Func ∘ Map (trg u))
    using Exp.Map'.simulation-axioms U.G.simulation-axioms
          simulation-comp Dom-exp H.arrI Map-simps(4) assms
    by auto
  thus ?thesis
    using B.simulation-axioms P_0.transformation-axioms
          P_1.transformation-axioms
```

```
        comp-product-simulation-tuple2
          [of Dom a EXP.resid Func ∘ Map (trg u)
              Dom b Dom b B.map
            AxB.resid AxB.P₁ AxB.P₁ AxB.P₁ AxB.P₀ AxB.P₀ AxB.P₀]
    by (simp add: comp-assoc)
qed
also have ... = Eval.map ∘
                  (FuncxB.map ∘
                    ⟨⟨Map (trg u) ∘ AxB.P₁, AxB.P₀⟩⟩) ∘
                   Unpack a b
proof −
  have 1: Trg u = Map (trg u)
    using assms Map-simps(4) by fastforce
  interpret trg-uoP₁: simulation AxB.resid Exp.resid
                      ‹Map (trg u) ∘ AxB.P₁›
    using 1 AxB.P₁.simulation-axioms U.G.simulation-axioms
          simulation-comp
    by auto
  interpret src-uoP₁: simulation-as-transformation AxB.resid Exp.resid
                      ‹Map (trg u) ∘ AxB.P₁›
    ..
  interpret P₀: simulation-as-transformation AxB.resid ‹Dom b› AxB.P₀
    ..
  show ?thesis
    using src-uoP₁.transformation-axioms B.simulation-axioms
          Exp.Map'.simulation-axioms P₀.transformation-axioms
          comp-product-simulation-tuple2
            [of Exp.resid EXP.resid Func Dom b Dom b B.map
              AxB.resid - - Map (trg u) ∘ AxB.P₁ - - AxB.P₀]
    by (simp add: comp-assoc)
qed
also have ... = (Eval.map ∘
                  FuncxB.map ∘
                    ⟨⟨Map (trg u) ∘ AxB.P₁, AxB.P₀⟩⟩) ∘
                   Unpack a b
  by auto
also have ... = (Eval.map ∘
                  (FuncxB.map ∘
                     (Unpack exp b ∘ Pack exp b)) ∘
                    ⟨⟨Map (trg u) ∘ AxB.P₁, AxB.P₀⟩⟩) ∘
                    Unpack a b
  using obj-b obj-exp Unpack-o-Pack Dom-exp FuncxB.simulation-axioms
        comp-simulation-identity [of ExpxB.resid EXPxB.resid FuncxB.map]
  by presburger
also have ... = Map eval ∘
                 Pack exp b ∘
                   ⟨⟨Map (trg u) ∘ AxB.P₁ ∘ Unpack a b,
                     AxB.P₀ ∘ Unpack a b⟩⟩
  using Map-eval
```

```
        comp-pointwise-tuple
          [of Map (trg u) ∘ AxB.P₁ AxB.P₀ Unpack a b]
    by (simp add: comp-assoc)
  also have ... = Map eval ∘
                  (Pack exp b ∘
                    ⟨⟨Map (trg u) ∘ AxB.P₁ ∘ Unpack a b,
                      AxB.P₀ ∘ Unpack a b⟩⟩)
    using comp-assoc by metis
  also have ... = Map eval ∘
                  (Pack exp b ∘
                    ⟨⟨Map (trg u ⋆ p₁ a b),
                      AxB.P₀ ∘ Unpack a b⟩⟩)
    by (metis (no-types, lifting) H.in-homE H.seqI Map-hcomp
        Map-p₁ assms cod-pr1 comp-assoc dom-trg obj-a obj-b
        pr-simps(4) trg.preserves-arr)
  also have ... = Map eval ∘ Map ⟨trg u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩
  proof −
    have «trg u ⋆ 𝔭₁[a, b] : a ⊗ b → exp»
      using assms(1) obj-a obj-b sta-p₀ [of a b] sta-p₁ [of a b] H.seqI
      by auto
    moreover have «𝔭₀[a, b] : a ⊗ b → b»
      using obj-a obj-b by blast
    ultimately show ?thesis
      using assms(1) obj-a obj-b Map-p₀
            Map-tuple [of trg u ⋆ 𝔭₁[a, b] a ⊗ b exp 𝔭₀[a, b] b]
      by auto
  qed
  also have ... = Map (eval ⋆ ⟨trg u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)
    using assms 0 Map-eval Map-hcomp H.cod-comp H.dom-comp H.seqI
          cod-pr0 cod-pr1 cod-trg cod-tuple dom-trg eval-in-hom
          obj-a obj-b pr-simps(1−2,4−5) trg.preserves-arr tuple-simps(1)
      by (elim H.in-homE) presburger
  also have ... = Map (trg (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩))
  proof −
    have H.seq eval ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩
      using 0 by blast
    moreover have H.span (u ⋆ 𝔭₁[a, b]) 𝔭₀[a, b]
      by (metis (no-types, lifting) H.not-arr-null H-seq-char
          arr-coincidence calculation tuple-ext)
    ultimately show ?thesis
      using obj-a obj-b sta-p₀ sta-p₁ by auto
  qed
  also have ... = Trg (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)
    using 0 trg-char Cod Dom Trn.simps(1) H.arrI Map-simps(4) by blast
  also have ... = aXb-C.Cod (Trn (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩))
    by simp
  finally show ?thesis by simp
qed
fix x
```

assume x: *aXb.ide x*
show *aXb-C.Map (Trn (uncurry u)) x =*
aXb-C.Map (Trn (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩)) x
proof −
have *aXb-C.Map (Trn (uncurry u)) x =*
(Uncurry (Func ∘ Map u) ∘ Unpack a b) x
unfolding *uncurry-def mkarr-def* **by** *simp*
also have ... = *(Eval.map ∘*
product-transformation.map (Dom a) (Dom b)
EXP.resid (Dom b)
(Func ∘ A-Exp.Dom (Trn u)) B.map
(Func ∘ A-Exp.Map (Trn u)) B.map ∘
Unpack a b) x
proof −
have *transformation (Dom a) EXP.resid*
(Func ∘ A-Exp.Dom (Trn u)) (Func ∘ A-Exp.Cod (Trn u))
(Func ∘ A-Exp.Map (Trn u))
using *assms Exp.Map′.simulation-axioms arr-char A-Exp.ide-char*$_{ERTS}$
EXP.weakly-extensional-rts-axioms Dom-exp
U.transformation-axioms transformation-whisker-left
by *simp*
thus *?thesis*
using *Eval.Uncurry-transformation-expansion*
[of Dom a Func ∘ A-Exp.Dom (Trn u)
Func ∘ A-Exp.Cod (Trn u) Func ∘ A-Exp.Map (Trn u)]
A.weakly-extensional-rts-axioms
by *auto*
qed
also have ... = *(Eval.map ∘*
product-transformation.map (Dom a) (Dom b)
EXP.resid (Dom b)
(Func ∘ A-Exp.Dom (Trn u)) B.map
(Func ∘ Map u) B.map ∘
(⟨⟨AxB.P₁, AxB.P₀⟩⟩ ∘ Unpack a b)) x
proof −
have *pointwise-tuple AxB.P₁ AxB.P₀ = I AxB.resid*
using *AxB.tuple-proj [of AxB.resid I AxB.resid]*
comp-simulation-identity [of AxB.resid Dom b AxB.P₀]
comp-simulation-identity [of AxB.resid Dom a AxB.P₁]
AxB.P₀.simulation-axioms AxB.P₁.simulation-axioms
AxB.simulation-axioms
by *simp*
thus *?thesis*
using *obj-a obj-b simulation-Unpack*
comp-identity-simulation
[of Dom (a ⊗ b) AxB.resid Unpack a b]
by *auto*
qed
also have ... = *(Eval.map ∘*

```
                ((product-transformation.map (Dom a) (Dom b)
                   EXP.resid (Dom b)
                   (Func ∘ Src u) B.map
                   (Func ∘ Map u) B.map ∘
                     ⟨⟨AxB.P₁, AxB.P₀⟩⟩) ∘
                   Unpack a b)) x
  by auto
also have ... = (Eval.map ∘
                 (⟨⟨Func ∘ (Map u ∘ AxB.P₁), B.map ∘ AxB.P₀⟩⟩ ∘
                  Unpack a b)) x
proof −
  have transformation (Dom a) EXP.resid
         (Func ∘ Src u) (Func ∘ Trg u) (Func ∘ Map u)
    using assms Exp.Map'.simulation-axioms U.transformation-axioms
         EXP.weakly-extensional-rts-axioms Dom-exp H.arrI
         Map-simps(4) transformation-whisker-left
    by auto
  hence transformation-to-extensional-rts (Dom a) EXP.resid
         (Func ∘ Src u) (Func ∘ Trg u) (Func ∘ Map u)
    using EXP.extensional-rts-axioms
         transformation-to-extensional-rts.intro
    by blast
  thus ?thesis
    using B.simulation-axioms AxB.P₀-is-simulation
         AxB.P₁-is-simulation
         B.transformation-to-extensional-rts-axioms
         comp-product-transformation-tuple
           [of Dom a EXP.resid
               Func ∘ Src u Func ∘ Trg u Func ∘ Map u
               Dom b Dom b B.map B.map B.map
               AxB.resid AxB.P₁ AxB.P₀]
    by (simp add: comp-assoc)
qed
also have ... = (Eval.map ∘
                 (FuncxB.map ∘
                   ⟨⟨Map u ∘ AxB.P₁, AxB.P₀⟩⟩) ∘
                  Unpack a b) x
proof −
  have transformation (\_AxB) Exp.resid
         (Src u ∘ AxB.P₁) (Trg u ∘ AxB.P₁) (Map u ∘ AxB.P₁)
    using transformation-whisker-right AxB.P₁.simulation-axioms
         U.transformation-axioms Dom-exp
         AxB.weakly-extensional-rts-axioms
    by auto
  thus ?thesis
    using B.simulation-axioms Exp.Map'.simulation-axioms
         P₀.transformation-axioms P₁.transformation-axioms
         comp-product-simulation-tuple2
           [of Exp.resid EXP.resid Func Dom b Dom b B.map
```

```
                    AxB.resid - - Map u ∘ AxB.P₁ - - AxB.P₀]
        by simp
      qed
      also have ... = ((Eval.map ∘
                        FuncxB.map ∘
                          ⟨⟨Map u ∘ AxB.P₁, AxB.P₀⟩⟩) ∘
                          Unpack a b) x
        by auto
      also have ... = ((Eval.map ∘
                         (FuncxB.map ∘
                            (Unpack exp b ∘ Pack exp b)) ∘
                              ⟨⟨Map u ∘ AxB.P₁, AxB.P₀⟩⟩) ∘
                              Unpack a b) x
        using obj-b obj-exp Unpack-o-Pack Dom-exp FuncxB.simulation-axioms
              comp-simulation-identity
                [of ExpxB.resid EXPxB.resid FuncxB.map]
        by presburger
      also have ... = (Map eval ∘
                       Pack exp b ∘
                         ⟨⟨Map u ∘ AxB.P₁ ∘ Unpack a b,
                           AxB.P₀ ∘ Unpack a b⟩⟩) x
        using Map-eval
              comp-pointwise-tuple [of Map u ∘ AxB.P₁ AxB.P₀ Unpack a b]
        by (simp add: comp-assoc)
      also have ... = (Map eval ∘
                       (Pack exp b ∘
                          ⟨⟨Map u ∘ (AxB.P₁ ∘ Unpack a b),
                            AxB.P₀ ∘ Unpack a b⟩⟩)) x
        using comp-assoc by metis
      also have ... = (Map eval ∘
                       (Pack exp b ∘
                          ⟨⟨Map (u ⋆ p₁ a b),
                            AxB.P₀ ∘ Unpack a b⟩⟩)) x
        by (metis (no-types, lifting) H.seqI' Map-p₁
            Map-hcomp arr-coincidence assms obj-a obj-b pr-in-hom(2))
      also have ... = (Map eval ∘ Map ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩) x
        by (metis (mono-tags, lifting) H.comp-in-homI Map-p₀ Map-tuple
            assms obj-a obj-b pr-in-hom(1) pr-in-hom(2))
      also have ... = (aXb-C.Map (Trn (eval ⋆ ⟨u ⋆ 𝔭₁[a, b], 𝔭₀[a, b]⟩))) x
        using 0 Map-eval Map-hcomp by auto
      finally show ?thesis by simp
    qed
  qed
qed

end
```

Once again, we transfer the things we want to *rtscatx*.

```
context rtscatx
```

begin

interpretation *elementary-category-with-binary-products hcomp* p_0 p_1
using *extends-to-elementary-category-with-binary-products* **by** *blast*

notation *hcomp* (**infixr** $\star$ *53*)
notation p_0 ($\mathfrak{p}_0$[-, -])
notation p_1 ($\mathfrak{p}_1$[-, -])
notation *tuple* ($\langle$-, -$\rangle$)
notation *prod* (**infixr** $\otimes$ *51*)

definition *curry* :: *'A arr* $\Rightarrow$ *'A arr* $\Rightarrow$ *'A arr* $\Rightarrow$ *'A arr* $\Rightarrow$ *'A arr*
where *curry* $\equiv$ *currying-in-rtscat.curry*

definition *uncurry* :: *'A arr* $\Rightarrow$ *'A arr* $\Rightarrow$ *'A arr* $\Rightarrow$ *'A arr* $\Rightarrow$ *'A arr*
where *uncurry* $\equiv$ *currying-in-rtscat.uncurry*

lemma *curry-in-hom* [*intro*, *simp*]:
assumes *obj a* **and** *obj b*
and «*f* : *a* $\otimes$ *b* $\rightarrow$ *c*»
shows «*curry a b c f* : *a* $\rightarrow$ *exp b c*»
proof −
interpret *Currying*: *currying-in-rtscat arr-type a b c*
using *assms* **by** *unfold-locales auto*
show *?thesis*
unfolding *curry-def exp-def*
using *assms Currying.curry-in-hom* **by** *blast*
qed

lemma *curry-simps* [*simp*]:
assumes *obj a* **and** *obj b*
and «*f* : *a* $\otimes$ *b* $\rightarrow$ *c*»
shows *arr* (*curry a b c f*)
and *dom* (*curry a b c f*) = *a* **and** *cod* (*curry a b c f*) = *exp b c*
and *src* (*curry a b c f*) = *curry a b c* (*src f*)
and *trg* (*curry a b c f*) = *curry a b c* (*trg f*)
proof −
interpret *Currying*: *currying-in-rtscat arr-type a b c*
using *assms* **by** *unfold-locales auto*
show *arr* (*curry a b c f*)
and *dom* (*curry a b c f*) = *a* **and** *cod* (*curry a b c f*) = *exp b c*
using *assms curry-in-hom H.in-homE H-arr-char arr-char*
apply (*metis* (*no-types*, *lifting*))
by (*metis* (*no-types*, *lifting*) *H.in-homE assms curry-in-hom*)+
show *src* (*curry a b c f*) = *curry a b c* (*src f*)
and *trg* (*curry a b c f*) = *curry a b c* (*trg f*)
unfolding *curry-def*
using *assms* **by** *auto*
qed

lemma *sta-curry*:
assumes *obj a* **and** *obj b*
and «*f* : *a* ⊗ *b* → *c*» **and** *sta f*
shows *sta* (*curry a b c f*)
using *assms V.ide-iff-src-self* [*of curry a b c f*] **by** *auto*

lemma *uncurry-in-hom* [*intro*, *simp*]:
assumes *obj b* **and** *obj c*
and «*g* : *a* → *exp b c*»
shows «*uncurry a b c g* : *a* ⊗ *b* → *c*»
proof −
interpret *Currying*: *currying-in-rtscat arr-type a b c*
using *assms* **by** *unfold-locales auto*
show *?thesis*
using *assms*
unfolding *uncurry-def exp-def*
using *Currying.uncurry-in-hom* **by** *blast*
qed

lemma *uncurry-simps* [*simp*]:
assumes *obj b* **and** *obj c*
and «*g* : *a* → *exp b c*»
shows *arr* (*uncurry a b c g*)
and *dom* (*uncurry a b c g*) = *a* ⊗ *b* **and** *cod* (*uncurry a b c g*) = *c*
and *src* (*uncurry a b c g*) = *uncurry a b c* (*src g*)
and *trg* (*uncurry a b c g*) = *uncurry a b c* (*trg g*)
proof −
interpret *Currying*: *currying-in-rtscat arr-type a b c*
using *assms* **by** *unfold-locales auto*
show *arr* (*uncurry a b c g*)
and *dom* (*uncurry a b c g*) = *a* ⊗ *b* **and** *cod* (*uncurry a b c g*) = *c*
using *assms uncurry-in-hom H.in-homE H-arr-char arr-char*
apply (*metis* (*no-types*, *lifting*))
by (*metis* (*no-types*, *lifting*) *H.in-homE assms uncurry-in-hom*)+
show *src* (*uncurry a b c g*) = *uncurry a b c* (*src g*)
and *trg* (*uncurry a b c g*) = *uncurry a b c* (*trg g*)
using *assms*
by (*auto simp add*: *uncurry-def exp-def*)
qed

lemma *sta-uncurry*:
assumes *obj b* **and** *obj c*
and «*g* : *a* → *exp b c*» **and** *sta g*
shows *sta* (*uncurry a b c g*)
using *assms V.ide-iff-src-self* [*of uncurry a b c g*] **by** *auto*

lemma *uncurry-curry*:
assumes *obj a* **and** *obj b*

and «$t : a \otimes b \to c$»
shows *uncurry a b c* (*curry a b c t*) = *t*
proof −
  interpret *Currying*: *currying-in-rtscat arr-type a b c*
    using *assms* **by** *unfold-locales auto*
  show *?thesis*
    unfolding *curry-def uncurry-def*
    using *assms Currying.uncurry-curry* **by** *blast*
qed

lemma *curry-uncurry*:
assumes *obj b* **and** *obj c*
and «$u : a \to exp\ b\ c$»
shows *curry a b c* (*uncurry a b c u*) = *u*
proof −
  interpret *Currying*: *currying-in-rtscat arr-type a b c*
    using *assms* **by** *unfold-locales auto*
  show *?thesis*
    using *assms*
    unfolding *curry-def uncurry-def exp-def*
    using *Currying.curry-uncurry* **by** *blast*
qed

lemma *uncurry-expansion*:
assumes *obj b* **and** *obj c*
and «$u : a \to exp\ b\ c$»
shows *uncurry a b c u* = *eval b c* ⋆ (*u* ⊗ *b*)
proof −
  have *a*: *obj a*
    using *assms*(*3*) **by** *auto*
  interpret *Currying*: *currying-in-rtscat arr-type a b c*
    using *assms* **by** *unfold-locales auto*
  have *uncurry a b c u* = *eval b c* ⋆ ⟨*u* ⋆ $\mathfrak{p}_1[a,\ b]$, $\mathfrak{p}_0[a,\ b]$⟩
    using *assms*
    unfolding *curry-def uncurry-def exp-def eval-def*
    using *Currying.uncurry-expansion* **by** *blast*
  also have ... = *eval b c* ⋆ (*u* ⊗ *b*) ⋆ ⟨$\mathfrak{p}_1[a,\ b]$, $\mathfrak{p}_0[a,\ b]$⟩
  proof −
    have *b* ⋆ $\mathfrak{p}_0[a,\ b]$ = $\mathfrak{p}_0[a,\ b]$
      using *assms a sta-p*$_0$
      by (*simp add*: *H.comp-cod-arr*)
    moreover have *H.seq u* $\mathfrak{p}_1[a,\ b]$
      using *assms sta-p*$_1$ [*of a b*]
      by (*intro H.seqI*) *auto*
    ultimately show *?thesis*
      using *assms prod-tuple* [*of* $\mathfrak{p}_1[a,\ b]$ $\mathfrak{p}_0[a,\ b]$ *u b*]
            sta-p$_0$ [*of a b*] *sta-p*$_1$ [*of a b*]
      by *auto*
  qed

```
  also have ... = eval b c ⋆ (u ⊗ b)
    using assms a tuple-pr [of a b] H.comp-arr-ide
    by (metis (no-types, lifting) H.comp-arr-dom H.comp-ide-self H.ideD(1)
        H.in-homE interchange)
  finally show ?thesis by blast
qed

lemma Map-curry:
assumes obj a and obj b and obj c
shows Map (curry a b c f) =
      Unfunc b c ∘
        Currying.Curry (Dom a) (Dom b) (Dom c)
          (Src f ∘ Pack a b) (Trg f ∘ Pack a b) (Map f ∘ Pack a b)
proof −
  interpret Currying: currying-in-rtscat arr-type a b c
    using assms by unfold-locales auto
  show ?thesis
    unfolding curry-def Currying.curry-def Unfunc-def mkarr-def by simp
qed

lemma Map-uncurry:
assumes obj a and obj b and obj c
shows Map (uncurry a b c g) =
      Currying.Uncurry (Dom a) (Dom b) (Dom c)
        (Func b c ∘ exponential-rts.Map (Trn g)) ∘ Unpack a b
proof −
  interpret Currying: currying-in-rtscat arr-type a b c
    using assms by unfold-locales auto
  show ?thesis
    unfolding uncurry-def Currying.uncurry-def Func-def mkarr-def by simp
qed

end
```

4.3.4 Cartesian Closure

We can now show that the category $\mathbf{RTS}^\dagger$ is cartesian closed.

```
context rtscatx
begin

  interpretation elementary-category-with-binary-products hcomp p0 p1
    using extends-to-elementary-category-with-binary-products by blast

  notation prod    (infixr ⊗ 51)

  interpretation elementary-cartesian-closed-category
                   hcomp p0 p1 one trm exp eval curry
  proof
    fix b c
```

```
    assume b: obj b and c: obj c
    show «eval b c : exp b c ⊗ b → c»
      using b c eval-in-hom_RCR by blast
    show obj (exp b c)
      using b c obj-exp by blast
    fix a
    assume a: obj a
    show ⋀t. «t : a ⊗ b → c» ⟹ «curry a b c t : a → exp b c»
      using a b c curry-in-hom by blast
    show ⋀t. «t : a ⊗ b → c» ⟹ eval b c ⋆ (curry a b c t ⊗ b) = t
      by (metis ‹⋀t. «t : a ⊗ b → c» ⟹ «curry a b c t : a → exp b c»›
          a b c uncurry-curry uncurry-expansion)
    show ⋀u. «u : a → exp b c» ⟹ curry a b c (eval b c ⋆ (u ⊗ b)) = u
      using b c curry-uncurry uncurry-expansion by force
  qed

  lemma is-elementary-cartesian-closed-category:
  shows elementary-cartesian-closed-category
          hcomp p_0 p_1 one trm exp eval curry
    ..

  theorem is-cartesian-closed-category:
  shows cartesian-closed-category hcomp
    ..

end
```

4.3.5 Repleteness

```
context rtscatx
begin
```

We have shown that the RTS-category $\mathbf{RTS}^\dagger$ has objects that are in bijective correspondence with small extensional RTS's, states (identities for the vertical residuation) that are in bijective correspondence with simulations, and arrows that are in bijective correspondence with transformations. These results allow us to pass back and forth between external constructions on RTS's and internal structure of the RTS-category, as was demonstrated in the proof of cartesian closure. However, these results make use of extra structure beyond that of an RTS-category; namely the mapping *Dom* that takes an object to its underlying RTS. We would like to have a characterization of $\mathbf{RTS}^\dagger$ in terms that make sense for an abstract RTS-category without additional structure. It seems that it should be possible to do this, because as we have shown, for any object *a* the RTS *Dom a* is isomorphic to *Hom* **1** *a*. So we ought to be able to dispense with the extrinsic mapping *Dom* and work instead with the intrinsic mapping *Hom* **1**. However, there is an issue here to do with types. The mapping *Dom* takes an object *a* to a small extensional RTS *Dom a* having arrow type $'A$. On the other hand, the RTS

Hom **1** *a* has arrow type *'A arr*. So one thing that needs to be done in order to carry out this program is to express the "object repleteness" of **RTS**† in terms of small extensional RTS's with arrow type *'A arr*, as opposed to small extensional RTS's with arrow type *'A*. However, the type *'A arr* is larger than the type *'A*, and consequently it could admit a larger class of small extensional RTS's than type *'A* does. It is possible, though, to define a mapping from *'A arr* to *'A* whose restriction to the set of arrows (and null) of *rtscatx* is injective. This will allow us to take any small extensional RTS *A* with arrow type *'A arr*, as long as its arrows and null are drawn from the set of arrows and null of *rtscatx* as a whole, and obtain an isomorphic image of it with arrow type *'A*.

We first define the required mapping from *'A arr* to *'A*.

```
fun inj-arr :: 'A arr ⇒ 'A
where inj-arr (MkArr A B F) =
      lifting.some-lift
        (Some (pairing.some-pair
                (some-inj-resid A,
                 pairing.some-pair
                   (some-inj-resid B, inj-exp F))))
  | inj-arr Null = lifting.some-lift None
```

The mapping *inj-arr* has the required injectiveness property.

```
lemma inj-inj-arr:
fixes A :: 'A arr resid
assumes small-rts A and extensional-rts A
and Collect (residuation.arr A) ∪
              {ResiduatedTransitionSystem.partial-magma.null A} ⊆
    Collect arr ∪ {Null}
shows inj-on inj-arr
        (Collect (residuation.arr A) ∪
          {ResiduatedTransitionSystem.partial-magma.null A})
proof
  interpret A: small-rts A
    using assms(1) by blast
  interpret A: extensional-rts A
    using assms(2) by blast
  fix x y :: 'A arr
  assume x: x ∈ Collect A.arr ∪ {A.null}
  assume y: y ∈ Collect A.arr ∪ {A.null}
  assume eq: inj-arr x = inj-arr y
  show x = y
  proof −
    have ⟦x = Null; y ≠ Null⟧ ⟹ ?thesis
      using eq
      apply (cases x; cases y)
      by (auto simp add: inj-eq inj-some-lift)
    moreover have ⟦x ≠ Null; y = Null⟧ ⟹ ?thesis
```

```
  using eq
  apply (cases x; cases y)
  by (auto simp add: inj-eq inj-some-lift)
moreover have ⟦x ≠ Null; y ≠ Null⟧ ⟹ x = y
proof −
  assume x′: x ≠ Null and y′: y ≠ Null
  have lifting.some-lift
         (Some (pairing.some-pair
                 (some-inj-resid (Dom x),
                  pairing.some-pair
                    (some-inj-resid (Cod x), inj-exp (Trn x))))) =
       lifting.some-lift
         (Some (pairing.some-pair
                 (some-inj-resid (Dom y),
                  pairing.some-pair
                    (some-inj-resid (Cod y), inj-exp (Trn y)))))
    using eq x′ y′
    by (cases x; cases y) auto
  hence Some (pairing.some-pair
                 (some-inj-resid (Dom x),
                  pairing.some-pair
                    (some-inj-resid (Cod x), inj-exp (Trn x)))) =
        Some (pairing.some-pair
                 (some-inj-resid (Dom y),
                  pairing.some-pair
                    (some-inj-resid (Cod y), inj-exp (Trn y))))
    using inj-some-lift injD by metis
  hence pairing.some-pair
                 (some-inj-resid (Dom x),
                  pairing.some-pair
                    (some-inj-resid (Cod x), inj-exp (Trn x))) =
        pairing.some-pair
                 (some-inj-resid (Dom x),
                  pairing.some-pair
                    (some-inj-resid (Cod x), inj-exp (Trn x)))
    by auto
  hence some-inj-resid (Dom x) = some-inj-resid (Dom y) ∧
        pairing.some-pair (some-inj-resid (Cod x), inj-exp (Trn x)) =
        pairing.some-pair (some-inj-resid (Cod x), inj-exp (Trn y))
    using inj-some-pair
    by (metis ‹Some (some-pair
                      (some-inj-resid (Dom x),
                       some-pair (some-inj-resid (Cod x), inj-exp (Trn x)))) =
               Some (some-pair
                      (some-inj-resid (Dom y),
                       some-pair (some-inj-resid (Cod y), inj-exp (Trn y))))›
        first-conv option.inject second-conv)
  hence 1: some-inj-resid (Dom x) = some-inj-resid (Dom y) ∧
           some-inj-resid (Cod x) = some-inj-resid (Cod y) ∧
```

```
            inj-exp (Trn x) = inj-exp (Trn y)
        using inj-some-pair
        by (metis Pair-inject
            ‹Some (some-pair (some-inj-resid (Dom x),
             some-pair (some-inj-resid (Cod x), inj-exp (Trn x)))) =
             Some (some-pair (some-inj-resid (Dom y),
             some-pair (some-inj-resid (Cod y), inj-exp (Trn y))))›
             injD option.inject)
      have Dom x = Dom y ∧ Cod x = Cod y ∧ Trn x = Trn y
      proof −
        have 2: small-rts (Dom x) ∧ small-rts (Dom y) ∧
                small-rts (Cod x) ∧ small-rts (Cod y)
       using assms x y x′ y′ 1 arr-char inj-on-some-inj-resid small-function-resid
          by blast
        have 3: Dom x = Dom y ∧ Cod x = Cod y
          using 1 2 inj-on-some-inj-resid inj-on-def
          by (metis mem-Collect-eq)
        moreover have Trn x = Trn y
        proof −
         have residuation.arr (exponential-rts.resid (Dom x) (Cod x)) (Trn x) ∧
              residuation.arr (exponential-rts.resid (Dom x) (Cod x)) (Trn y)
           by (metis (no-types, lifting) Un-insert-right arrE assms(3) calculation
              insertE mem-Collect-eq subsetD sup-bot.right-neutral x x′ y y′)
          thus ?thesis
            using 1 2 inj-inj-exp inj-on-def [of inj-exp]
                  arr-char assms(3) x x′
            by auto
        qed
        ultimately show ?thesis by blast
      qed
      thus x = y
        apply (cases x; cases y)
          apply auto[4]
        using x′ y′ by blast+
    qed
    ultimately show ?thesis by blast
  qed
qed
```

The following result says that, for any small extensional RTS A whose arrows inhabit type *'A arr resid* and are drawn from among the arrows and null of *rtscatx*, there is an object a of *rtscatx* such that the RTS *HOM* **1** a is isomorphic to A. It is expressed in terms that are intrinsic to $\mathbf{RTS}^\dagger$ as an abstract RTS-category, as opposed to the fact *bij-mkobj*, which uses the extrinsically given mapping *Dom*. The result is proved by taking an isomorphic image of the given RTS A under the injective mapping *inj-arr* :: *'A arr* ⇒ *'A*, then applying *bij-mkobj* to obtain the corresponding object a, and finally using the isomorphism *Dom a* ≅ *HOM* **1** a to conclude that *HOM* **1** $a \cong A$.

lemma *obj-replete*:
fixes $A :: 'A\ arr\ resid$
assumes *small-rts A* $\wedge$ *extensional-rts A*
and *Collect* (*residuation.arr A*) $\cup$
$\{$*ResiduatedTransitionSystem.partial-magma.null A*$\}$
$\subseteq$ *Collect arr* $\cup$ $\{$*null*$\}$
shows $\exists a.$ *obj a* $\wedge$ *isomorphic-rts A* (*HOM* **1** *a*)
proof −
interpret *A*: *small-rts A*
using *assms* **by** *blast*
interpret *A*: *extensional-rts A*
using *assms* **by** *blast*
obtain $\iota :: 'A\ arr \Rightarrow 'A$
where ι: *inj-on* ι (*Collect A.arr* $\cup$ $\{$*A.null*$\}$)
using *assms inj-inj-arr* [*of A*] *null-char* **by** *auto*
interpret ιA: *inj-image-rts* ι *A*
using ι **by** *unfold-locales blast*
have *small-rts* $\iota A.resid$ $\wedge$ *extensional-rts* $\iota A.resid$
using *assms* $\iota A.preserves\text{-}reflects\text{-}small\text{-}rts$ $\iota A.preserves\text{-}extensional\text{-}rts$
by *blast*
have ιA: *isomorphic-rts A* $\iota A.resid$
using $\iota A.F.invertible$ *isomorphic-rts-def* **by** *blast*

let *?a* = *mkobj* $\iota A.resid$
have *a*: *obj ?a* $\wedge$ *Dom ?a* = $\iota A.resid$
using ‹*small-rts* $\iota A.resid$ $\wedge$ *extensional-rts* $\iota A.resid$› *bij-mkobj* **by** *auto*
hence *obj ?a* $\wedge$ *isomorphic-rts* $\iota A.resid$ (*HOM* **1** *?a*)
using *inverse-simulations-DN-UP* [*of ?a*] *isomorphic-rts-def* **by** *auto*
hence *obj ?a* $\wedge$ *isomorphic-rts A* (*HOM* **1** *?a*)
using ιA *isomorphic-rts-transitive* **by** *blast*
thus *?thesis* **by** *blast*
qed

We now turn our attention to showing that, for any given objects a and b, the states from a to b correspond bijectively (via the "covariant hom" mapping *cov-HOM*) to simulations from *HOM* **1** a to *HOM* **1** b and the arrows from a to b correspond bijectively to the transformations between such simulations.

lemma *HOM1-faithful-for-sta*:
assumes «$f : a \rightarrow_{sta} b$» **and** «$g : a \rightarrow_{sta} b$»
and *cov-HOM* **1** f = *cov-HOM* **1** g
shows $f = g$
proof −
interpret *A*: *extensional-rts* ‹*Dom a*›
using *assms*(*1*) *obj-char arr-char*
by (*metis* (*no-types*, *lifting*) *H.ide-dom H.in-homE Int-Collect mem-Collect-eq*)
interpret *A*: *small-rts* ‹*Dom a*›
using *assms*(*1*) *obj-char arr-char*
by (*metis* (*no-types*, *lifting*) *H.ide-dom H.in-homE Int-Collect*)

interpret *B*: *extensional-rts* ‹*Dom b*›
  using *assms*(*1*) *obj-char arr-char*
by (*metis* (*no-types*, *lifting*) *H.ide-cod H.in-homE Int-Collect mem-Collect-eq*)
interpret *AB*: *exponential-rts* ‹*Dom a*› ‹*Dom b*› **..**
interpret *HOM-1a*: *sub-rts resid* ‹$\lambda t.$ «$t : \mathbf{1} \to a$»›
  using *sub-rts-HOM* **by** *simp*
interpret *F*: *simulation* ‹*Dom a*› ‹*Dom b*› ‹*AB.Map* (*Trn f*)›
  using *assms*(*1*) *sta-char*
  by (*metis* (*no-types*, *lifting*) *AB.ide-char*$_{ERTS}$ *Dom-cod Dom-dom*
      H.in-homE V.residuation-axioms residuation.ide-implies-arr)
interpret *G*: *simulation* ‹*Dom a*› ‹*Dom b*› ‹*AB.Map* (*Trn g*)›
  using *assms*(*2*) *sta-char*
  by (*metis* (*no-types*, *lifting*) *AB.ide-char*$_{ERTS}$ *Dom-cod Dom-dom*
      H.in-homE V.residuation-axioms residuation.ide-implies-arr)
have $\bigwedge Q\ R\ T.$ *transformation* ($\backslash_1$) (*Dom a*) *Q R T*
            $\Longrightarrow$ *AB.Map* (*Trn f*) $\circ$ *T* = *AB.Map* (*Trn g*) $\circ$ *T*
proof −
  fix *Q R T*
  assume *T*: *transformation* ($\backslash_1$) (*Dom a*) *Q R T*
  interpret *T*: *transformation* ‹($\backslash_1$)› ‹*Dom a*› *Q R T*
    using *T* **by** *blast*
  let *?t* = *mkarr One.resid* (*Dom a*) *Q R T*
  have *t*: «*?t* : $\mathbf{1} \to a$»
  proof
    show *H.arr ?t*
     using *A.extensional-rts-axioms A.small-rts-axioms One.is-extensional-rts*
          One.small-rts-axioms T
      by *auto*
    show *dom ?t* = $\mathbf{1}$
      by (*simp add*: *A.extensional-rts-axioms A.small-rts-axioms*
        One.is-extensional-rts One.small-rts-axioms T arr-mkarr(*4*) *one-def*)
    show *cod ?t* = *a*
     using *One.is-extensional-rts One.small-rts-axioms T assms*(*1*) *dom-char*
     by *fastforce*
  qed
  hence *HOM-1a.arr ?t*
    using *assms HOM-1a.arr-char* **by** *blast*
  moreover have *dom f* = *a* **and** *dom g* = *a*
    using *assms* **by** *blast*+
  ultimately have *1*: *f* ⋆ *?t* = *g* ⋆ *?t*
    using *assms*
    by *auto meson*
  have *AB.Map* (*Trn f*) $\circ$ *T* = *Map f* $\circ$ *T*
    by *simp*
  also have ... = *Map* (*f* ⋆ *?t*)
    by (*metis* (*no-types*, *lifting*) *A.extensional-rts-axioms*
      A.small-rts-axioms H.seqI′ Map-hcomp One.is-extensional-rts
      One.small-rts-axioms T assms(*1*) *bij-mkarr*(*3*) *t transformation-def*)
  also have ... = *Map* (*g* ⋆ *?t*)

```
    using 1 by simp
  also have ... = Map g ∘ T
    using assms t Map-hcomp [of g ?t] mkarr-def by auto
  also have ... = AB.Map (Trn g) ∘ T
    by simp
  finally show AB.Map (Trn f) ∘ T = AB.Map (Trn g) ∘ T by blast
 qed
 thus ?thesis
  using One.eq-simulation-iff A.weakly-extensional-rts-axioms
      B.weakly-extensional-rts-axioms F.simulation-axioms
      G.simulation-axioms
  by (metis (no-types, lifting) AB.MkIde-Map Dom-cod Dom-dom H.in-homE
      MkArr-Trn V.ide-implies-arr assms(1-2) sta-char)
qed

lemma HOM1-faithful-for-arr:
assumes arr t and arr u and src t = src u and trg t = trg u
and cov-HOM 𝟭 t = cov-HOM 𝟭 u
shows t = u
proof (intro arr-eqI)
 let ?a = dom t and ?b = cod t
 have a: dom t = ?a ∧ dom u = ?a
  using assms dom-src [of t] by auto
 have b: cod t = ?b ∧ cod u = ?b
  using assms cod-src [of t] by auto
 let ?A = Dom t and ?B = Cod t
 have A: Dom t = ?A ∧ Dom u = ?A
  using assms Dom-src [of t] by auto
 have B: Cod t = ?B ∧ Cod u = ?B
  using assms Cod-src [of t] by auto
 interpret A: extensional-rts ?A
  using assms(1) arr-char by blast
 interpret A: small-rts ?A
  using assms(1) arr-char by auto
 interpret B: extensional-rts ?B
  using assms(1) arr-char by auto
 interpret AB: exponential-rts ?A ?B ..
 interpret HOM-1a: sub-rts resid ‹λx. «x : 𝟭 → dom t»›
  using sub-rts-HOM by blast
 interpret HOM-1b: sub-rts resid ‹λx. «x : 𝟭 → cod t»›
  using sub-rts-HOM by blast
 have ∗: ⋀Q R X. transformation (\₁) ?A Q R X
                  ⟹ AB.Map (Trn t) ∘ X = AB.Map (Trn u) ∘ X
 proof −
  fix Q R X
  assume X: transformation (\₁) ?A Q R X
  interpret X: transformation ‹(\₁)› ?A Q R X
   using X by blast
  let ?x = mkarr One.resid ?A Q R X
```

```
have x: «?x : 𝟭 → ?a»
proof
  show 1: H.arr (mkarr (\₁) (Dom t) Q R X)
    using X arr-mkarr(1) [of One.resid ?A Q R X]
          One.small-rts-axioms One.extensional-rts-axioms
          A.small-rts-axioms A.extensional-rts-axioms
    by simp
  show dom (mkarr (\₁) (Dom t) Q R X) = 𝟭
    using X arr-mkarr(4) [of One.resid ?A Q R X] one-def
          One.small-rts-axioms One.extensional-rts-axioms
          A.small-rts-axioms A.extensional-rts-axioms
    by metis
  show cod (mkarr (\₁) (Dom t) Q R X) = ?a
  proof −
    have (λta. if A.arr ta then ta
                else ResiduatedTransitionSystem.partial-magma.null
                     (Dom (dom t))) =
         I (Dom t)
      using assms(1) Dom-dom by presburger
    thus ?thesis
      using assms(1) X arr-mkarr(5) obj-char [of ?a] Dom-dom
            One.small-rts-axioms One.extensional-rts-axioms
            A.small-rts-axioms A.extensional-rts-axioms
      by simp
  qed
qed
have 1: t ⋆ ?x = u ⋆ ?x
proof −
  have HOM-1a.arr ?x
    using assms x HOM-1a.arr-char by blast
  moreover have dom t = ?a and dom u = ?a
    using a by blast+
  ultimately show ?thesis
    using assms
    by auto meson
qed
have AB.Map (Trn t) ∘ X = Map t ∘ X
  by simp
also have ... = Map (t ⋆ ?x)
proof −
  have H.seq t ?x
    using assms x H.seqI by auto
  thus ?thesis
    using assms x Map-hcomp mkarr-def by simp
qed
also have ... = Map (u ⋆ ?x)
  using 1 by simp
also have ... = Map u ∘ X
proof −
```

```
    have H.seq u ?x
      by (metis (no-types, lifting) 1 H.arr-cod-iff-arr H.dom-null
          H.ext H.in-homE H.seqI dom-char x)
    thus ?thesis
      using assms x Map-hcomp mkarr-def by simp
  qed
  also have ... = AB.Map (Trn u) ∘ X
    by simp
  finally show AB.Map (Trn t) ∘ X = AB.Map (Trn u) ∘ X by blast
 qed
 show t ≠ Null and u ≠ Null
  using assms(1−2) arr-char by blast+
 show Dom t = Dom u and Cod t = Cod u
  using A B by auto
 show Trn t = Trn u
 proof (intro AB.arr-eqI)
  show AB.arr (Trn t) and AB.arr (Trn u)
   using assms(1−2) A B arr-char by auto
  show Dom: AB.Dom (Trn t) = AB.Dom (Trn u)
   using assms(1−3) arr-char Map-simps
   apply auto[1]
   by (metis (no-types, lifting))
  show Cod: AB.Cod (Trn t) = AB.Cod (Trn u)
   using assms(1−2,4) arr-char Map-simps
   apply auto[1]
   by (metis (no-types, lifting))
  have AB.Map (Trn t) = AB.Map (Trn u)
   using assms(1−2) ∗ Dom Cod A B arr-char arr-char
         AB.arr-char AB.arr-char
         One.eq-transformation-iff
           [of ?A ?B AB.Dom (Trn t) AB.Cod (Trn t)
               AB.Map (Trn t) AB.Map (Trn u)]
         A.weakly-extensional-rts-axioms
         B.weakly-extensional-rts-axioms
   by simp
  thus ⋀a. A.ide a ⟹ AB.Map (Trn t) a = AB.Map (Trn u) a by simp
 qed
qed

lemma HOM1-full-for-sta:
assumes obj a and obj b and simulation (HOM 1 a) (HOM 1 b) F
shows ∃f. «f : a → b» ∧ sta f ∧ cov-HOM 1 f = F
proof −
 interpret A: extensional-rts ‹Dom a›
  using assms obj-char arr-char by blast
 interpret A: small-rts ‹Dom a›
  using assms obj-char arr-char by blast
 interpret B: extensional-rts ‹Dom b›
  using assms obj-char arr-char by blast
```

interpret *B*: *small-rts* ‹*Dom b*›
  using *assms obj-char arr-char* **by** *blast*
interpret *A1*: *exponential-by-One arr-type* ‹*Dom a*› **..**
interpret *B1*: *exponential-by-One arr-type* ‹*Dom b*› **..**
interpret *HOM-1a*: *sub-rts resid* ‹$\lambda t.$ «*t*: **1** → *a*»›
  using *assms sub-rts-HOM* **by** *blast*
interpret *HOM-1b*: *sub-rts resid* ‹$\lambda t.$ «*t*: **1** → *b*»›
  using *assms sub-rts-HOM* **by** *blast*
interpret *UP-DN-a*: *inverse-simulations*
                ‹*Dom a*› *HOM-1a.resid* ‹DN_{rts} *a*› ‹UP_{rts} *a*›
  using *assms inverse-simulations-DN-UP* [*of a*] *dom-char*
  by (*metis one-def*)
interpret *UP-DN-b*: *inverse-simulations*
                ‹*Dom b*› *HOM-1b.resid* ‹DN_{rts} *b*› ‹UP_{rts} *b*›
  using *assms inverse-simulations-DN-UP* [*of b*] *dom-char*
  by (*metis one-def*)
interpret *F*: *simulation* ‹*HOM* **1** *a*› ‹*HOM* **1** *b*› *F*
  using *assms* **by** *blast*
interpret *F′*: *simulation* ‹*Dom a*› ‹*Dom b*› ‹DN_{rts} *b* ∘ *F* ∘ UP_{rts} *a*›
  using *simulation-comp UP-DN-a.G.simulation-axioms*
      UP-DN-b.F.simulation-axioms F.simulation-axioms
  by *blast*
interpret *F′*: *simulation-as-transformation*
            ‹*Dom a*› ‹*Dom b*› ‹DN_{rts} *b* ∘ *F* ∘ UP_{rts} *a*› **..**
show ∃*f*. «*f* : *a* → *b*» ∧ *sta f* ∧ *cov-HOM* **1** *f* = *F*
proof −
  define *f*
  where *f-def*: *f* = *mksta* (*Dom a*) (*Dom b*) (DN_{rts} *b* ∘ *F* ∘ UP_{rts} *a*)
  have *sta-f*: *sta f*
    unfolding *f-def*
    using *sta-mksta*(*1*) *F′.simulation-axioms*
        A.small-rts-axioms A.extensional-rts-axioms
        B.small-rts-axioms B.extensional-rts-axioms
    by *blast*
  moreover have *f*: «*f* : *a* → *b*»
    using *assms sta-f f-def*
    by (*intro H.in-homI*) *auto*
  moreover have *cov-HOM* **1** *f* = *F*
  proof −
    have *cov-HOM* **1** *f* = UP_{rts} *b* ∘ *Map f* ∘ DN_{rts} *a*
      using *f sta-f dom-char cod-char UP-DN-naturality*(*3*) [*of f*]
      by *auto*
    also have ... = (UP_{rts} *b* ∘ DN_{rts} *b*) ∘ *F* ∘ (UP_{rts} *a* ∘ DN_{rts} *a*)
    proof −
      have ... = UP_{rts} *b* ∘ (DN_{rts} *b* ∘ *F* ∘ UP_{rts} *a*) ∘ DN_{rts} *a*
        using *f-def mkarr-def* **by** *auto*
      thus *?thesis*
        using *comp-assoc* **by** (*metis* (*no-types*, *lifting*))
    qed

```
    also have ... = F ∘ (UP_rts a ∘ DN_rts a)
      using comp-identity-simulation [of HOM-1a.resid HOM-1b.resid F]
            F.simulation-axioms UP-DN-b.inv
      by auto
    also have ... = F
      using comp-simulation-identity [of HOM-1a.resid HOM-1b.resid F]
            F.simulation-axioms UP-DN-a.inv
      by auto
    finally show ?thesis by blast
  qed
  ultimately show ?thesis by blast
 qed
qed

lemma HOM1-full-for-arr:
assumes sta f and sta g and H.par f g
and transformation (HOM 1 (dom f)) (HOM 1 (cod f))
      (cov-HOM 1 f) (cov-HOM 1 g) T
shows ∃t. arr t ∧ src t = f ∧ trg t = g ∧ cov-HOM 1 t = T
proof −
 let ?a = dom f and ?b = cod f
 have a: obj ?a and b: obj ?b
   using assms by auto
 have f: «f : ?a → ?b» and g: «g : ?a → ?b»
   using assms by auto
 let ?A = Dom ?a and ?B = Dom ?b
 have 0: Dom f = ?A ∧ Cod f = ?B ∧ Dom g = ?A ∧ Cod g = ?B
   using assms f dom-char cod-char
   by (metis (no-types, lifting) Dom-cod Dom-dom arr-coincidence)
 have A: small-rts ?A ∧ extensional-rts ?A
   using assms arr-char by auto
 have B: small-rts ?B ∧ extensional-rts ?B
   using assms arr-char by auto
 interpret A: extensional-rts ?A using A by blast
 interpret A: small-rts ?A using A by blast
 interpret B: extensional-rts ?B using B by blast
 interpret B: small-rts ?B using B by blast
 interpret AB: exponential-rts ?A ?B ..
 interpret HOM-1a: sub-rts resid ‹λt. «t: 1 → ?a»›
   using a sub-rts-HOM by blast
 interpret HOM-1a: sub-rts-of-extensional-rts resid ‹λt. «t: 1 → ?a»› ..
 interpret HOM-1a: small-rts ‹HOM 1 ?a›
 proof −
   have Collect HOM-1a.arr ⊆ H.hom 1 ?a
     using HOM-1a.arr-char by blast
   thus small-rts (HOM 1 ?a)
     using assms obj-one H.terminal-def small-homs [of 1 ?a]
           smaller-than-small
     by unfold-locales auto
```

qed
interpret *HOM-1b*: *sub-rts resid* ‹λ*t*. «*t*: **1** → *?b*»›
  using *b sub-rts-HOM* **by** *blast*
interpret *HOM-1b*: *sub-rts-of-extensional-rts resid* ‹λ*t*. «*t*: **1** → *?b*»› **..**
interpret *HOM-1b*: *small-rts* ‹*HOM* **1** *?b*›
proof −
  have *Collect HOM-1b.arr* ⊆ *H.hom* **1** *?b*
    using *HOM-1b.arr-char* **by** *blast*
  thus *small-rts* (*HOM* **1** *?b*)
    using *assms obj-one H.terminal-def small-homs* [*of* **1** *?b*]
        smaller-than-small
    by *unfold-locales auto*
qed
interpret *UP-DN-a*: *inverse-simulations*
                  ?A HOM-1a.resid ‹DN_{rts} *?a*› ‹UP_{rts} *?a*›
  using *assms a inverse-simulations-DN-UP* [*of ?a*] *dom-char one-def*
  by *metis*
interpret *UP-DN-b*: *inverse-simulations*
                  ?B HOM-1b.resid ‹DN_{rts} *?b*› ‹UP_{rts} *?b*›
  using *assms b inverse-simulations-DN-UP* [*of ?b*] *dom-char one-def*
  by *metis*
interpret *T*: *transformation*
            ‹*HOM* **1** *?a*› ‹*HOM* **1** *?b*› ‹*cov-HOM* **1** *f*› ‹*cov-HOM* **1** *g*› *T*
  using *assms*(4) **by** *blast*
interpret *F′*: *simulation ?A ?B* ‹DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*›
  using *simulation-comp UP-DN-a.G.simulation-axioms*
      UP-DN-b.F.simulation-axioms T.F.simulation-axioms
  by *blast*
interpret *G′*: *simulation ?A ?B* ‹DN_{rts} *?b* ∘ *cov-HOM* **1** *g* ∘ UP_{rts} *?a*›
  using *simulation-comp UP-DN-a.G.simulation-axioms*
      UP-DN-b.F.simulation-axioms T.G.simulation-axioms
  by *blast*
interpret *DN-T*: *transformation HOM-1a.resid ?B*
              ‹DN_{rts} *?b* ∘ *cov-HOM* **1** *f*› ‹DN_{rts} *?b* ∘ *cov-HOM* **1** *g*›
              ‹DN_{rts} *?b* ∘ *T*›
  using *UP-DN-a.G.simulation-axioms UP-DN-b.F.simulation-axioms*
      T.transformation-axioms F′.simulation-axioms G′.simulation-axioms
      transformation-whisker-left B.weakly-extensional-rts-axioms
  by *fastforce*
interpret *T′*: *transformation ?A ?B*
            ‹DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*›
            ‹DN_{rts} *?b* ∘ *cov-HOM* **1** *g* ∘ UP_{rts} *?a*›
            ‹DN_{rts} *?b* ∘ *T* ∘ UP_{rts} *?a*›
  using *UP-DN-a.G.simulation-axioms UP-DN-b.F.simulation-axioms*
      T.transformation-axioms DN-T.transformation-axioms
      DN-T.F.simulation-axioms DN-T.G.simulation-axioms
      transformation-whisker-right A.weakly-extensional-rts-axioms
  by *fastforce*
define *t*

```
  where t-def: t = mkarr ?A ?B
                   (DN_rts ?b ∘ cov-HOM 1 f ∘ UP_rts ?a)
                   (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a)
                   (DN_rts ?b ∘ T ∘ UP_rts ?a)
have t: «t : ?a → ?b»
proof
  show t: H.arr t
  proof −
    have V.arr t
      unfolding t-def
      using arr-mkarr(1) T'.transformation-axioms
            F'.simulation-axioms G'.simulation-axioms
            A.small-rts-axioms A.extensional-rts-axioms
            B.small-rts-axioms B.extensional-rts-axioms
      by blast
    thus ?thesis by auto
  qed
  show dom t = ?a
    using assms(3) a t f dom-char t-def by auto
  show cod t = ?b
    using assms(3) b t f cod-char t-def by auto
qed
have 1: arr t
  using t by auto
moreover have src t = f
proof −
  have 2: src t = mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 f ∘ UP_rts ?a)
    using t t-def mkarr-simps(3) A.small-rts-axioms A.extensional-rts-axioms
        B.small-rts-axioms B.extensional-rts-axioms T'.transformation-axioms
    by blast
  also have ... = f
  proof (intro arr-eqI)
    show mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 f ∘ UP_rts ?a) ≠ Null
      unfolding mkarr-def by blast
    show f ≠ Null
      using f H-arr-char by blast
    show Dom (mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 f ∘ UP_rts ?a)) =
         Dom f
      using f dom-char mkarr-def by auto
    show Cod (mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 f ∘ UP_rts ?a)) =
         Cod f
      using f cod-char mkarr-def by auto
    show Trn (mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 f ∘ UP_rts ?a)) =
         Trn f
    proof −
      have Trn (mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 f ∘ UP_rts ?a)) =
           AB.MkIde (DN_rts ?b ∘ cov-HOM 1 f ∘ UP_rts ?a)
        using mkarr-def by simp
      also have ... = Trn f
```

proof (*intro AB.arr-eqI*)
show *AB.arr* (*AB.MkIde* (DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*))
proof −
have *arr* (*src t*)
using *1 V.arr-src-iff-arr* **by** *blast*
thus *?thesis*
using *2 arr-char mkarr-def* **by** *auto*
qed
show *AB.arr* (*Trn f*)
using *f arr-char* [*of f*] *dom-char cod-char* **by** *auto*
show *AB.Dom* (*AB.MkIde* (DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*)) =
AB.Dom (*Trn f*)
proof −
have *AB.Dom* (*AB.MkIde* (DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*)) =
DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*
by *simp*
also have ... = *AB.Map* (*Trn f*)
using *assms f 0 UP-DN-naturality*(*4*) **by** *simp*
also have ... = *AB.Dom* (*Trn f*)
using *assms sta-char dom-char cod-char* $AB.ide\text{-}char_{ERTS}$
by *fastforce*
finally show *?thesis* **by** *blast*
qed
show *AB.Cod* (*AB.MkIde* (DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*)) =
AB.Cod (*Trn f*)
proof −
have *AB.Cod* (*AB.MkIde* (DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*)) =
DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*
by *simp*
also have ... = *AB.Map* (*Trn f*)
using *assms f 0 UP-DN-naturality*(*4*) **by** *simp*
also have ... = *AB.Cod* (*Trn f*)
using *assms sta-char dom-char cod-char* $AB.ide\text{-}char_{ERTS}$
by *fastforce*
finally show *?thesis* **by** *blast*
qed
show ⋀*x*. *A.ide x* ⟹
AB.Map
(*AB.MkIde* (DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*)) *x* =
AB.Map (*Trn f*) *x*
proof −
fix *x*
assume *x*: *A.ide x*
have *AB.Map*
(*AB.MkIde* (DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*)) *x* =
(DN_{rts} *?b* ∘ *cov-HOM* **1** *f* ∘ UP_{rts} *?a*) *x*
by *simp*
also have ... = *AB.Map* (*Trn f*) *x*
using *assms f 0 UP-DN-naturality*(*4*) [*of f*] **by** *simp*

```
        finally
        show AB.Map
              (AB.MkIde (DN_rts ?b ∘ cov-HOM 1 f ∘ UP_rts ?a)) x =
            AB.Map (Trn f) x
          by blast
      qed
    qed
    finally show ?thesis by blast
  qed
 qed
 finally show ?thesis by blast
qed
moreover have trg t = g
proof −
 have 2: trg t = mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a)
  using t t-def mkarr-simps(4) A.small-rts-axioms A.extensional-rts-axioms
      B.small-rts-axioms B.extensional-rts-axioms T'.transformation-axioms
  by blast
 also have ... = g
 proof (intro arr-eqI)
  show mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a) ≠ Null
   unfolding mkarr-def by blast
  show g ≠ Null
   using g H-arr-char by blast
  show Dom (mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a)) =
       Dom g
   using assms g dom-char mkarr-def by auto
 show Cod (mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a)) = Cod g
   using assms g cod-char mkarr-def by auto
 show Trn (mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a)) = Trn g
  proof −
   have 3: DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a = AB.Map (Trn g)
    using assms g 0 UP-DN-naturality(4)
    apply simp
    by presburger
   have Trn (mksta ?A ?B (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a)) =
        AB.MkIde (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a)
    using mkarr-def by simp
   also have ... = Trn g
   proof (intro AB.arr-eqI)
    show AB.arr (AB.MkIde (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a))
     using AB.ide-implies-arr G'.simulation-axioms by blast
    show AB.arr (Trn g)
     using assms g arr-char [of g] dom-char cod-char by auto
   show AB.Dom (AB.MkIde (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a)) =
        AB.Dom (Trn g)
     using assms 3 sta-char AB.ide-char_ERTS by simp
    show AB.Cod (AB.MkIde (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a)) =
        AB.Cod (Trn g)
```

```
          using assms 3 sta-char AB.ide-char_ERTS by simp
        show ⋀x. A.ide x ⟹
                 AB.Map
                   (AB.MkIde (DN_rts ?b ∘ cov-HOM 1 g ∘ UP_rts ?a)) x =
                 AB.Map (Trn g) x
          using 3 by simp
      qed
      finally show ?thesis by blast
    qed
  qed
  finally show ?thesis by blast
qed
moreover have cov-HOM 1 t = T
proof −
  have cov-HOM 1 t = UP_rts ?b ∘ Map t ∘ DN_rts ?a
  proof −
    have arr t ∧ dom f = dom t ∧ cod f = cod t
      using f t by auto
    thus ?thesis
      using UP-DN-naturality(3) [of t] by presburger
  qed
  also have ... = UP_rts ?b ∘ (DN_rts ?b ∘ T ∘ UP_rts ?a) ∘ DN_rts ?a
    unfolding t-def mkarr-def by simp
  also have ... = (UP_rts ?b ∘ DN_rts ?b) ∘ T ∘ (UP_rts ?a ∘ DN_rts ?a)

    using comp-assoc by smt
  also have ... = T ∘ (UP_rts ?a ∘ DN_rts ?a)
    using comp-identity-transformation
            [of HOM-1a.resid HOM-1b.resid - - T]
          T.transformation-axioms UP-DN-b.inv
    by auto
  also have ... = T
    using comp-transformation-identity
            [of HOM-1a.resid HOM-1b.resid - - T]
          T.transformation-axioms UP-DN-a.inv
    by auto
  finally show ?thesis by blast
qed
ultimately show ?thesis by blast
qed

lemma bij-HOM1-sta:
assumes obj a and obj b
shows bij-betw (cov-HOM 1) {f. «f : a →_sta b»}
       (Collect (simulation (HOM 1 a) (HOM 1 b)))
proof −
  interpret HOM-1a: sub-rts resid ‹λt. «t : 1 → a»›
    using assms(1) sub-rts-HOM by blast
  interpret HOM-1b: sub-rts resid ‹λt. «t : 1 → b»›
```

```
    using assms(2) sub-rts-HOM by blast
  have 1: cov-HOM 1 ∈ {f. «f : a →_sta b»}
                        → Collect (simulation (HOM 1 a) (HOM 1 b))
  proof
    fix f
    assume f: f ∈ {f. «f : a →_sta b»}
    have sta f ∧ dom f = a ∧ cod f = b
      using f by auto
    thus cov-HOM 1 f ∈ Collect (simulation (HOM 1 a) (HOM 1 b))
      using simulation-cov-HOM-sta [of 1 f] by blast
  qed
  show bij-betw (cov-HOM 1) {f. «f : a →_sta b»}
         (Collect (simulation (HOM 1 a) (HOM 1 b)))
  proof (unfold bij-betw-def, intro conjI)
    show inj-on (cov-HOM 1) {f. «f : a →_sta b»}
      using assms HOM1-faithful-for-sta [of - a b]
      unfolding inj-on-def
      by auto
    show cov-HOM 1 ` {f. «f : a →_sta b»} =
         Collect (simulation (HOM 1 a) (HOM 1 b))
    proof
      show cov-HOM 1 ` {f. «f : a →_sta b»}
             ⊆ Collect (simulation HOM-1a.resid HOM-1b.resid)
        using 1 by blast
      show Collect (simulation HOM-1a.resid HOM-1b.resid)
             ⊆ cov-HOM 1 ` {f. «f : a →_sta b»}
      proof
        fix F
        assume F: F ∈ Collect (simulation HOM-1a.resid HOM-1b.resid)
        obtain f where f: «f : a →_sta b» ∧ cov-HOM 1 f = F
          using assms F HOM1-full-for-sta [of a b F]
                HOM-1a.arr-char HOM-1b.arr-char
          by auto
        show F ∈ cov-HOM 1 ` {f. «f : a →_sta b»}
          using f by blast
      qed
    qed
  qed
qed

lemma bij-HOM1-arr:
assumes «f : a →_sta b» and «g : a →_sta b»
shows bij-betw (cov-HOM 1) {t. «t : f ⇒ g»}
        (Collect (transformation (HOM 1 a) (HOM 1 b)
          (cov-HOM 1 f) (cov-HOM 1 g)))
proof −
  interpret HOM-1a: sub-rts resid ‹λt. «t : 1 → a»›
    using assms(1) sub-rts-HOM by blast
  interpret HOM-1b: sub-rts resid ‹λt. «t : 1 → b»›
```

```
    using assms(2) sub-rts-HOM by blast
  have 1: cov-HOM 𝟭 ∈
          {t. «t : f ⇒ g»}
            → Collect (transformation
                       (HOM 𝟭 a) (HOM 𝟭 b) (cov-HOM 𝟭 f) (cov-HOM 𝟭 g))
  proof
    fix t
    assume t: t ∈ {t. «t : f ⇒ g»}
    thus cov-HOM 𝟭 t ∈ Collect (transformation (HOM 𝟭 a) (HOM 𝟭 b)
                                (cov-HOM 𝟭 f) (cov-HOM 𝟭 g))
      using assms(1) t transformation-cov-HOM-arr [of 𝟭 t] obj-one by auto
  qed
  show bij-betw (cov-HOM 𝟭) {t. «t : f ⇒ g»}
       (Collect (transformation (HOM 𝟭 a) (HOM 𝟭 b)
                 (cov-HOM 𝟭 f) (cov-HOM 𝟭 g)))
  proof (unfold bij-betw-def, intro conjI)
    show inj-on (cov-HOM 𝟭) {t. «t : f ⇒ g»}
      using assms HOM1-faithful-for-arr
      unfolding inj-on-def
      by auto
    show cov-HOM 𝟭 ‘ {t. «t : f ⇒ g»} =
         Collect (transformation HOM-1a.resid HOM-1b.resid
                  (cov-HOM 𝟭 f) (cov-HOM 𝟭 g))
    proof
      show cov-HOM 𝟭 ‘ {t. «t : f ⇒ g»} ⊆
           Collect (transformation HOM-1a.resid HOM-1b.resid
                    (cov-HOM 𝟭 f) (cov-HOM 𝟭 g))
        using 1 by blast
      show Collect (transformation HOM-1a.resid HOM-1b.resid
                    (cov-HOM 𝟭 f) (cov-HOM 𝟭 g)) ⊆
           cov-HOM 𝟭 ‘ {t. «t : f ⇒ g»}
      proof
        fix T
        assume T: T ∈ Collect (transformation HOM-1a.resid HOM-1b.resid
                               (cov-HOM 𝟭 f) (cov-HOM 𝟭 g))
        obtain t where t: «t : f ⇒ g» ∧ cov-HOM 𝟭 t = T
          using assms T HOM1-full-for-arr [of f g T] arr-char
                HOM-1a.arr-char HOM-1b.arr-char
          by blast
        show T ∈ cov-HOM 𝟭 ‘ {t. arr t ∧ src t = f ∧ trg t = g}
          using t by blast
      qed
    qed
  qed
qed
```

My original objective for the results in this section was to obtain a characterization up to equivalence of the RTS-category $\mathbf{RTS}^\dagger$ in terms of intrinsic notions that make sense for any RTS-category, and to carry out the proof

of cartesian closure using *HOM* **1** in place of *Dom*. This can probably be done, and I did push the idea through the construction of products, but for exponentials there were some technicalities that started to get messy and become distractions from the main things that I was trying to do. So I decided to leave this program for future work.

end

end

4.4 The Category of RTS's and Simulations

In this section, we show that the subcategory of $\mathbf{RTS}^\dagger$, comprised of the arrows that are identities with respect to the residuation, is also cartesian closed. In informal text, we will refer to this category as **RTS**. In a later section, we will show that the entire structure of the RTS-category $\mathbf{RTS}^\dagger$ is already determined by the ordinary subcategory **RTS**.

theory *RTSCat*
imports *Main RTSCatx EnrichedCategoryBasics.CartesianClosedMonoidalCategory*
begin

locale *rtscat* =
universe arr-type
for *arr-type* :: *'A itself*
begin

sublocale *One*: *one-arr-rts arr-type* **..**

interpretation *RTSx*: *rtscatx arr-type* **..**
interpretation *RTSx*: *elementary-category-with-binary-products*
RTSx.hcomp RTSx.p_0 *RTSx.*p_1
using *RTSx.extends-to-elementary-category-with-binary-products* **by** *blast*
interpretation RTS_S: *subcategory RTSx.hcomp RTSx.sta*
using *RTSx.dom.preserves-ide RTSx.cod.preserves-ide RTSx.sta-hcomp*
RTSx.arr-hcomp RTSx.H.seqI
by *unfold-locales auto*

type-synonym *'a arr* = *'a rtscatx.arr*

definition *comp* :: *'A arr comp* (**infixr** · *53*)
where *comp* ≡ *subcategory.comp RTSx.hcomp RTSx.sta*

sublocale *category comp*
unfolding *comp-def*
using RTS_S*.is-category* **by** *blast*

notation *in-hom* («- : - → -»)

lemma *ide-iff-RTS-obj*:
shows *ide a* $\longleftrightarrow$ *RTSx.obj a*
using *RTSx.obj-is-sta* $RTS_S.ideI_{SbC}$ $RTS_S.ide\text{-}char_{SbC}$ *comp-def* **by** *auto*

lemma *arr-iff-RTS-sta*:
shows *arr f* $\longleftrightarrow$ *RTSx.sta f*
by (*simp add*: $RTS_S.arr\text{-}char_{SbC}$ *comp-def*)

We want *rtscat* to stand on its own, so here we embark on an extended development designed to bootstrap away from dependence on the supporting locale *rtscatx*.

abbreviation *Obj*
where *Obj A* $\equiv$ *extensional-rts A* $\wedge$ *small-rts A*

definition *mkide* :: *'A resid* $\Rightarrow$ *'A arr*
where *mkide* $\equiv$ *RTSx.mkobj*

definition *mkarr* :: *'A resid* $\Rightarrow$ *'A resid* $\Rightarrow$ (*'A* $\Rightarrow$ *'A*) $\Rightarrow$ *'A arr*
where *mkarr* $\equiv$ *RTSx.mksta*

definition *Rts* :: *'A arr* $\Rightarrow$ *'A resid*
where *Rts* $\equiv$ *RTSx.Dom*

abbreviation *Dom* :: *'A arr* $\Rightarrow$ *'A resid*
where *Dom t* $\equiv$ *Rts* (*dom t*)

abbreviation *Cod* :: *'A arr* $\Rightarrow$ *'A resid*
where *Cod t* $\equiv$ *Rts* (*cod t*)

definition *Map* :: *'A arr* $\Rightarrow$ *'A* $\Rightarrow$ *'A*
where *Map* $\equiv$ *RTSx.Map*

lemma $ideD_{RTSC}$ [*intro*, *simp*]:
assumes *ide a*
shows *Obj* (*Rts a*)
using *assms Rts-def* $RTS_S.arr\text{-}char_{SbC}$ *ideD*(*1*) *comp-def* **by** *auto*

lemma *ide-mkide* [*intro*, *simp*]:
assumes *Obj A*
shows *ide* (*mkide A*)
using *assms comp-def mkide-def RTSx.obj-implies-sta RTSx.obj-mkobj*
$RTS_S.ideI_{SbC}$
by *simp*

lemma *Rts-mkide* [*simp*]:
shows *Rts* (*mkide A*) = *A*
by (*simp add*: *Rts-def mkide-def*)

lemma *mkide-Rts* [*simp*]:

assumes *ide a*
shows *mkide* (*Rts a*) = *a*
using *assms RTSx.bij-mkobj*(*4*) *Rts-def ide-iff-RTS-obj mkide-def*
by *force*

lemma *Dom-mkide* [*simp*]:
assumes *ide* (*mkide A*)
shows *Dom* (*mkide A*) = *A*
using *assms* **by** *force*

lemma *Cod-mkide* [*simp*]:
assumes *ide* (*mkide A*)
shows *Cod* (*mkide A*) = *A*
using *assms* **by** *force*

lemma *Map-mkide* [*simp*]:
assumes *ide* (*mkide A*)
shows *Map* (*mkide A*) = *I A*
using *assms mkide-def RTSx.mkobj-def Map-def RTSx.bij-mksta*(*3*)
RTSx.mkarr-def Rts-def ideD$_{RTSC}$
by *fastforce*

lemma *bij-mkide*:
shows *mkide* ∈ *Collect Obj* → *Collect ide*
and *Rts* ∈ *Collect ide* → *Collect Obj*
and *Rts* (*mkide A*) = *A*
and *ide a* ⟹ *mkide* (*Rts a*) = *a*
and *bij-betw mkide* (*Collect Obj*) (*Collect ide*)
and *bij-betw Rts* (*Collect ide*) (*Collect Obj*)
proof −
show *mkide* ∈ *Collect Obj* → *Collect ide* **by** *simp*
show *Rts* ∈ *Collect ide* → *Collect Obj* **by** *simp*
show *Rts* (*mkide A*) = *A* **by** *simp*
show *ide a* ⟹ *mkide* (*Rts a*) = *a* **by** *simp*
show *bij-betw mkide* (*Collect Obj*) (*Collect ide*)
unfolding *bij-betw-def*
apply *auto*[*1*]
apply (*metis RTSx.mkobj-simps*(*1*) *inj-on-inverseI mkide-def*)
by (*metis* (*no-types*, *lifting*) *CollectI ideD*$_{RTSC}$ *image-eqI mkide-Rts*)
show *bij-betw Rts* (*Collect ide*) (*Collect Obj*)
unfolding *bij-betw-def*
apply *auto*[*1*]
apply (*metis CollectD inj-onI mkide-Rts*)
using *image-iff* **by** *fastforce*
qed

lemma *arrD*:
assumes *arr f*
shows *Obj* (*Rts* (*dom f*)) **and** *Obj* (*Rts* (*cod f*))

and *simulation* (*Rts* (*dom f*)) (*Rts* (*cod f*)) (*Map f*)
proof −
interpret *A*: *extensional-rts* ‹*Rts* (*dom f*)›
by (*simp add*: *assms*)
interpret *A*: *small-rts* ‹*Rts* (*dom f*)›
by (*simp add*: *assms*)
interpret *B*: *extensional-rts* ‹*Rts* (*cod f*)›
by (*simp add*: *assms*)
interpret *B*: *small-rts* ‹*Rts* (*cod f*)›
by (*simp add*: *assms*)
interpret *AB*: *exponential-rts* ‹*Rts* (*dom f*)› ‹*Rts* (*cod f*)› **..**
have *1*: *mkarr* (*Rts* (*dom f*)) (*Rts* (*cod f*)) (*Map f*) = *f*
using *assms comp-def mkarr-def Rts-def RTSx.mkarr-def*
by (*metis* (*no-types*, *lifting*) *Map-def RTSx.Dom-cod RTSx.Dom-dom*
RTSx.Map-simps(*4*) *RTSx.V.ide-implies-arr RTSx.V.trg-ide*
RTSx.trg-simp RTS_S.*cod-char*$_{SbC}$ RTS_S.*dom-char*$_{SbC}$ *arr-iff-RTS-sta*)
show *Obj* (*Rts* (*dom f*))
using *A.extensional-rts-axioms A.small-rts-axioms* **by** *simp*
show *Obj* (*Rts* (*cod f*))
using *B.extensional-rts-axioms B.small-rts-axioms* **by** *simp*
show *simulation* (*Rts* (*dom f*)) (*Rts* (*cod f*)) (*Map f*)
using *assms 1 RTSx.sta-char RTSx.simulation-Map-sta*
RTS_S.*arr-char*$_{SbC}$ RTS_S.*cod-simp* RTS_S.*dom-simp comp-def*
by (*simp add*: *Map-def Rts-def*)
qed

lemma *arr-mkarr* [*intro*, *simp*]:
assumes *Obj A* **and** *Obj B* **and** *simulation A B F*
shows *arr* (*mkarr A B F*)
unfolding *mkarr-def comp-def*
using *assms* RTS_S.*arr-char*$_{SbC}$ **by** *blast*

lemma $arrI_{RTSC}$:
assumes *f* ∈ *mkarr A B* ‘ *Collect* (*simulation A B*)
and *Obj A* **and** *Obj B*
shows *arr f*
using *assms arr-mkarr* **by** *blast*

lemma *Dom-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F*)
shows *Dom* (*mkarr A B F*) = *A*
using *assms*
by (*simp add*: RTS_S.*arrE* RTS_S.*dom-simp Rts-def comp-def mkarr-def*)

lemma *Cod-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F*)
shows *Cod* (*mkarr A B F*) = *B*
using *assms*
by (*simp add*: RTS_S.*arrE* RTS_S.*cod-simp Rts-def comp-def mkarr-def*)

lemma *Map-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F*)
shows *Map* (*mkarr A B F*) = *F*
using *assms mkarr-def RTSx.mkarr-def*
by (*metis Cod-mkarr Dom-mkarr Map-def RTSx.bij-mksta*(*3*) *arrD*(*1*−*2*))

lemma *mkarr-Map* [*simp*]:
assumes *Obj A* **and** *Obj B* **and** *t* ∈ {*t*. «*t* : *mkide A* → *mkide B*»}
shows *mkarr A B* (*Map t*) = *t*
using *assms mkarr-def Map-def comp-def mkide-def*
by (*metis* (*no-types*, *lifting*) $RTS_S.arrE$ $RTS_S.in\text{-}hom\text{-}char_{SbC}$
RTSx.bij-mksta(*4*) *mem-Collect-eq*)

lemma *dom-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F*)
shows *dom* (*mkarr A B F*) = *mkide A*
using *assms*
by (*metis Dom-mkarr ide-dom mkide-Rts*)

lemma *cod-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F*)
shows *cod* (*mkarr A B F*) = *mkide B*
using *assms*
by (*metis Cod-mkarr ide-cod mkide-Rts*)

lemma *mkarr-in-hom* [*intro*]:
assumes *simulation A B F* **and** *Rts a* = *A* **and** *Rts b* = *B*
and *ide a* **and** *ide b*
shows «*mkarr A B F* : *a* → *b*»
using *assms* **by** *auto*

lemma *bij-mkarr*:
assumes *Obj A* **and** *Obj B*
shows *mkarr A B* ∈ *Collect* (*simulation A B*)
→ {*t*. «*t* : *mkide A* → *mkide B*»}
and *Map* ∈ {*t*. «*t* : *mkide A* → *mkide B*»} → *Collect* (*simulation A B*)
and *Map* (*mkarr A B F*) = *F*
and *t* ∈ {*t*. «*t* : *mkide A* → *mkide B*»} ⟹ *mkarr A B* (*Map t*) = *t*
and *bij-betw* (*mkarr A B*) (*Collect* (*simulation A B*))
{*t*. «*t* : *mkide A* → *mkide B*»}
and *bij-betw Map* {*t*. «*t* : *mkide A* → *mkide B*»}
(*Collect* (*simulation A B*))
proof −
show *1*: *mkarr A B* ∈ *Collect* (*simulation A B*) → *hom* (*mkide A*) (*mkide B*)
by (*simp add*: *assms in-homI*)
show *2*: *Map* ∈ {*t*. «*t* : *mkide A* → *mkide B*»} → *Collect* (*simulation A B*)
using *arrD*(*3*) **by** *fastforce*
show *3*: *Map* (*mkarr A B F*) = *F*

using *assms*
by (*metis Map-def RTSx.bij-mksta*(*3*) *mkarr-def*)
show *4*: $t \in \{t.\ «t : mkide\ A \to mkide\ B»\} \Longrightarrow mkarr\ A\ B\ (Map\ t) = t$
using *assms* **by** *auto*
show *bij-betw* (*mkarr A B*) (*Collect* (*simulation A B*))
$\{t.\ «t : mkide\ A \to mkide\ B»\}$
using *assms 1 2 3 4* **by** (*intro bij-betwI*) *auto*
show *bij-betw Map* $\{t.\ «t : mkide\ A \to mkide\ B»\}$ (*Collect* (*simulation A B*))
using *assms 1 2 3 4* **by** (*intro bij-betwI*) *auto*
qed

lemma *arr-eqI*:
assumes *par f g* **and** *Map f* = *Map g*
shows *f* = *g*
proof (*intro RTSx.arr-eqI*)
show *f* ≠ *RTSx.Null*
using *assms* RTS_S*.null-char RTSx.null-char not-arr-null comp-def* **by** *force*
show *g* ≠ *RTSx.Null*
using *assms* RTS_S*.null-char RTSx.null-char not-arr-null comp-def* **by** *force*
show *1*: *RTSx.Dom f* = *RTSx.Dom g*
using *assms comp-def*
by (*metis RTSx.Dom-dom RTSx.V.ide-implies-arr* RTS_S*.arrE* RTS_S*.dom-simp*)
show *2*: *RTSx.Cod f* = *RTSx.Cod g*
using *assms comp-def*
by (*metis RTSx.Dom-cod RTSx.V.ide-implies-arr* RTS_S*.arrE* RTS_S*.cod-char*$_{SbC}$)
interpret *A*: *extensional-rts* ‹*RTSx.Dom f*›
using *assms* RTS_S*.arrE comp-def* **by** *auto*
interpret *B*: *extensional-rts* ‹*RTSx.Cod f*›
using *assms* RTS_S*.arrE comp-def* **by** *auto*
interpret *AB*: *exponential-rts* ‹*RTSx.Dom f*› ‹*RTSx.Cod f*› **..**
show *RTSx.Trn f* = *RTSx.Trn g*
using *assms 1 2 comp-def Map-def*
by (*metis* (*no-types*, *lifting*) *AB.ide-implies-arr RTSx.Map-simps*(*4*)
RTSx.V.trg-ide RTSx.arr-char RTSx.sta-char RTSx.trg-simp
RTS_S*.arr-char*$_{SbC}$)
qed

lemma *Map-ide*:
assumes *ide a*
shows *Map a* = *I* (*Rts a*)
using *assms Map-mkide* [*of Rts a*] **by** *simp*

lemma *Map-comp*:
assumes *seq g f*
shows *Map* (*g* · *f*) = *Map g* ∘ *Map f*
using *assms Map-def comp-def RTSx.Map-hcomp* RTS_S*.arr-char*$_{SbC}$ RTS_S*.comp-def*
by *auto*

lemma *comp-mkarr*:

assumes *arr* (*mkarr A B F*) **and** *arr* (*mkarr B C G*)
shows *mkarr B C G* · *mkarr A B F* = *mkarr A C* (*G* ∘ *F*)
proof (*intro arr-eqI*)
have *1*: *arr* (*mkarr A C* (*G* ∘ *F*))
using *assms arrD simulation-comp bij-mkarr*(*3*) *Cod-mkarr Dom-mkarr arr-mkarr*
by *metis*
show *par* (*mkarr B C G* · *mkarr A B F*) (*mkarr A C* (*G* ∘ *F*))
using *assms 1* **by** *auto*
show *Map* (*mkarr B C G* · *mkarr A B F*) = *Map* (*mkarr A C* (*G* ∘ *F*))
using *assms 1 Map-comp Map-mkarr* **by** *auto*
qed

lemma *iso-char*:
shows *iso t* ⟷ *arr t* ∧ *invertible-simulation* (*Dom t*) (*Cod t*) (*Map t*)
using *Map-def comp-def Rts-def RTSx.iso-char RTSx.iso-implies-sta*
by (*metis* (*no-types*, *lifting*) *RTSx.Dom-cod RTSx.Dom-dom RTSx.H.iso-inv-iso RTSx.Map-simps*(*3*−*4*) *RTSx.V.ide-iff-src-self RTSx.V.ide-implies-arr RTSx.V.trg-ide* $RTS_S.arr\text{-}char_{SbC}$ $RTS_S.cod\text{-}simp$ $RTS_S.dom\text{-}char_{SbC}$ $RTS_S.iso\text{-}char_{SbC}$)

lemma *isomorphic-char*:
shows *isomorphic a b* ⟷
ide a ∧ *ide b* ∧ (∃ *F*. *invertible-simulation* (*Rts a*) (*Rts b*) *F*)
proof
show *isomorphic a b* ⟹
ide a ∧ *ide b* ∧ (∃ *F*. *invertible-simulation* (*Rts a*) (*Rts b*) *F*)
by (*metis* (*no-types*, *lifting*) *ide-cod ide-dom in-homE iso-char isomorphicE*)
show *ide a* ∧ *ide b* ∧ (∃ *F*. *invertible-simulation* (*Rts a*) (*Rts b*) *F*)
⟹ *isomorphic a b*
by (*metis arr-mkarr cod-mkarr dom-mkarr* $ideD_{RTSC}$ *isomorphic-def iso-char invertible-simulation.axioms*(*1*) *mkarr-in-hom mkide-Rts Map-mkarr*)
qed

4.4.1 Terminal Object

definition *one* (**1**)
where *one* ≡ *RTSx.one*
no-notation *RTSx.one* (**1**)

definition *trm*
where *trm* ≡ *RTSx.trm*

lemma *Rts-one* [*simp*]:
shows *Rts* **1** = *One.resid*
unfolding *one-def Rts-def RTSx.one-def* **by** *simp*

lemma *mkide-One* [*simp*]:
shows *mkide One.resid* = **1**
unfolding *one-def mkide-def RTSx.one-def* **by** *simp*

sublocale *elementary-category-with-terminal-object comp one trm*
proof
show *ide* **1**
unfolding *one-def*
using *RTSx.obj-one RTSx.obj-is-sta* $RTS_S.ideI_{SbC}$ *comp-def* **by** *force*
show $\bigwedge a.$ *ide a* $\Longrightarrow$ «*trm a* : *a* $\rightarrow$ **1**»
using $RTS_S.arr\text{-}char_{SbC}$ $RTS_S.ide\text{-}char_{SbC}$ $RTS_S.in\text{-}hom\text{-}char_{SbC}$ *comp-def*
by (*metis* (*no-types*, *lifting*) *RTSx.trm-in-hom RTSx.trm-simps'*(*6*)
‹*ide* **1**› *one-def trm-def*)
show $\bigwedge a\ f.$ ⟦*ide a*; «*f* : *a* $\rightarrow$ **1**»⟧ $\Longrightarrow$ *f* = *trm a*
by (*metis RTSx.trm-eqI* $RTS_S.ide\text{-}char_{SbC}$ $RTS_S.in\text{-}hom\text{-}char_{SbC}$ *comp-def*
one-def trm-def)
qed

lemma *Map-trm*:
assumes *ide a*
shows *Map* (*trm a*) = *constant-simulation.map* (*Rts a*) *One.resid One.the-arr*
unfolding *Map-def trm-def Rts-def*
using *assms RTSx.Map-trm* $RTS_S.ide\text{-}char_{SbC}$ *comp-def* **by** *fastforce*

lemma *terminal-char*:
shows *terminal x* $\longleftrightarrow$ *ide x* $\wedge$ ($\exists!t.$ *residuation.arr* (*Rts x*) *t*)
proof −
have $\bigwedge x.$ *terminal x* $\longleftrightarrow$ *RTSx.H.terminal x*
proof −
fix *x*
have *1*: *terminal x* $\longleftrightarrow$ *isomorphic x one*
using *terminal-one terminal-objs-isomorphic*
by (*meson isomorphic-symmetric isomorphic-to-terminal-is-terminal*)
also have ... $\longleftrightarrow$ *RTSx.obj x* $\wedge$ *isomorphic-rts* (*RTSx.Dom x*) *One.resid*
using *Rts-one ide-one isomorphic-char Rts-def ide-iff-RTS-obj*
isomorphic-rts-def [*of Rts x One.resid*]
invertible-simulation-def' [*of Rts x Rts* **1**]
by *auto*
also have ... $\longleftrightarrow$ *RTSx.H.terminal x*
by (*metis* (*mono-tags*, *lifting*) *1* $RTS_S.arrI_{SbC}$ $RTS_S.in\text{-}hom\text{-}char_{SbC}$
$RTS_S.iso\text{-}char_{SbC}$ $RTS_S.isomorphic\text{-}def$ *RTSx.H.iso-inv-iso*
RTSx.H.isomorphic-def RTSx.H.isomorphic-symmetric
RTSx.H.isomorphic-to-terminal-is-terminal RTSx.H.terminal-def
RTSx.iso-implies-sta RTSx.obj-implies-sta RTSx.terminal-one
calculation comp-def one-def RTSx.H.terminal-objs-isomorphic)
finally show *terminal x* $\longleftrightarrow$ *RTSx.H.terminal x* **by** *blast*
qed
thus *?thesis*
using *RTSx.terminal-char Rts-def ide-iff-RTS-obj* **by** *presburger*

qed

4.4.2 Products

definition $p_0 :: 'A\ arr \Rightarrow 'A\ arr \Rightarrow 'A\ arr$
where $p_0 \equiv RTSx.p_0$

definition $p_1 :: 'A\ arr \Rightarrow 'A\ arr \Rightarrow 'A\ arr$
where $p_1 \equiv RTSx.p_1$

no-notation $RTSx.p_0$ $(\mathfrak{p}_0[\text{-}, \text{-}])$
no-notation $RTSx.p_1$ $(\mathfrak{p}_1[\text{-}, \text{-}])$
notation p_0 $(\mathfrak{p}_0[\text{-}, \text{-}])$
notation p_1 $(\mathfrak{p}_1[\text{-}, \text{-}])$

sublocale *elementary-cartesian-category comp* p_0 p_1 *one trm*
proof
fix *a b*
assume *a*: *ide a* **and** *b*: *ide b*
show *1*: *span* $\mathfrak{p}_1[a, b]$ $\mathfrak{p}_0[a, b]$
using *a b RTSx.sta-*p_0 *RTSx.sta-*p_1 $RTS_S.arr\text{-}char_{S\flat C}$ $RTS_S.dom\text{-}simp$ $RTS_S.ide\text{-}char_{S\flat C}$
by (*simp add*: *comp-def* p_0*-def* p_1*-def*)
show *cod* $\mathfrak{p}_0[a, b] = b$
using *a b 1* $RTS_S.cod\text{-}simp$ *RTSx.cod-pr0 ide-iff-RTS-obj comp-def* p_0*-def*
by *metis*
show *cod* $\mathfrak{p}_1[a, b] = a$
using *a b 1* $RTS_S.cod\text{-}simp$ *RTSx.cod-pr1 ide-iff-RTS-obj comp-def* p_1*-def*
by *metis*
next
fix *f g*
assume *fg*: *span f g*
have *a*: *ide* (*cod f*) **and** *b*: *ide* (*cod g*)
using *fg* **by** *auto*
have *1*: *RTSx.H.span f g*
using *fg* $RTS_S.arrE$ $RTS_S.dom\text{-}simp$ *comp-def* **by** *force*
have *2*: *RTSx.cod f* = *cod f* ∧ *RTSx.cod g* = *cod g*
using *fg* **by** (*simp add*: $RTS_S.cod\text{-}char_{S\flat C}$ *comp-def*)
show ∃!*l*. $\mathfrak{p}_1[cod\ f, cod\ g] \cdot l = f \wedge \mathfrak{p}_0[cod\ f, cod\ g] \cdot l = g$
proof −
have ∃ *l*. $\mathfrak{p}_1[cod\ f, cod\ g] \cdot l = f \wedge \mathfrak{p}_0[cod\ f, cod\ g] \cdot l = g$
proof −
let *?l* = *RTSx.tuple f g*
have $\mathfrak{p}_1[cod\ f, cod\ g] \cdot ?l = f \wedge \mathfrak{p}_0[cod\ f, cod\ g] \cdot ?l = g$
using *a b fg 1 2 RTSx.sta-*p_0 *RTSx.sta-*p_1 *arr-iff-RTS-sta ide-iff-RTS-obj*
by (*simp add*: *comp-def* $RTS_S.comp\text{-}def$ p_0*-def* p_1*-def*)

thus $\exists\, l.\ \mathfrak{p}_1[cod\ f,\ cod\ g] \cdot l = f \wedge \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot l = g$
by *blast*
qed
moreover
have $\bigwedge l\ l'.\ [\![\mathfrak{p}_1[cod\ f,\ cod\ g] \cdot l = f \wedge \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot l = g;$
$\mathfrak{p}_1[cod\ f,\ cod\ g] \cdot l' = f \wedge \mathfrak{p}_0[cod\ f,\ cod\ g] \cdot l' = g]\!] \Longrightarrow l' = l$
by (*metis* (*no-types*, *lifting*) *1 RTSx.tuple-eqI* RTS_S*.comp-simp a b fg ide-iff-RTS-obj comp-def* p_0*-def* p_1*-def*)
ultimately show *?thesis* **by** *blast*
qed
qed

notation *prod* (**infixr** $\otimes$ *51*)
notation *tuple* ($\langle$-, -$\rangle$)
notation *dup* (d[-])
notation *assoc* (a[-, -, -])
notation *assoc′* (a^{-1}[-, -, -])
notation *lunit* (l[-])
notation *lunit′* (l^{-1}[-])
notation *runit* (r[-])
notation *runit′* (r^{-1}[-])

lemma *tuple-char*:
shows *tuple* = ($\lambda f\ g.$ *if span f g then RTSx.tuple f g else null*)
proof −
have $\bigwedge f\ g.\ \langle f,\ g\rangle$ = (*if span f g then RTSx.tuple f g else null*)
proof −
fix *f g*
show $\langle f,\ g\rangle$ = (*if span f g then RTSx.tuple f g else null*)
proof (*cases span f g*)
case *True*
have *RTSx.tuple f g* = $\langle f,\ g\rangle$
by (*metis* (*no-types*, *lifting*) *RTSx.tuple-pr-arr* RTS_S*.comp-simp* RTS_S*.seq-char*$_{SbC}$ *True ide-cod ide-iff-RTS-obj comp-def* p_0*-def* p_1*-def tuple-eqI universal*)
thus $\langle f,\ g\rangle$ = (*if span f g then RTSx.tuple f g else null*)
using *True* **by** *auto*
next
case *False*
show $\langle f,\ g\rangle$ = (*if span f g then RTSx.tuple f g else null*)
using *False tuple-ext* **by** *auto*
qed
qed
thus *?thesis* **by** *blast*
qed

lemma *prod-char*:
shows ($\otimes$) = ($\lambda f\ g.$ *if arr f* $\wedge$ *arr g then RTSx.prod f g else null*)

```
proof −
  have ⋀f g. f ⊗ g = (if arr f ∧ arr g then RTSx.prod f g else null)
  proof −
    fix f g
    have ¬ arr f ⟹ f ⊗ g = (if arr f ∧ arr g then RTSx.prod f g else null)
      using prod-def tuple-def by auto
    moreover
    have ¬ arr g ⟹ f ⊗ g = (if arr f ∧ arr g then RTSx.prod f g else null)
      using prod-def tuple-def by auto
    moreover have ⟦arr f; arr g⟧
                    ⟹ f ⊗ g = (if arr f ∧ arr g then RTSx.prod f g else null)
    proof −
      assume f: arr f and g: arr g
      have f ⊗ g = ⟨f · 𝔭₁[dom f, dom g], g · 𝔭₀[dom f, dom g]⟩
        unfolding prod-def by simp
      also have ... = ⟨f · 𝔭₁[RTSx.dom f, RTSx.dom g],
                       g · 𝔭₀[RTSx.dom f, RTSx.dom g]⟩
        using f g RTS_S.dom-char_SbC comp-def by force
      also have ... = ⟨RTSx.hcomp f 𝔭₁[RTSx.dom f, RTSx.dom g],
                       RTSx.hcomp g 𝔭₀[RTSx.dom f, RTSx.dom g]⟩
        using f g RTS_S.comp-char RTS_S.arr-char_SbC
        by (metis (no-types, lifting) RTS_S.null-char calculation
            comp-def not-arr-null prod-simps(1) tuple-ext)
      also have ... = RTSx.tuple
                        (RTSx.hcomp f 𝔭₁[RTSx.dom f, RTSx.dom g])
                        (RTSx.hcomp g 𝔭₀[RTSx.dom f, RTSx.dom g])
        by (metis (no-types, lifting) calculation f g not-arr-null
            prod-simps(1) tuple-char)
      also have ... = RTSx.prod f g
        using RTSx.prod-def by (simp add: p₀-def p₁-def)
      finally show ?thesis
        using f g by auto
    qed
    ultimately show f ⊗ g = (if arr f ∧ arr g then RTSx.prod f g else null)
      by blast
  qed
  thus ?thesis by blast
qed

definition Pack :: 'A arr ⇒ 'A arr ⇒ 'A × 'A ⇒ 'A
where Pack ≡ RTSx.Pack

definition Unpack :: 'A arr ⇒ 'A arr ⇒ 'A ⇒ 'A × 'A
where Unpack ≡ RTSx.Unpack

lemma inverse-simulations-Pack-Unpack:
assumes ide a and ide b
shows inverse-simulations (Rts (a ⊗ b)) (product-rts.resid (Rts a) (Rts b))
        (Pack a b) (Unpack a b)
```

using *Pack-def RTSx.inverse-simulations-Pack-Unpack Rts-def Unpack-def assms*(1−2) *ide-iff-RTS-obj prod-char*
by *force*

lemma *simulation-Pack*:
assumes *ide a* **and** *ide b*
shows *simulation*
(*product-rts.resid* (*Rts a*) (*Rts b*)) (*Rts* (*a* ⊗ *b*)) (*Pack a b*)
using *assms inverse-simulations-Pack-Unpack inverse-simulations-def*
by *fast*

lemma *simulation-Unpack*:
assumes *ide a* **and** *ide b*
shows *simulation*
(*Rts* (*a* ⊗ *b*)) (*product-rts.resid* (*Rts a*) (*Rts b*)) (*Unpack a b*)
using *assms inverse-simulations-Pack-Unpack inverse-simulations-def*
by *fast*

lemma *Pack-o-Unpack*:
assumes *ide a* **and** *ide b*
shows *Pack a b* ∘ *Unpack a b* = *I* (*Rts* (*a* ⊗ *b*))
unfolding *Pack-def Unpack-def Rts-def*
using *assms RTSx.Pack-o-Unpack ide-iff-RTS-obj prod-char* **by** *auto*

lemma *Unpack-o-Pack*:
assumes *ide a* **and** *ide b*
shows *Unpack a b* ∘ *Pack a b* = *I* (*product-rts.resid* (*Rts a*) (*Rts b*))
unfolding *Pack-def Unpack-def Rts-def*
using *assms RTSx.Unpack-o-Pack ide-iff-RTS-obj prod-char* **by** *auto*

lemma *Pack-Unpack* [*simp*]:
assumes *ide a* **and** *ide b*
and *residuation.arr* (*Rts* (*a* ⊗ *b*)) *t*
shows *Pack a b* (*Unpack a b t*) = *t*
using *assms* **by** (*metis Pack-o-Unpack comp-apply*)

lemma *Unpack-Pack* [*simp*]:
assumes *ide a* **and** *ide b*
and *residuation.arr* (*product-rts.resid* (*Rts a*) (*Rts b*)) *t*
shows *Unpack a b* (*Pack a b t*) = *t*
using *assms* **by** (*metis Unpack-o-Pack comp-apply*)

lemma *Map-*p_0:
assumes *ide a* **and** *ide b*
shows *Map* $\mathfrak{p}_0[a, b]$ = *product-rts.P*$_0$ (*Rts a*) (*Rts b*) ∘ *Unpack a b*
unfolding *Map-def* p_0*-def Unpack-def*
using *assms RTSx.Map-*p_0 *Rts-def ide-iff-RTS-obj* **by** *auto*

lemma *Map-*p_1:

assumes *ide a* **and** *ide b*
shows *Map* $\mathfrak{p}_1[a,\ b]$ = *product-rts.*P_1 (*Rts a*) (*Rts b*) ∘ *Unpack a b*
unfolding *Map-def* p_1*-def Unpack-def*
using *assms RTSx.Map-*p_1 *Rts-def ide-iff-RTS-obj* **by** *auto*

lemma *Map-tuple*:
assumes «$f : x \to a$» **and** «$g : x \to b$»
shows *Map* ⟨$f,\ g$⟩ = *Pack a b* ∘ ⟨⟨*Map f*, *Map g*⟩⟩
unfolding *Map-def Pack-def comp-def*
using *assms RTSx.Map-tuple*
by (*metis* (*no-types*, *lifting*) RTS_S*.in-hom-char*$_{SbC}$ *in-homE comp-def tuple-char*)

corollary *Map-dup*:
assumes *ide a*
shows *Map* $\mathrm{d}[a]$ = *Pack a a* ∘ ⟨⟨*Map a*, *Map a*⟩⟩
using *assms Map-tuple ide-in-hom* **by** *blast*

proposition *Map-lunit*:
assumes *ide a*
shows *Map* $\mathrm{l}[a]$ = *product-rts.*P_0 (*Rts* **1**) (*Rts a*) ∘ *Unpack* **1** *a*
using *assms Map-*p_0 *ide-one* **by** *auto*

proposition *Map-runit*:
assumes *ide a*
shows *Map* $\mathrm{r}[a]$ = *product-rts.*P_1 (*Rts a*) (*Rts* **1**) ∘ *Unpack a* **1**
using *assms Map-*p_1 *ide-one* **by** *auto*

lemma *assoc-expansion*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows $\mathrm{a}[a,\ b,\ c]$ =
⟨$\mathfrak{p}_1[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c]$, ⟨$\mathfrak{p}_0[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c]$, $\mathfrak{p}_0[a \otimes b,\ c]$⟩ ⟩
using *assms RTSx.assoc-expansion assoc-def* p_0*-def* p_1*-def* **by** *simp*

lemma *Unpack-naturality*:
assumes *arr f* **and** *arr g*
shows *Unpack* (*cod f*) (*cod g*) ∘ *Map* (*f* ⊗ *g*) =
product-simulation.map (*Rts* (*dom f*)) (*Rts* (*dom g*)) (*Map f*) (*Map g*) ∘
Unpack (*dom f*) (*dom g*)
proof −
interpret *Dom-f*: *extensional-rts* ‹*Rts* (*dom f*)›
by (*simp add*: *assms*(*1*))
interpret *Dom-g*: *extensional-rts* ‹*Rts* (*dom g*)›
by (*simp add*: *assms*(*2*))
interpret *Cod-f*: *extensional-rts* ‹*Rts* (*cod f*)›
by (*simp add*: *assms*(*1*))
interpret *Cod-g*: *extensional-rts* ‹*Rts* (*cod g*)›
by (*simp add*: *assms*(*2*))
interpret *Dom*: *product-rts* ‹*Rts* (*dom f*)› ‹*Rts* (*dom g*)› **..**

interpret *Cod*: *product-rts* ‹*Rts* (*cod f*)› ‹*Rts* (*cod g*)› **..**
interpret *Dom*: *extensional-rts Dom.resid*
using *Dom.preserves-extensional-rts Dom-f.extensional-rts-axioms*
Dom-g.extensional-rts-axioms
by *simp*
interpret *Unpack*: *simulation*
‹*Rts* (*dom f* ⊗ *dom g*)› *Dom.resid* ‹*Unpack* (*dom f*) (*dom g*)›
using *assms simulation-Unpack* [*of dom f dom g*] **by** *simp*
interpret *Unpack.A*: *extensional-rts* ‹*Rts* (*dom f* ⊗ *dom g*)›
by (*simp add*: *assms*)
interpret *Unpack*: *simulation-as-transformation*
‹*Rts* (*dom f* ⊗ *dom g*)› *Dom.resid* ‹*Unpack* (*dom f*) (*dom g*)›
..
have *Unpack* (*cod f*) (*cod g*) ∘ *Map* (*f* ⊗ *g*) =
Unpack (*cod f*) (*cod g*) ∘
Map ⟨*f* · 𝔭₁[*dom f*, *dom g*], *g* · 𝔭₀[*dom f*, *dom g*]⟩
using *assms prod-def* **by** *force*
also have ... = *Unpack* (*cod f*) (*cod g*) ∘
Pack (*cod f*) (*cod g*) ∘
(*Cod.tuple* (*Map* (*f* · 𝔭₁[*dom f*, *dom g*]))
(*Map* (*g* · 𝔭₀[*dom f*, *dom g*])))
using *assms Map-tuple*
by (*metis* (*no-types*, *lifting*) *Fun.comp-assoc cod-pr0 cod-pr1*
comp-in-homI′ ide-dom pr-simps(*1*−*2*,*4*−*5*))
also have ... = *I* (*Cod.resid*) ∘
Cod.tuple (*Map* (*f* · 𝔭₁[*dom f*, *dom g*]))
(*Map* (*g* · 𝔭₀[*dom f*, *dom g*]))
using *assms Unpack-o-Pack* **by** *auto*
also have ... = *Cod.tuple* (*Map* (*f* · 𝔭₁[*dom f*, *dom g*]))
(*Map* (*g* · 𝔭₀[*dom f*, *dom g*]))
proof −
have *simulation* (*Rts* (*dom f* ⊗ *dom g*)) *Cod.resid*
(*Cod.tuple*
(*Map* (*f* · 𝔭₁[*dom f*, *dom g*]))
(*Map* (*g* · 𝔭₀[*dom f*, *dom g*])))
by (*metis arrD*(*3*) *assms*(*1*−*2*) *cod-comp cod-pr0 cod-pr1 dom-comp*
ide-dom pr-simps(*1*−*2*,*4*−*5*) *seqI simulation-tuple*)
thus *?thesis*
using *assms comp-identity-simulation* **by** *blast*
qed
also have ... = *Cod.tuple*
(*Map f* ∘ (*Dom.P*₁ ∘ *Unpack* (*dom f*) (*dom g*)))
(*Map g* ∘ (*Dom.P*₀ ∘ *Unpack* (*dom f*) (*dom g*)))
using *assms Map-comp Map-p*₁ *Map-p*₀ **by** *auto*
also have ... = *product-simulation.map* (*Rts f*) (*Rts g*) (*Map f*) (*Map g*) ∘
Dom.tuple
(*Dom.P*₁ ∘ *Unpack* (*dom f*) (*dom g*))
(*Dom.P*₀ ∘ *Unpack* (*dom f*) (*dom g*))
proof −

```
    interpret P1oUnpack: simulation ‹Rts (dom f ⊗ dom g)› ‹Rts (dom f)›
                        ‹Dom.P1 ∘ Unpack (dom f) (dom g)›
      using assms Dom.P1-is-simulation simulation-comp
            simulation-Unpack [of dom f dom g]
      by auto
    interpret P1oUnpack: simulation-as-transformation
                        ‹Rts (dom f ⊗ dom g)› ‹Rts (dom f)›
                        ‹Dom.P1 ∘ Unpack (dom f) (dom g)›
      ..
    interpret P0oUnpack: simulation ‹Rts (dom f ⊗ dom g)› ‹Rts (dom g)›
                        ‹Dom.P0 ∘ Unpack (dom f) (dom g)›
      using assms Dom.P0-is-simulation simulation-comp
            simulation-Unpack [of dom f dom g]
      by auto
    interpret P0oUnpack: simulation-as-transformation
                        ‹Rts (dom f ⊗ dom g)› ‹Rts (dom g)›
                        ‹Dom.P0 ∘ Unpack (dom f) (dom g)›
      ..
    show ?thesis
      by (metis (no-types, lifting) P0oUnpack.transformation-axioms
      P1oUnpack.transformation-axioms RTSx.Dom-dom RTSx.V.ide-implies-arr
          RTS_S.dom-char_SbC Rts-def arrD(3) arr-iff-RTS-sta assms(1−2)
          comp-product-simulation-tuple2 comp-def)
  qed
  also have ... = product-simulation.map
                    (Rts (dom f)) (Rts (dom g)) (Map f) (Map g) ∘
                      Unpack (dom f) (dom g)
    using assms Unpack.simulation-axioms Dom.tuple-proj RTSx.arr-char
    by (metis (no-types, lifting) RTSx.Dom-dom RTSx.V.ide-implies-arr
        RTS_S.arrE RTS_S.dom-char_SbC Rts-def comp-def)
  finally show ?thesis by blast
qed

lemma Map-prod:
assumes arr f and arr g
shows Map (f ⊗ g) =
      Pack (cod f) (cod g) ∘
       product-simulation.map (Rts (dom f)) (Rts (dom g)) (Map f) (Map g) ∘
         Unpack (dom f) (dom g)
proof −
  have Map (f ⊗ g) =
       Pack (cod f) (cod g) ∘ (Unpack (cod f) (cod g) ∘ Map (f ⊗ g))
  proof −
    have Map (f ⊗ g) =
         (Pack (cod f) (cod g) ∘ (Unpack (cod f) (cod g)) ∘ Map (f ⊗ g))
      using assms Pack-o-Unpack [of cod f cod g] arrD(3) [of f ⊗ g]
      by (simp add: comp-identity-simulation)
    thus ?thesis
      using Fun.comp-assoc by metis
```

```
  qed
  also have ... =
          Pack (cod f) (cod g) ∘
           (product-simulation.map
              (Rts (dom f)) (Rts (dom g)) (Map f) (Map g) ∘
              Unpack (dom f) (dom g))
    using assms Unpack-naturality [of f g] by auto
  also have ... = Pack (cod f) (cod g) ∘
                  product-simulation.map
                    (Rts (dom f)) (Rts (dom g)) (Map f) (Map g) ∘
                    Unpack (dom f) (dom g)
    by auto
  finally show ?thesis by blast
qed
```

4.4.3 Exponentials

```
definition exp :: 'A arr ⇒ 'A arr ⇒ 'A arr
where exp ≡ RTSx.exp

definition eval :: 'A arr ⇒ 'A arr ⇒ 'A arr
where eval ≡ RTSx.eval

definition curry :: 'A arr ⇒ 'A arr ⇒ 'A arr ⇒ 'A arr ⇒ 'A arr
where curry ≡ RTSx.curry

definition uncurry :: 'A arr ⇒ 'A arr ⇒ 'A arr ⇒ 'A arr ⇒ 'A arr
where uncurry ≡ RTSx.uncurry

definition Func :: 'A arr ⇒ 'A arr ⇒ 'A ⇒ ('A, 'A) exponential-rts.arr
where Func ≡ RTSx.Func

definition Unfunc :: 'A arr ⇒ 'A arr ⇒ ('A, 'A) exponential-rts.arr ⇒ 'A
where Unfunc ≡ RTSx.Unfunc

lemma inverse-simulations-Func-Unfunc:
assumes ide b and ide c
shows inverse-simulations
        (exponential-rts.resid (Rts b) (Rts c)) (Rts (exp b c))
        (Func b c) (Unfunc b c)
  unfolding Func-def Unfunc-def Rts-def exp-def
  using assms RTSx.inverse-simulations-Func-Unfunc ide-iff-RTS-obj by blast

lemma simulation-Func:
assumes ide b and ide c
shows simulation
        (Rts (exp b c)) (exponential-rts.resid (Rts b) (Rts c)) (Func b c)
  using assms inverse-simulations-Func-Unfunc inverse-simulations-def
  by fast
```

lemma *simulation-Unfunc*:
assumes *ide b* **and** *ide c*
shows *simulation*
(*exponential-rts.resid* (*Rts b*) (*Rts c*)) (*Rts* (*exp b c*)) (*Unfunc b c*)
using *assms inverse-simulations-Func-Unfunc inverse-simulations-def*
by *fast*

lemma *Func-o-Unfunc*:
assumes *ide b* **and** *ide c*
shows *Func b c* ∘ *Unfunc b c* = *I* (*exponential-rts.resid* (*Rts b*) (*Rts c*))
using *assms*
by (*meson inverse-simulations.inv′ inverse-simulations-Func-Unfunc*)

lemma *Unfunc-o-Func*:
assumes *ide b* **and** *ide c*
shows *Unfunc b c* ∘ *Func b c* = *I* (*Rts* (*exp b c*))
using *assms*
by (*meson inverse-simulations.inv inverse-simulations-Func-Unfunc*)

lemma *Func-Unfunc* [*simp*]:
assumes *ide b* **and** *ide c*
and *residuation.arr* (*exponential-rts.resid* (*Rts b*) (*Rts c*)) *t*
shows *Func b c* (*Unfunc b c t*) = *t*
using *assms*
by (*meson inverse-simulations.inv′-simp inverse-simulations-Func-Unfunc*)

lemma *Unfunc-Func* [*simp*]:
assumes *ide b* **and** *ide c*
and *residuation.arr* (*Rts* (*exp b c*)) *t*
shows *Unfunc b c* (*Func b c t*) = *t*
using *assms*
by (*meson inverse-simulations.inv-simp inverse-simulations-Func-Unfunc*)

lemma *Map-eval*:
assumes *ide b* **and** *ide c*
shows *Map* (*eval b c*) =
evaluation-map.map (*Rts b*) (*Rts c*) ∘
product-simulation.map (*Rts* (*exp b c*)) (*Rts b*) (*Func b c*) (*I* (*Rts b*)) ∘
Unpack (*exp b c*) *b*
unfolding *Map-def eval-def Rts-def exp-def Func-def Unpack-def*
using *assms RTSx.Map-eval* [*of b c*] *ide-iff-RTS-obj* **by** *force*

lemma *Map-curry*:
assumes *ide a* **and** *ide b* **and** «*f* : *a* ⊗ *b* → *c*»
shows *Map* (*curry a b c f*) =
Unfunc b c ∘
Currying.Curry3 (*Rts a*) (*Rts b*) (*Rts c*) (*Map f* ∘ *Pack a b*)
unfolding *Map-def curry-def Unfunc-def Rts-def Pack-def*

using *assms RTSx.Map-curry* [*of a b c f*] *ide-iff-RTS-obj arr-iff-RTS-sta*
by (*metis* (*no-types*, *lifting*) *RTSx.H.category-axioms RTSx.Map-simps*(*3*−*4*)
RTSx.V.ide-iff-src-self RTSx.V.trg-ide RTSx.arr-coincidence$_{CRC}$
RTS$_S$*.ide-cod RTS*$_S$*.in-hom-char*$_{SbC}$ *category.in-homE category-axioms*
comp-def)

lemma *Map-uncurry*:
assumes *ide b* **and** *ide c* **and** «*g* : *a* → *exp b c*»
shows *Map* (*uncurry a b c g*) =
Currying.Uncurry (*Rts a*) (*Rts b*) (*Rts c*) (*Func b c* ∘ *Map g*) ∘
Unpack a b
unfolding *Map-def uncurry-def Func-def Rts-def Unpack-def*
using *assms RTSx.Map-uncurry ide-iff-RTS-obj arr-iff-RTS-sta ide-dom*
by *auto*

4.4.4 Cartesian Closure

sublocale *elementary-cartesian-closed-category*
comp p_0 p_1 *one trm exp eval curry*
proof
fix *b c*
assume *b*: *ide b* **and** *c*: *ide c*
show «*eval b c* : *exp b c* ⊗ *b* → *c*»
unfolding *eval-def exp-def*
using *b c RTSx.eval-in-hom*$_{RCR}$ *prod-char ide-iff-RTS-obj*
arr-iff-RTS-sta RTSx.obj-is-sta
by (*simp add*: *RTSx.obj-exp RTS*$_S$*.in-hom-char*$_{SbC}$ *comp-def*)
show *ide* (*exp b c*)
unfolding *exp-def*
using *b c ide-iff-RTS-obj RTSx.obj-exp* **by** *simp*
fix *a*
assume *a*: *ide a*
fix *g*
assume *g*: «*g* : *a* ⊗ *b* → *c*»
show *1*: «*curry a b c g* : *a* → *exp b c*»
unfolding *curry-def*
using *a b c g ide-char prod-char ide-iff-RTS-obj*
RTS$_S$*.arr-char*$_{SbC}$ *RTS*$_S$*.in-hom-char*$_{SbC}$ ‹*ide* (*exp b c*)›
exp-def comp-def RTSx.curry-in-hom RTSx.sta-curry
by (*metis* (*no-types*, *lifting*))
show *eval b c* · (*curry a b c g* ⊗ *b*) = *g*
using *a b c g 1 RTSx.uncurry-curry RTSx.uncurry-expansion*
arr-iff-RTS-sta ide-iff-RTS-obj RTS$_S$*.comp-char*
RTS$_S$*.in-hom-char*$_{SbC}$ *ideD*(*1*)
apply (*auto simp add*: *exp-def comp-def curry-def eval-def*)[*1*]
apply (*metis* (*no-types*, *lifting*) *comp-def prod-char*)
apply (*metis comp-def prod-simps*(*1*))
by (*metis* (*no-types*, *lifting*) *RTSx.H.ext RTSx.arr-coincidence*$_{CRC}$
RTSx.null-coincidence$_{CRC}$ *comp-def prod-char*)

```
    next
    fix a b c h
    assume a: ide a and b: ide b and c: ide c
    and h: «h : a → exp b c»
    show curry a b c (eval b c · (h ⊗ b)) = h
      using a b c h prod-char RTSx.curry-uncurry ide-iff-RTS-obj
            RTSx.uncurry-expansion RTS_S.comp-char RTS_S.in-hom-char_SbC
            RTSx.obj-is-sta RTSx.sta-prod RTS_S.arr-char_SbC
      apply (auto simp add: eval-def curry-def comp-def exp-def)[1]
      by (metis (no-types, lifting) RTSx.H.ext RTSx.arr-coincidence_CRC
          RTSx.null-coincidence_CRC)
  qed

  theorem is-elementary-cartesian-closed-category:
  shows elementary-cartesian-closed-category
              comp p_0 p_1 one trm exp eval curry
    ..

end
```

4.4.5 Associators

Here we expose the relationship between the associators internal to **RTS** and those external to it.

```
locale Association =
  rtscat arr-type
for arr-type :: 'A itself
and a :: 'A rtscat.arr
and b :: 'A rtscat.arr
and c :: 'A rtscat.arr +
assumes a: ide a
and b: ide b
and c: ide c
begin

  interpretation A: extensional-rts ‹Rts a›
    using a by simp
  interpretation B: extensional-rts ‹Rts b›
    using b by simp
  interpretation C: extensional-rts ‹Rts c›
    using c by simp

  interpretation A: identity-simulation ‹Rts a› ..
  interpretation B: identity-simulation ‹Rts b› ..
  interpretation C: identity-simulation ‹Rts c› ..

  interpretation AxB: product-rts ‹Rts a› ‹Rts b› ..
  interpretation AxB-xC: product-rts AxB.resid ‹Rts c› ..
  interpretation BxC: product-rts ‹Rts b› ‹Rts c› ..
```

interpretation *Ax-BxC*: *product-rts ‹Rts a› BxC.resid* **..**

interpretation *AxB*: *extensional-rts AxB.resid*
  using *A.extensional-rts-axioms B.extensional-rts-axioms*
      AxB.preserves-extensional-rts
  by *blast*
interpretation *BxC*: *extensional-rts BxC.resid*
  using *B.extensional-rts-axioms C.extensional-rts-axioms*
      BxC.preserves-extensional-rts
  by *blast*
interpretation *AxB-xC*: *extensional-rts AxB-xC.resid*
  using *AxB.extensional-rts-axioms C.extensional-rts-axioms*
      AxB-xC.preserves-extensional-rts
  by *blast*
interpretation *Ax-BxC*: *extensional-rts Ax-BxC.resid*
  using *A.extensional-rts-axioms BxC.extensional-rts-axioms*
      Ax-BxC.preserves-extensional-rts
  by *blast*

sublocale *ASSOC*: *ASSOC ‹Rts a› ‹Rts b› ‹Rts c›* **..**

interpretation *axb*: *extensional-rts ‹Rts* $(a \otimes b)$*›*
  using *a b* **by** *simp*
interpretation *bxc*: *extensional-rts ‹Rts* $(b \otimes c)$*›*
  using *b c* **by** *simp*
interpretation *axb-xc*: *extensional-rts ‹Rts* $((a \otimes b) \otimes c)$*›*
  using *a b c* **by** *simp*
interpretation *ax-bxc*: *extensional-rts ‹Rts* $(a \otimes (b \otimes c))$*›*
  using *a b c* **by** *simp*

interpretation *PU-ab*: *inverse-simulations*
                  ‹Rts $(a \otimes b)$*› AxB.resid ‹Pack a b› ‹Unpack a b›*
  using *a b inverse-simulations-Pack-Unpack* [*of a b*] **by** *fastforce*
interpretation *PU-bc*: *inverse-simulations*
                  ‹Rts $(b \otimes c)$*› BxC.resid ‹Pack b c› ‹Unpack b c›*
  using *b c inverse-simulations-Pack-Unpack* [*of b c*] **by** *fastforce*

interpretation *Axbc*: *product-rts ‹Rts a› ‹Rts* $(b \otimes c)$*›* **..**
interpretation *Axbc*: *identity-simulation Axbc.resid* **..**
interpretation *abxC*: *product-rts ‹Rts* $(a \otimes b)$*› ‹Rts c›* **..**
interpretation *abxC*: *identity-simulation abxC.resid* **..**

interpretation *AxPack-bc*:
  product-simulation ‹Rts a› BxC.resid ‹Rts a› ‹Rts $(b \otimes c)$*›*
    ‹I (Rts a)› ‹Pack b c› **..**
interpretation *AxUnpack-bc*:
  product-simulation ‹Rts a› ‹Rts $(b \otimes c)$*› ‹Rts a› BxC.resid*
    ‹I (Rts a)› ‹Unpack b c› **..**
interpretation *Unpack-abxC*:

```
  product-simulation ‹Rts (a ⊗ b)› ‹Rts c› AxB.resid ‹Rts c›
    ‹Unpack a b› ‹I (Rts c)› ..

sublocale PU-Axbc: inverse-simulations Axbc.resid Ax-BxC.resid
                   AxPack-bc.map AxUnpack-bc.map
proof
  show AxPack-bc.map ∘ AxUnpack-bc.map = Axbc.map
  proof −
    have AxPack-bc.map ∘ AxUnpack-bc.map =
         product-simulation.map (Rts a) (Rts (b ⊗ c))
           (A.map ∘ A.map) (Pack b c ∘ Unpack b c)
      using A.simulation-axioms PU-bc.F.simulation-axioms
            PU-bc.G.simulation-axioms PU-bc.inv'
            simulation-interchange
              [of Rts a Rts a A.map Rts (b ⊗ c)
                  BxC.resid Unpack b c Rts a
                  A.map Rts (b ⊗ c) Pack b c]
      by simp
    also have ... =
               product-simulation.map (Rts a) (Rts (b ⊗ c))
                 A.map (I (Rts (b ⊗ c)))
      using PU-bc.inv' A.simulation-axioms
            comp-identity-simulation [of Rts a Rts a A.map]
      by simp
    also have ... = Axbc.map
      using product-identity-simulation A.rts-axioms bxc.rts-axioms
      by blast
    finally show ?thesis by blast
  qed
  show AxUnpack-bc.map ∘ AxPack-bc.map = I Ax-BxC.resid
  proof −
    have AxUnpack-bc.map ∘ AxPack-bc.map =
         product-simulation.map (Rts a) BxC.resid
           (A.map ∘ A.map) (Unpack b c ∘ Pack b c)
      using A.simulation-axioms PU-bc.F.simulation-axioms
            PU-bc.G.simulation-axioms PU-bc.inv
            simulation-interchange
              [of Rts a Rts a A.map BxC.resid Rts (b ⊗ c)
                  Pack b c Rts a A.map BxC.resid Unpack b c]
      by simp
    also have ... =
               product-simulation.map (Rts a) BxC.resid
                 A.map (I BxC.resid)
      using PU-bc.inv A.simulation-axioms
            comp-identity-simulation [of Rts a Rts a A.map]
      by simp
    also have ... = I Ax-BxC.resid
      using product-identity-simulation A.rts-axioms BxC.rts-axioms
      by blast
```

```
    finally show ?thesis by blast
  qed
qed
```

The following shows that the simulation *Map* a[*a*, *b*, *c*] underlying an internal associator a[*a*, *b*, *c*] is transformed into a corresponding external associator *ASSOC.map* via invertible simulations that "unpack" product objects in **RTS** into corresponding product RTS's.

```
lemma Unpack-o-Map-assoc:
shows (AxUnpack-bc.map ∘ Unpack a (b ⊗ c)) ∘ Map a[a, b, c] =
      ASSOC.map ∘ (Unpack-abxC.map ∘ Unpack (a ⊗ b) c)
proof −
  have (AxUnpack-bc.map ∘ Unpack a (b ⊗ c)) ∘ Map a[a, b, c] =
       (AxUnpack-bc.map ∘ Unpack a (b ⊗ c)) ∘
         Pack a (b ⊗ c) ∘
          (Axbc.tuple
             (Map (𝔭₁[a, b] · 𝔭₁[a ⊗ b, c]))
             (Map ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩))
    using a b c
          Map-tuple
           [of 𝔭₁[a, b] · 𝔭₁[a ⊗ b, c] (a ⊗ b) ⊗ c a
              ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩ b ⊗ c]
          assoc-expansion [of a b c]
    by auto
  also have ... = AxUnpack-bc.map ∘
                  (Unpack a (b ⊗ c) ∘ Pack a (b ⊗ c)) ∘
                    (Axbc.tuple
                       (Map (𝔭₁[a, b] · 𝔭₁[a ⊗ b, c]))
                       (Map ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩))
    by force
  also have ... = AxUnpack-bc.map ∘
                  I Axbc.resid ∘
                    (Axbc.tuple
                       (Map (𝔭₁[a, b] · 𝔭₁[a ⊗ b, c]))
                       (Map ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩))
    using a b c Unpack-o-Pack [of a b ⊗ c] by fastforce
  also have ... = AxUnpack-bc.map ∘
                  (Axbc.tuple
                    (Map (𝔭₁[a, b] · 𝔭₁[a ⊗ b, c]))
                    (Map ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩))
    using comp-simulation-identity
          [of Axbc.resid Ax-BxC.resid AxUnpack-bc.map]
          AxUnpack-bc.simulation-axioms
    by simp
  also have ... = Ax-BxC.tuple
                  (I (Rts a) ∘ Map (𝔭₁[a, b] · 𝔭₁[a ⊗ b, c]))
                  (Unpack b c ∘ Map ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩)
  proof −
    have simulation (Rts ((a ⊗ b) ⊗ c)) (Rts a) (Map (𝔭₁[a, b] · 𝔭₁[a ⊗ b, c]))
```

```
    using a b c
   by (metis (no-types, lifting) arrD(3) cod-comp cod-pr1 dom-comp ide-prod
       pr-simps(4) pr-simps(5) seqI)
  moreover have simulation (Rts ((a ⊗ b) ⊗ c)) (Rts (b ⊗ c))
                 (Map ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩)
  proof −
    have «⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩ : (a ⊗ b) ⊗ c → b ⊗ c»
      using a b c by blast
    thus ?thesis
      using arrD(3) by fastforce
  qed
  ultimately show ?thesis
    using PU-bc.G.simulation-axioms A.simulation-axioms
          simulation-as-transformation.intro
          comp-product-simulation-tuple
            [of Rts a Rts a A.map Rts (b ⊗ c) BxC.resid Unpack b c
                Rts ((a ⊗ b) ⊗ c) Map (𝔭₁[a, b] · 𝔭₁[a ⊗ b, c])
                Map ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩]
    by blast
qed
also have ... = Ax-BxC.tuple
                  (Map (𝔭₁[a, b] · 𝔭₁[a ⊗ b, c]))
                  (Unpack b c ∘ Map ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩)
proof −
  have «p₁ a b · p₁ (a ⊗ b) c : (a ⊗ b) ⊗ c → a»
    using a b c by blast
  hence simulation (Rts ((a ⊗ b) ⊗ c)) (Rts a)
          (Map (𝔭₁[a, b] · 𝔭₁[a ⊗ b, c]))
    using arrD(3) by (metis (no-types, lifting) in-homE)
  thus ?thesis
    using comp-identity-simulation
          [of Rts ((a ⊗ b) ⊗ c) Rts a Map (𝔭₁[a, b] · 𝔭₁[a ⊗ b, c])]
    by fastforce
qed
also have ... = Ax-BxC.tuple
                  (Map 𝔭₁[a, b] ∘ Map 𝔭₁[a ⊗ b, c])
                  (Unpack b c ∘ Map ⟨𝔭₀[a, b] · 𝔭₁[a ⊗ b, c], 𝔭₀[a ⊗ b, c]⟩)
  using a b c Map-comp by auto
also have ... = Ax-BxC.tuple
                  (Map 𝔭₁[a, b] ∘ Map 𝔭₁[a ⊗ b, c])
                  (Unpack b c ∘
                     (Pack b c ∘
                        abxC.tuple
                          (Map (𝔭₀[a, b] · 𝔭₁[a ⊗ b, c]))
                          (Map 𝔭₀[a ⊗ b, c])))
  using a b c Map-tuple
  by (metis (no-types, lifting) comp-in-homI ide-prod pr-in-hom(1−2))
also have ... = Ax-BxC.tuple
                  (Map 𝔭₁[a, b] ∘ Map 𝔭₁[a ⊗ b, c])
```

```
                ((Unpack b c ∘ Pack b c) ∘
                    abxC.tuple
                      (Map (𝔭₀[a, b] · 𝔭₁[a ⊗ b, c]))
                      (Map 𝔭₀[a ⊗ b, c]))
  using fun.map-comp by metis
also have ... = Ax-BxC.tuple
                (Map 𝔭₁[a, b] ∘ Map 𝔭₁[a ⊗ b, c])
                (I BxC.resid ∘
                  BxC.tuple
                    (Map (𝔭₀[a, b] · 𝔭₁[a ⊗ b, c]))
                    (Map 𝔭₀[a ⊗ b, c]))
  using PU-bc.inv by simp
also have ... = Ax-BxC.tuple
                (Map 𝔭₁[a, b] ∘ Map 𝔭₁[a ⊗ b, c])
                (BxC.tuple
                  (Map 𝔭₀[a, b] ∘ Map 𝔭₁[a ⊗ b, c])
                  (Map 𝔭₀[a ⊗ b, c]))
proof −
  have «𝔭₀[a, b] · 𝔭₁[a ⊗ b, c] : (a ⊗ b) ⊗ c → b» ∧
       «𝔭₀[a ⊗ b, c] : (a ⊗ b) ⊗ c → c»
    using a b c by blast
  hence simulation (Rts ((a ⊗ b) ⊗ c)) BxC.resid
          (pointwise-tuple (Map (𝔭₀[a, b] · 𝔭₁[a ⊗ b, c])) (Map 𝔭₀[a ⊗ b, c]))
    using a b c arrD(3) simulation-tuple in-homE by metis
  thus ?thesis
    using a b c Map-comp
          comp-identity-simulation
            [of Rts ((a ⊗ b) ⊗ c) BxC.resid
                BxC.tuple
                  (Map (𝔭₀[a, b] · 𝔭₁[a ⊗ b, c]))
                  (Map 𝔭₀[a ⊗ b, c])]
    by auto
qed
also have ... = Ax-BxC.tuple
                ((AxB.P₁ ∘ Unpack a b) ∘
                  (abxC.P₁ ∘ Unpack (a ⊗ b) c))
                (BxC.tuple
                  ((AxB.P₀ ∘ Unpack a b) ∘
                    (abxC.P₁ ∘ Unpack (a ⊗ b) c))
                  (abxC.P₀ ∘ Unpack (a ⊗ b) c))
  using a b c Map-p₁ Map-p₀ by auto
also have ... = Ax-BxC.tuple
                (AxB.P₁ ∘ (Unpack a b ∘ abxC.P₁) ∘ Unpack (a ⊗ b) c)
                (BxC.tuple
                  (AxB.P₀ ∘ (Unpack a b ∘ abxC.P₁) ∘ Unpack (a ⊗ b) c)
                  (abxC.P₀ ∘ Unpack (a ⊗ b) c))
  using fun.map-comp by metis
also have ... = Ax-BxC.tuple
                (AxB.P₁ ∘ (AxB-xC.P₁ ∘ Unpack-abxC.map) ∘
```

```
                  Unpack (a ⊗ b) c)
                (BxC.tuple
                  (AxB.P₀ ∘ (AxB-xC.P₁ ∘ Unpack-abxC.map) ∘
                  Unpack (a ⊗ b) c)
                    (abxC.P₀ ∘ Unpack (a ⊗ b) c))
proof −

  have Unpack a b ∘ abxC.P₁ = AxB-xC.P₁ ∘ Unpack-abxC.map
  proof
    fix x
    show (Unpack a b ∘ abxC.P₁) x =
         (AxB-xC.P₁ ∘ Unpack-abxC.map) x
    proof (cases abxC.arr x)
      show ¬ abxC.arr x ⟹ ?thesis
        using Unpack-abxC.extensional AxB-xC.P₁.extensional
              abxC.P₁.extensional PU-ab.G.extensional
        by auto
      assume x: abxC.arr x
      show ?thesis
        using x a b c abxC.P₁-def Unpack-abxC.map-def AxB-xC.P₁-def
        apply auto[1]
        using AxB.arr-char by blast+
    qed
  qed
  thus ?thesis by simp
qed
also have ... = Ax-BxC.tuple
                (AxB.P₁ ∘ AxB-xC.P₁ ∘ Unpack-abxC.map ∘
                  Unpack (a ⊗ b) c)
                (BxC.tuple
                  (AxB.P₀ ∘ AxB-xC.P₁ ∘ Unpack-abxC.map ∘
                    Unpack (a ⊗ b) c)
                  (abxC.P₀ ∘ Unpack (a ⊗ b) c))
proof −
  have AxB.P₁ ∘ (AxB-xC.P₁ ∘ Unpack-abxC.map) ∘ Unpack (a ⊗ b) c =
       AxB.P₁ ∘ AxB-xC.P₁ ∘ Unpack-abxC.map ∘ Unpack (a ⊗ b) c
    by auto
  moreover have AxB.P₀ ∘ (AxB-xC.P₁ ∘ Unpack-abxC.map) ∘
                  Unpack (a ⊗ b) c =
                AxB.P₀ ∘ AxB-xC.P₁ ∘ Unpack-abxC.map ∘
                  Unpack (a ⊗ b) c
    by auto
  ultimately show ?thesis by simp
qed
also have ... = Ax-BxC.tuple
                (AxB.P₁ ∘ AxB-xC.P₁ ∘ Unpack-abxC.map)
                (BxC.tuple
                  (AxB.P₀ ∘ AxB-xC.P₁ ∘ Unpack-abxC.map)
                  abxC.P₀) ∘
```

```
                Unpack (a ⊗ b) c
    by (simp add: comp-pointwise-tuple)
  also have ... = Ax-BxC.tuple
                  (AxB.P₁ ∘ AxB-xC.P₁ ∘ Unpack-abxC.map)
                  (BxC.tuple
                    (AxB.P₀ ∘ AxB-xC.P₁ ∘ Unpack-abxC.map)
                    (AxB-xC.P₀ ∘ Unpack-abxC.map)) ∘
                  Unpack (a ⊗ b) c
  proof −

    have AxB-xC.P₀ ∘ Unpack-abxC.map = abxC.P₀
    proof
      fix x
      show (AxB-xC.P₀ ∘ Unpack-abxC.map) x = abxC.P₀ x
        using a b c abxC.P₀-def Unpack-abxC.map-def AxB-xC.P₀-def
              PU-ab.G.preserves-reflects-arr abxC.arr-char PU-ab.G.extensional
              abxC.P₁.extensional axb.not-arr-null AxB-xC.P₁.extensional
              AxB-xC.not-arr-null
        by auto
    qed
    thus ?thesis by simp
  qed
  also have ... = Ax-BxC.tuple
                  (AxB.P₁ ∘ AxB-xC.P₁)
                  (BxC.tuple (AxB.P₀ ∘ AxB-xC.P₁) AxB-xC.P₀) ∘
                  Unpack-abxC.map ∘ Unpack (a ⊗ b) c
    by (simp add: comp-pointwise-tuple)
  also have ... = ASSOC.map ∘ (Unpack-abxC.map ∘ Unpack (a ⊗ b) c)
    unfolding ASSOC.map-def by auto
  finally show ?thesis by blast
qed
```

As a corollary, we obtain an explicit formula for *Map* a[*a*, *b*, *c*] in terms of the external associator *ASSOC.map* and packing and unpacking isomorphisms.

```
corollary Map-assoc:
shows Map a[a, b, c] =
      (Pack a (b ⊗ c) ∘ AxPack-bc.map) ∘
        ASSOC.map ∘
          (Unpack-abxC.map ∘ Unpack (a ⊗ b) c)
proof −
  interpret PU-axbc: inverse-simulations
                  ‹Rts (a ⊗ (b ⊗ c))› Axbc.resid
                  ‹Pack a (b ⊗ c)› ‹Unpack a (b ⊗ c)›
    using a b c
        inverse-simulations-Pack-Unpack [of a b ⊗ c]
    by force
  interpret PU-abxc: inverse-simulations
                  ‹Rts ((a ⊗ b) ⊗ c)› abxC.resid
```

```
                ‹Pack (a ⊗ b) c› ‹Unpack (a ⊗ b) c›
  using a b c
      inverse-simulations-Pack-Unpack [of a ⊗ b c]
  by force
interpret PoAxP: composite-simulation
                Ax-BxC.resid Axbc.resid ‹Rts (a ⊗ b ⊗ c)›
                AxPack-bc.map ‹Pack a (b ⊗ c)› ..
interpret UxCoU: composite-simulation
                ‹Rts ((a ⊗ b) ⊗ c)› abxC.resid AxB-xC.resid
                ‹Unpack (a ⊗ b) c› Unpack-abxC.map ..
interpret AxUoU: composite-simulation
                ‹Rts (a ⊗ b ⊗ c)› Axbc.resid Ax-BxC.resid
                ‹Unpack a (b ⊗ c)› AxUnpack-bc.map ..

have ∗: inverse-simulations (Rts (a ⊗ b ⊗ c)) Ax-BxC.resid
          (Pack a (b ⊗ c) ∘ AxPack-bc.map)
          (AxUnpack-bc.map ∘ Unpack a (b ⊗ c))
proof
  show AxUoU.map ∘ PoAxP.map = I Ax-BxC.resid
  proof −
    have AxUoU.map ∘ PoAxP.map =
          AxUnpack-bc.map ∘ (Unpack a (b ⊗ c) ∘ Pack a (b ⊗ c)) ∘
            AxPack-bc.map
      using fun.map-comp by force
    also have ... = AxUnpack-bc.map ∘ I Axbc.resid ∘ AxPack-bc.map
      using PU-axbc.inv by simp
    also have ... = AxUnpack-bc.map ∘ AxPack-bc.map
      using comp-simulation-identity AxUnpack-bc.simulation-axioms
      by fastforce
    also have ... = I Ax-BxC.resid
      using PU-Axbc.inv by blast
    finally show ?thesis by blast
  qed
  show PoAxP.map ∘ AxUoU.map = I (Rts (a ⊗ b ⊗ c))
  proof −
    have PoAxP.map ∘ AxUoU.map =
          Pack a (b ⊗ c) ∘ (AxPack-bc.map ∘ AxUnpack-bc.map) ∘
            Unpack a (b ⊗ c)
      using fun.map-comp by auto
    also have ... = Pack a (b ⊗ c) ∘ I Axbc.resid ∘ Unpack a (b ⊗ c)
      using PU-Axbc.inv′ by simp
    also have ... = Pack a (b ⊗ c) ∘ Unpack a (b ⊗ c)
      using comp-simulation-identity PU-axbc.F.simulation-axioms
      by fastforce
    also have ... = I (Rts (a ⊗ b ⊗ c))
      using PU-axbc.inv′ by blast
    finally show ?thesis by blast
  qed
qed
```

```
    have simulation (Rts ((a ⊗ b) ⊗ c)) (Rts (a ⊗ b ⊗ c)) (Map a[a, b, c])
      using a b c arrD(3)
      by (metis (no-types, lifting) assoc-simps(1−3))
    hence Map a[a, b, c] = I (Rts (a ⊗ b ⊗ c)) ∘ Map a[a, b, c]
      using a b c
            comp-identity-simulation
              [of Rts ((a ⊗ b) ⊗ c) Rts (a ⊗ b ⊗ c) Map a[a, b, c]]
      by auto
    also have  ... = PoAxP.map ∘ AxUoU.map ∘ Map a[a, b, c]
      using a b c ∗ inverse-simulations.inv' by fastforce
    also have ... = PoAxP.map ∘ (AxUoU.map ∘ Map a[a, b, c])
      using fun.map-comp by fastforce
    also have ... = PoAxP.map ∘ (ASSOC.map ∘ UxCoU.map)
      using Unpack-o-Map-assoc by simp
    also have ... = PoAxP.map ∘ ASSOC.map ∘ UxCoU.map
      using fun.map-comp by fastforce
    finally show ?thesis by blast
  qed

end

context rtscat
begin

  proposition Map-assoc:
  assumes ide a and ide b and ide c
  shows Map a[a, b, c] =
        Pack a (b ⊗ c) ∘
          product-simulation.map (Rts a) (product-rts.resid (Rts b) (Rts c))
            (I (Rts a)) (Pack b c) ∘
            ASSOC.map (Rts a) (Rts b) (Rts c) ∘
              (product-simulation.map
                  (Rts (a ⊗ b)) (Rts c) (Unpack a b) (I (Rts c)) ∘
                Unpack (a ⊗ b) c)
  proof −
    interpret A: Association arr-type a b c
      using assms by unfold-locales auto
    show ?thesis
      using assms A.Map-assoc by force
  qed

end
```

4.4.6 Compositors

Here we expose the relationship between the compositors internal to **RTS** (inherited from *closed-monoidal-category*) and those external to it (given by horizontal composition of simulations).

```
context rtscat
```

begin

sublocale *CMC*: *cartesian-monoidal-category comp Prod* α ι
using *extends-to-cartesian-monoidal-category*$_{ECC}$ **by** *blast*

sublocale *ECMC*: *elementary-closed-monoidal-category comp Prod* α ι
exp eval curry
using *extends-to-elementary-closed-monoidal-category*$_{ECCC}$ **by** *blast*

end

locale *Composition* =
rtscat arr-type
for *arr-type* :: *'A itself*
and *a* :: *'A rtscat.arr*
and *b* :: *'A rtscat.arr*
and *c* :: *'A rtscat.arr* +
assumes *a*: *ide a*
and *b*: *ide b*
and *c*: *ide c*
begin

abbreviation *EXP*
where *EXP* $\equiv \lambda a\ b.$ *Rts* (*exp a b*)

interpretation *A*: *extensional-rts* ‹*Rts a*›
using *a* **by** *simp*
interpretation *B*: *extensional-rts* ‹*Rts b*›
using *b* **by** *simp*
interpretation *C*: *extensional-rts* ‹*Rts c*›
using *c* **by** *simp*
interpretation *AxB*: *product-rts* ‹*Rts a*› ‹*Rts b*› **..**
interpretation *BxC*: *product-rts* ‹*Rts b*› ‹*Rts c*› **..**
interpretation *AB*: *exponential-rts* ‹*Rts a*› ‹*Rts b*› **..**
interpretation *BC*: *exponential-rts* ‹*Rts b*› ‹*Rts c*› **..**
interpretation *AC*: *exponential-rts* ‹*Rts a*› ‹*Rts c*› **..**
interpretation *ABxA*: *product-rts AB.resid* ‹*Rts a*› **..**
interpretation *BCxAB*: *product-rts BC.resid AB.resid* **..**
interpretation *BCxAB*: *extensional-rts BCxAB.resid*
using *AB.is-extensional-rts BC.is-extensional-rts*
BCxAB.preserves-extensional-rts
by *fastforce*
interpretation *BCxAB-x-A*: *product-rts BCxAB.resid* ‹*Rts a*› **..**

interpretation *ab*: *extensional-rts* ‹*EXP a b*›
using *a b ide-exp-ax* **by** *simp*
interpretation *bc*: *extensional-rts* ‹*EXP b c*›
using *b c ide-exp-ax* **by** *simp*
interpretation *bcxab*: *product-of-extensional-rts* ‹*EXP b c*› ‹*EXP a b*› **..**

interpretation *abxA*: *product-rts ‹EXP a b› ‹Rts a›* **..**
interpretation *bcxB*: *product-rts ‹EXP b c› ‹Rts b›* **..**
interpretation *bc-x-abxA*: *product-rts ‹EXP b c› abxA.resid* **..**
interpretation *bcxab-x-A*: *product-rts bcxab.resid ‹Rts a›* **..**

interpretation *ASSOC*: *ASSOC BC.resid AB.resid ‹Rts a›* **..**
interpretation *COMP*: *COMP ‹Rts a› ‹Rts b› ‹Rts c›* **..**

interpretation *Assoc-abc*: *Association arr-type a b c*
using *a b c* **by** *unfold-locales*
interpretation *Assoc-bc-ab-a*: *Association arr-type ‹exp b c› ‹exp a b› a*
using *a b c ide-exp-ax* **by** *unfold-locales*

interpretation *Func-Unfunc-ab*: *inverse-simulations AB.resid ‹EXP a b›*
‹Func a b› ‹Unfunc a b›
using *a b inverse-simulations-Func-Unfunc* [*of a b*] **by** *blast*
interpretation *Func-Unfunc-bc*: *inverse-simulations BC.resid ‹EXP b c›*
‹Func b c› ‹Unfunc b c›
using *a b c inverse-simulations-Func-Unfunc* [*of b c*] **by** *blast*
interpretation *Func-bcxFunc-ab*: *product-simulation*
‹EXP b c› ‹EXP a b› BC.resid AB.resid
‹Func b c› ‹Func a b› **..**

The following shows that the simulation *Map* (*Comp a b c*) underlying an internal compositor *Comp a b c* is transformed into a corresponding external compositor *COMP.map* by invertible simulations that "unpack" product and exponential objects in RTS_S into corresponding RTS products and exponentials.

lemma *Func-o-Map-Comp*:
shows *Func a c ∘ Map* (*ECMC.Comp a b c*) =
COMP.map ∘ (*Func-bcxFunc-ab.map ∘ Unpack* (*exp b c*) (*exp a b*))
proof −
interpret *A*: *identity-simulation ‹Rts a›* **..**
interpret *B*: *identity-simulation ‹Rts b›* **..**
interpret *BC*: *identity-simulation BC.resid* **..**
interpret *ABxA*: *identity-simulation ABxA.resid* **..**
interpret *BCxB*: *product-rts BC.resid ‹Rts b›* **..**
interpret *abxa*: *extensional-rts ‹Rts* (*exp a b ⊗ a*)›
using *a b ide-exp-ax* **by** *simp*
interpret *eval-ab*: *simulation ‹Rts* (*exp a b ⊗ a*)› *‹Rts b› ‹Map* (*eval a b*)›
using *a b ide-exp-ax arrD eval-in-hom-ax* [*of a b*] **by** *force*
interpret *bc*: *identity-simulation ‹EXP b c›* **..**
interpret *bcxeval*: *product-simulation*
‹EXP b c› ‹Rts (*exp a b ⊗ a*)› *‹EXP b c› ‹Rts b›*
bc.map ‹Map (*eval a b*)› **..**
interpret *bcxab′*: *extensional-rts ‹Rts* (*exp b c ⊗ exp a b*)›
using *a b c ide-exp-ax* **by** *simp*
interpret *bcxab′-x-A*: *product-rts ‹Rts* (*exp b c ⊗ exp a b*)› *‹Rts a›* **..**
interpret *bcxab*: *identity-simulation bcxab.resid* **..**

interpret *bcxab-x-A*: *product-simulation bcxab.resid ‹Rts a› bcxab.resid ‹Rts a› bcxab.map A.map* **..**
interpret *bcxab*: *extensional-rts ‹Rts (exp b c ⊗ exp a b)›*
using *a b c ide-exp-ax* **by** *simp*
interpret *PU-abxa*: *inverse-simulations ‹Rts (exp a b ⊗ a)› abxA.resid ‹Pack (exp a b) a› ‹Unpack (exp a b) a›*
using *a b c inverse-simulations-Pack-Unpack* [*of exp a b a*] **by** *blast*
interpret *PU-bcxab-xa*: *inverse-simulations ‹Rts ((exp b c ⊗ exp a b) ⊗ a)› bcxab′-x-A.resid ‹Pack (exp b c ⊗ exp a b) a› ‹Unpack (exp b c ⊗ exp a b) a›*
using *a b c ide-exp-ax inverse-simulations-Pack-Unpack* **by** *simp*
interpret *PU-bcxab*: *inverse-simulations ‹Rts (exp b c ⊗ exp a b)› bcxab.resid ‹Pack (exp b c) (exp a b)› ‹Unpack (exp b c) (exp a b)›*
using *a b c inverse-simulations-Pack-Unpack* [*of exp b c exp a b*] **by** *blast*
interpret *FU-ac*: *inverse-simulations AC.resid ‹EXP a c› ‹Func a c› ‹Unfunc a c›*
using *a c inverse-simulations-Func-Unfunc* [*of a c*] **by** *blast*
interpret *Func-bcxB*: *product-simulation ‹EXP b c› ‹Rts b› BC.resid ‹Rts b› ‹Func b c› ‹I (Rts b)›*
..
interpret *UnpackxA*: *product-simulation ‹Rts (exp b c ⊗ exp a b)› ‹Rts a› bcxab.resid ‹Rts a› ‹Unpack (exp b c) (exp a b)› A.map* **..**
interpret *Func-abxA*: *product-simulation ‹EXP a b› ‹Rts a› AB.resid ‹Rts a› ‹Func a b› ‹I (Rts a)›*
..
interpret *Func-bcxFunc-ab-x-A*: *product-simulation bcxab.resid ‹Rts a› BCxAB.resid ‹Rts a› Func-bcxFunc-ab.map A.map* **..**
interpret *Func-bc-x-Func-abxA*: *product-simulation ‹EXP b c› abxA.resid BC.resid ABxA.resid ‹Func b c› Func-abxA.map* **..**
interpret *E-AB*: *RTSConstructions.evaluation-map ‹Rts a› ‹Rts b›* **..**
interpret *E-BC*: *RTSConstructions.evaluation-map ‹Rts b› ‹Rts c›* **..**
interpret *BCxE-AB*: *product-simulation BC.resid ABxA.resid BC.resid ‹Rts b› BC.map E-AB.map* **..**
interpret *E-ABoFunc-abxA*: *composite-simulation abxA.resid ABxA.resid ‹Rts b› Func-abxA.map E-AB.map* **..**
interpret *bc-x-E-ABoFunc-abxA*: *product-simulation ‹EXP b c› abxA.resid ‹EXP b c› ‹Rts b› bc.map ‹E-AB.map ∘ Func-abxA.map›* **..**

have *bc-map*: *bc.map ∘ bc.map = bc.map*
by *auto*

have *1*: «*eval b c* · (*exp b c* ⊗ *eval a b*) · a[*exp b c*, *exp a b*, *a*] :
(*exp b c* ⊗ *exp a b*) ⊗ *a* → *c*»
using *a b c eval-in-hom-ax* [*of b c*] *eval-in-hom-ax* [*of a b*] *ide-exp-ax*
by *fastforce*
have *Func a c* ∘ *Map* (*ECMC.Comp a b c*) =
Func a c ∘ *Map* (*curry* (*exp b c* ⊗ *exp a b*) *a c*
(*eval b c* · (*exp b c* ⊗ *eval a b*) ·
a[*exp b c*, *exp a b*, *a*]))
unfolding *ECMC.Comp-def*
using *Assoc-bc-ab-a.a Assoc-bc-ab-a.b a assoc-agreement* **by** *auto*
also have ... = *Func a c* ∘
Unfunc a c ∘
COMP.Currying.E.coext (*Rts* (*exp b c* ⊗ *exp a b*))
(*Map* (*eval b c* · (*exp b c* ⊗ *eval a b*) ·
a[*exp b c*, *exp a b*, *a*]) ∘
Pack (*exp b c* ⊗ *exp a b*) *a*)
proof −
have *ide* (*exp b c* ⊗ *exp a b*)
using *a b c ide-exp-ax* **by** *blast*
moreover have «*eval b c* · (*exp b c* ⊗ *eval a b*) ·
a[*exp b c*, *exp a b*, *a*] :
(*exp b c* ⊗ *exp a b*) ⊗ *a* → *c*»
using *a b c eval-in-hom-ax eval-in-hom-ax ide-exp-ax*
by *fastforce*
ultimately show *?thesis*
using *a Map-curry fun.map-comp* **by** *auto*
qed
also have ... = *I AC.resid* ∘
COMP.Currying.E.coext (*Rts* (*exp b c* ⊗ *exp a b*))
(*Map* (*eval b c* · (*exp b c* ⊗ *eval a b*) ·
a[*exp b c*, *exp a b*, *a*]) ∘
Pack (*exp b c* ⊗ *exp a b*) *a*)
using *FU-ac.inv′* **by** *simp*
also have ... = *I AC.resid* ∘
COMP.Currying.E.coext (*Rts* (*exp b c* ⊗ *exp a b*))
(*Map* (*eval b c*) ∘
Map (*exp b c* ⊗ *eval a b*) ∘
Map a[*exp b c*, *exp a b*, *a*] ∘
Pack (*exp b c* ⊗ *exp a b*) *a*)
proof −
have *Map* (*eval b c* · (*exp b c* ⊗ *eval a b*) · a[*exp b c*, *exp a b*, *a*]) =
Map (*eval b c*) ∘ *Map* (*exp b c* ⊗ *eval a b*) ∘ *Map* a[*exp b c*, *exp a b*, *a*]
using *1 Map-comp* **by** *fastforce*
thus *?thesis* **by** *auto*
qed
also have ... = *COMP.Currying.E.coext* (*Rts* (*exp b c* ⊗ *exp a b*))
(*E-BC.map* ∘ *Func-bcxB.map* ∘ *Unpack* (*exp b c*) *b* ∘
Map (*exp b c* ⊗ *eval a b*) ∘
Map a[*exp b c*, *exp a b*, *a*] ∘

```
                    Pack (exp b c ⊗ exp a b) a)
proof −
  have simulation bcxab'-x-A.resid (Rts c)
         (Map (eval b c) ∘
            Map (exp b c ⊗ eval a b) ∘
              Map a[exp b c, exp a b, a] ∘
                Pack (exp b c ⊗ exp a b) a)
  proof (intro simulation-comp)
    show simulation (Rts (exp b c ⊗ b)) (Rts c) (Map (eval b c))
      using a b c eval-in-hom-ax [of b c] arrD(3) by force
    show simulation (Rts (exp b c ⊗ exp a b ⊗ a)) (Rts (exp b c ⊗ b))
           (Map (exp b c ⊗ eval a b))
    proof −
      have «exp b c ⊗ eval a b : exp b c ⊗ exp a b ⊗ a → exp b c ⊗ b»
        using a b c Assoc-bc-ab-a.a eval-in-hom-ax eval-in-hom-ax
        by auto
      thus ?thesis
        using arrD(3) by fastforce
    qed
    show simulation bcxab'-x-A.resid (Rts ((exp b c ⊗ exp a b) ⊗ a))
           (Pack (exp b c ⊗ exp a b) a)
      using a b c PU-bcxab-xa.F.simulation-axioms by blast
    show simulation (Rts ((exp b c ⊗ exp a b) ⊗ a))
                    (Rts (exp b c ⊗ exp a b ⊗ a))
                    (Map a[exp b c, exp a b, a])
      using a b c ide-exp-ax arrD(3) [of a[exp b c, exp a b, a]] by auto
  qed
  moreover have Currying (Rts (exp b c ⊗ exp a b)) (Rts a) (Rts c) ..
  ultimately have simulation (Rts (exp b c ⊗ exp a b)) AC.resid
                    (COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                      (Map (eval b c) ∘
                        Map (exp b c ⊗ eval a b) ∘
                          Map a[exp b c, exp a b, a] ∘
                            Pack (exp b c ⊗ exp a b) a))
    using Currying.Curry-preserves-simulations by fastforce
  thus ?thesis
    using b c Assoc-abc.Map-assoc Map-eval comp-identity-simulation
    by auto
qed
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                  (E-BC.map ∘ Func-bcxB.map ∘ Unpack (exp b c) b ∘
                    (Pack (exp b c) b ∘
                      bcxeval.map ∘
                        Unpack (dom (exp b c)) (dom (eval a b))) ∘
                      Map a[exp b c, exp a b, a] ∘
                        Pack (exp b c ⊗ exp a b) a)
proof −
  have bcxeval.map = product-simulation.map
                       (Dom (exp b c)) (Dom (eval a b))
```

```
                (Map (exp b c)) (Map (eval a b))
  proof −
    have Dom (eval a b) = Rts (exp a b ⊗ a)
      using a b eval-in-hom-ax by fastforce
    thus ?thesis
      using a b c Map-ide ide-exp-ax by auto
  qed
  hence Map (exp b c ⊗ eval a b) =
        Pack (exp b c) b ∘
          bcxeval.map ∘
            Unpack (dom (exp b c)) (dom (eval a b))
    using a b c Map-prod ide-exp-ax eval-in-hom-ax by fastforce
  thus ?thesis by simp
qed
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                  (E-BC.map ∘ Func-bcxB.map ∘ Unpack (exp b c) b ∘
                    (Pack (exp b c) b ∘
                       bcxeval.map ∘
                         Unpack (exp b c) (exp a b ⊗ a)) ∘
                       Map a[exp b c, exp a b, a] ∘
                         Pack (exp b c ⊗ exp a b) a)
  using a b c ide-exp-ax eval-in-hom-ax by auto
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                  (E-BC.map ∘ (Func-bcxB.map ∘
                    (Unpack (exp b c) b ∘ Pack (exp b c) b)) ∘
                      bcxeval.map ∘
                        Unpack (exp b c) (exp a b ⊗ a) ∘
                          Map a[exp b c, exp a b, a] ∘
                            Pack (exp b c ⊗ exp a b) a)
  using fun.map-comp by metis
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                  (E-BC.map ∘ (Func-bcxB.map ∘ I bcxB.resid) ∘
                    bcxeval.map ∘
                      Unpack (exp b c) (exp a b ⊗ a) ∘
                        Map a[exp b c, exp a b, a] ∘
                          Pack (exp b c ⊗ exp a b) a)
  using b c Unpack-o-Pack ide-exp-ax by auto
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                  (E-BC.map ∘ Func-bcxB.map ∘
                    bcxeval.map ∘
                      Unpack (exp b c) (exp a b ⊗ a) ∘
                        Map a[exp b c, exp a b, a] ∘
                          Pack (exp b c ⊗ exp a b) a)
  using comp-simulation-identity [of bcxB.resid BCxB.resid Func-bcxB.map]
        Func-bcxB.simulation-axioms
  by auto
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                  (E-BC.map ∘ Func-bcxB.map ∘
                    product-simulation.map (EXP b c) (Rts (exp a b ⊗ a))
```

```
                    bc.map
                      (E-AB.map ∘ Func-abxA.map ∘ Unpack (exp a b) a) ∘
                    Unpack (exp b c) (exp a b ⊗ a) ∘
                      Map a[exp b c, exp a b, a] ∘
                        Pack (exp b c ⊗ exp a b) a)
  using a b c Map-eval eval-in-hom-ax by auto
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                  (E-BC.map ∘ Func-bcxB.map ∘
                    ((product-simulation.map
                        (EXP b c) ABxA.resid bc.map E-AB.map ∘
                      product-simulation.map
                        (EXP b c) abxA.resid bc.map Func-abxA.map) ∘
                      product-simulation.map
                        (EXP b c) (Rts (exp a b ⊗ a))
                        bc.map (Unpack (exp a b) a)) ∘
                      Unpack (exp b c) (exp a b ⊗ a) ∘
                        Map a[exp b c, exp a b, a] ∘
                          Pack (exp b c ⊗ exp a b) a)
proof −
  have product-simulation.map (EXP b c) (Rts (exp a b ⊗ a))
          bc.map (E-AB.map ∘ Func-abxA.map ∘ Unpack (exp a b) a) =
       product-simulation.map
         (EXP b c) abxA.resid bc.map (E-AB.map ∘ Func-abxA.map) ∘
         product-simulation.map
           (EXP b c) (Rts (exp a b ⊗ a)) bc.map (Unpack (exp a b) a)
    using bc-map Func-abxA.simulation-axioms E-AB.simulation-axioms
          bc.simulation-axioms E-ABoFunc-abxA.simulation-axioms
          PU-abxa.G.simulation-axioms
          simulation-interchange
            [of EXP b c EXP b c bc.map
                Rts (exp a b ⊗ a) abxA.resid Unpack (exp a b) a
                EXP b c bc.map Rts b E-AB.map ∘ Func-abxA.map]
    by simp
  also have ... = product-simulation.map
                    (EXP b c) ABxA.resid bc.map E-AB.map ∘
                    product-simulation.map
                      (EXP b c) abxA.resid bc.map Func-abxA.map ∘
                      product-simulation.map (EXP b c) (Rts (exp a b ⊗ a))
                        bc.map (Unpack (exp a b) a)
    using a b c bc-map Func-abxA.simulation-axioms E-AB.simulation-axioms
          bc.simulation-axioms
          simulation-interchange
            [of EXP b c EXP b c bc.map
                abxA.resid ABxA.resid Func-abxA.map
                EXP b c bc.map Rts b E-AB.map]
    by simp
  finally show ?thesis by simp
qed
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
```

```
                (E-BC.map ∘ Func-bcxB.map ∘
                  (bc-x-E-ABoFunc-abxA.map ∘
                    product-simulation.map (EXP b c) (Rts (exp a b ⊗ a))
                      bc.map (Unpack (exp a b) a)) ∘
                      Unpack (exp b c) (exp a b ⊗ a) ∘
                        Map a[exp b c, exp a b, a] ∘
                          Pack (exp b c ⊗ exp a b) a)
  using a b c E-AB.simulation-axioms Func-abxA.simulation-axioms
        bc.simulation-axioms bc-map
        simulation-interchange
          [of EXP b c EXP b c bc.map
              abxA.resid ABxA.resid Func-abxA.map
              EXP b c bc.map Rts b E-AB.map]
  by simp
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                ((E-BC.map ∘ Func-bcxB.map ∘
                  product-simulation.map
                    (EXP b c) abxA.resid
                    bc.map (E-AB.map ∘ Func-abxA.map)) ∘
                      (product-simulation.map
                        (EXP b c) (Rts (exp a b ⊗ a))
                        bc.map (Unpack (exp a b) a) ∘
                       Unpack (exp b c) (exp a b ⊗ a) ∘
                         Map a[exp b c, exp a b, a] ∘
                           Pack (exp b c ⊗ exp a b) a))
proof −
  have E-BC.map ∘ Func-bcxB.map ∘
         (product-simulation.map
           (EXP b c) abxA.resid bc.map (E-AB.map ∘ Func-abxA.map) ∘
            product-simulation.map
              (EXP b c) (Rts (exp a b ⊗ a))
              bc.map (Unpack (exp a b) a)) ∘
                Unpack (exp b c) (exp a b ⊗ a) ∘
                  Map a[exp b c, exp a b, a] ∘
                    Pack (exp b c ⊗ exp a b) a =
       (E-BC.map ∘ Func-bcxB.map ∘
          product-simulation.map
            (EXP b c) abxA.resid bc.map (E-AB.map ∘ Func-abxA.map)) ∘
            (product-simulation.map
              (EXP b c) (Rts (exp a b ⊗ a)) bc.map (Unpack (exp a b) a) ∘
                  Unpack (exp b c) (exp a b ⊗ a) ∘
                    Map a[exp b c, exp a b, a] ∘
                      Pack (exp b c ⊗ exp a b) a)
    using fun.map-comp by auto
  thus ?thesis by simp
qed
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                ((E-BC.map ∘ Func-bcxB.map ∘ bc-x-E-ABoFunc-abxA.map) ∘
                     (Assoc-bc-ab-a.ASSOC.map ∘ UnpackxA.map))
```

proof −
have *product-simulation.map*
(*EXP b c*) (*Rts* (*exp a b* ⊗ *a*)) *bc.map* (*Unpack* (*exp a b*) *a*) ∘
Unpack (*exp b c*) (*exp a b* ⊗ *a*) ∘
Map a[*exp b c*, *exp a b*, *a*] ∘
Pack (*exp b c* ⊗ *exp a b*) *a* =
product-simulation.map
(*EXP b c*) (*Rts* (*exp a b* ⊗ *a*)) *bc.map* (*Unpack* (*exp a b*) *a*) ∘
Unpack (*exp b c*) (*exp a b* ⊗ *a*) ∘
(*Pack* (*exp b c*) (*exp a b* ⊗ *a*) ∘
product-simulation.map
(*EXP b c*) *abxA.resid bc.map* (*Pack* (*exp a b*) *a*) ∘
Assoc-bc-ab-a.ASSOC.map ∘
(*UnpackxA.map* ∘
Unpack (*exp b c* ⊗ *exp a b*) *a*)) ∘
Pack (*exp b c* ⊗ *exp a b*) *a*
using *Assoc-bc-ab-a.Map-assoc* **by** *simp*
also have ... =
product-simulation.map
(*EXP b c*) (*Rts* (*exp a b* ⊗ *a*))
bc.map (*Unpack* (*exp a b*) *a*) ∘
(*Unpack* (*exp b c*) (*exp a b* ⊗ *a*) ∘ *Pack* (*exp b c*) (*exp a b* ⊗ *a*)) ∘
product-simulation.map
(*EXP b c*) *abxA.resid bc.map* (*Pack* (*exp a b*) *a*) ∘
Assoc-bc-ab-a.ASSOC.map ∘
(*UnpackxA.map* ∘
(*Unpack* (*exp b c* ⊗ *exp a b*) *a* ∘
Pack (*exp b c* ⊗ *exp a b*) *a*))
using *fun.map-comp* **by** *auto*
also have ... = *I bc-x-abxA.resid* ∘
Assoc-bc-ab-a.ASSOC.map ∘
(*UnpackxA.map* ∘ *I bcxab′-x-A.resid*)
proof −
have *product-simulation.map*
(*EXP b c*) (*Rts* (*exp a b* ⊗ *a*))
bc.map (*Unpack* (*exp a b*) *a*) ∘
(*Unpack* (*exp b c*) (*exp a b* ⊗ *a*) ∘
Pack (*exp b c*) (*exp a b* ⊗ *a*)) ∘
product-simulation.map
(*EXP b c*) *abxA.resid bc.map* (*Pack* (*exp a b*) *a*) =
product-simulation.map
(*EXP b c*) (*Rts* (*exp a b* ⊗ *a*)) *bc.map* (*Unpack* (*exp a b*) *a*) ∘
(*I bcxeval.A1xA0.resid*) ∘
product-simulation.map
(*EXP b c*) *abxA.resid bc.map* (*Pack* (*exp a b*) *a*)
using *a b c Unpack-o-Pack* [*of exp b c exp a b* ⊗ *a*] **by** *force*
also have ... = *product-simulation.map*
(*EXP b c*) (*Rts* (*exp a b* ⊗ *a*))
bc.map (*Unpack* (*exp a b*) *a*) ∘

```
                    product-simulation.map (EXP b c) abxA.resid
                      bc.map (Pack (exp a b) a)
    using comp-simulation-identity
          [of bcxeval.A1xA0.resid bc-x-abxA.resid
              product-simulation.map (EXP b c) (Rts (exp a b ⊗ a))
                bc.map (Unpack (exp a b) a)]
        Assoc-bc-ab-a.PU-Axbc.G.simulation-axioms
    by fastforce
  also have ... =
      product-simulation.map
        (EXP b c) abxA.resid (bc.map ∘ bc.map)
        (Unpack (exp a b) a ∘ Pack (exp a b) a)
    using simulation-interchange
        PU-abxa.F.simulation-axioms PU-abxa.G.simulation-axioms
        bc.simulation-axioms
    by fastforce
  also have ... = product-simulation.map (EXP b c) abxA.resid
                    bc.map (I abxA.resid)
    using a b bc-map Unpack-o-Pack ide-exp-ax by simp
  also have ... = I bc-x-abxA.resid
    using product-identity-simulation abxA.rts-axioms bc.rts-axioms
    by fastforce
  finally have product-simulation.map
                 (EXP b c) (Rts (exp a b ⊗ a))
                 bc.map (Unpack (exp a b) a) ∘
                 (Unpack (exp b c) (exp a b ⊗ a) ∘
                    Pack (exp b c) (exp a b ⊗ a)) ∘
                    product-simulation.map
                      (EXP b c) abxA.resid bc.map (Pack (exp a b) a) =
               I bc-x-abxA.resid
    by blast
  moreover have Unpack (exp b c ⊗ exp a b) a ∘
                  Pack (exp b c ⊗ exp a b) a =
                I bcxab'-x-A.resid
    using a b c Unpack-o-Pack [of exp b c ⊗ exp a b a] by blast
  ultimately show ?thesis by simp
 qed
 also have ... = Assoc-bc-ab-a.ASSOC.map ∘ UnpackxA.map
 using comp-identity-simulation [of - bc-x-abxA.resid Assoc-bc-ab-a.ASSOC.map]
       comp-simulation-identity [of bcxab'-x-A.resid - UnpackxA.map]
     Assoc-bc-ab-a.ASSOC.simulation-axioms UnpackxA.simulation-axioms
   by simp
 finally show ?thesis by simp
qed
also have ... =
       COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
             (E-BC.map ∘ BCxE-AB.map ∘
               (Func-bc-x-Func-abxA.map ∘ Assoc-bc-ab-a.ASSOC.map ∘
                 UnpackxA.map))
```

```
proof −
  have E-BC.map ∘ Func-bcxB.map ∘ bc-x-E-ABoFunc-abxA.map =
       E-BC.map ∘ Func-bcxB.map ∘
        (product-simulation.map (EXP b c) ABxA.resid bc.map E-AB.map ∘
       product-simulation.map (EXP b c) abxA.resid bc.map Func-abxA.map)
    using bc-map Func-abxA.simulation-axioms E-AB.simulation-axioms
          bc.simulation-axioms
          simulation-interchange
           [of EXP b c EXP b c bc.map abxA.resid ABxA.resid Func-abxA.map
               EXP b c bc.map Rts b E-AB.map]
    by simp
  also have ... = E-BC.map ∘
                  (Func-bcxB.map ∘
                     product-simulation.map (EXP b c) ABxA.resid
                       bc.map E-AB.map) ∘
                    product-simulation.map
                       (EXP b c) abxA.resid bc.map Func-abxA.map
    using fun.map-comp by auto
  also have ... = E-BC.map ∘
                  (product-simulation.map
                     BC.resid ABxA.resid BC.map E-AB.map ∘
                     product-simulation.map
                       (EXP b c) ABxA.resid (Func b c) ABxA.map) ∘
                       product-simulation.map
                         (EXP b c) abxA.resid bc.map Func-abxA.map
  proof −
    have Func-bcxB.map ∘
           product-simulation.map (EXP b c) ABxA.resid
           bc.map E-AB.map =
         product-simulation.map (EXP b c) ABxA.resid (Func b c) E-AB.map
      using simulation-interchange
              [of EXP b c EXP b c bc.map ABxA.resid Rts b E-AB.map
                  BC.residFunc b c Rts b B.map]
            E-AB.simulation-axioms Func-Unfunc-bc.G.simulation-axioms
            B.simulation-axioms bc.simulation-axioms
            comp-simulation-identity [of EXP b c BC.resid Func b c]
            comp-identity-simulation [of ABxA.resid Rts b E-AB.map]
            Func-Unfunc-bc.F.simulation-axioms
      by auto
    also have ... = product-simulation.map
                      BC.resid ABxA.resid BC.map E-AB.map ∘
                    product-simulation.map
                      (EXP b c) ABxA.resid (Func b c) ABxA.map
      using ABxA.simulation-axioms BC.simulation-axioms
            E-AB.simulation-axioms Func-Unfunc-bc.G.simulation-axioms
            comp-simulation-identity [of ABxA.resid Rts b E-AB.map]
            comp-identity-simulation [of EXP b c BC.resid Func b c]
            simulation-interchange
              [of EXP b c BC.resid Func b c
```

```
            ABxA.resid ABxA.resid ABxA.map
            BC.resid BC.map Rts b E-AB.map]
        Func-Unfunc-bc.F.simulation-axioms
    by auto
  finally have Func-bcxB.map ∘
                 product-simulation.map (EXP b c) ABxA.resid
                   bc.map E-AB.map =
               BCxE-AB.map ∘
                 product-simulation.map (EXP b c) ABxA.resid
                   (Func b c) ABxA.map
    by blast
  thus ?thesis by simp
qed
also have ... = E-BC.map ∘
                 (product-simulation.map BC.resid ABxA.resid
                    BC.map E-AB.map) ∘
                    (product-simulation.map
                       (EXP b c) ABxA.resid (Func b c) ABxA.map ∘
                     product-simulation.map
                       (EXP b c) abxA.resid bc.map Func-abxA.map)
  using fun.map-comp by auto
also have ... = E-BC.map ∘ BCxE-AB.map ∘ Func-bc-x-Func-abxA.map
  using simulation-interchange
        [of EXP b c EXP b c bc.map abxA.resid ABxA.resid Func-abxA.map
            BC.resid Func b c ABxA.resid ABxA.map]
    bc.simulation-axioms Func-abxA.simulation-axioms ABxA.simulation-axioms
        Func-Unfunc-bc.G.simulation-axioms
        comp-simulation-identity [of - - Func b c]
        comp-identity-simulation [of - ABxA.resid Func-abxA.map]
          Func-Unfunc-bc.F.simulation-axioms
  by auto
finally have E-BC.map ∘ Func-bcxB.map ∘ bc-x-E-ABoFunc-abxA.map =
             E-BC.map ∘ BCxE-AB.map ∘ Func-bc-x-Func-abxA.map
  by blast
thus ?thesis
  using fun.map-comp by metis
qed
also have ... = COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
                 (E-BC.map ∘ BCxE-AB.map ∘ ASSOC.map ∘
                   (Func-bcxFunc-ab-x-A.map ∘ UnpackxA.map))
proof −
  have 1: Func-bc-x-Func-abxA.map ∘ Assoc-bc-ab-a.ASSOC.map =
          ASSOC.map ∘ Func-bcxFunc-ab-x-A.map
  proof −
    have Func-bc-x-Func-abxA.map ∘ Assoc-bc-ab-a.ASSOC.map =
         Func-bc-x-Func-abxA.map ∘
          ⟨⟨bcxab.P_1 ∘ bcxab-x-A.P_1,
            AxB.tuple (bcxab.P_0 ∘ bcxab-x-A.P_1) bcxab-x-A.P_0⟩⟩
      unfolding Assoc-bc-ab-a.ASSOC.map-def by blast
```

also have ... = ⟨⟨*Func b c* ∘ (*bcxab.P*$_1$ ∘ *bcxab-x-A.P*$_1$),
Func-abxA.map ∘
(*AxB.tuple* (*bcxab.P*$_0$ ∘ *bcxab-x-A.P*$_1$) *bcxab-x-A.P*$_0$)⟩⟩
using *comp-product-simulation-tuple*
[*of EXP b c BC.resid Func b c abxA.resid ABxA.resid*
Func-abxA.map bcxab-x-A.resid bcxab.P$_1$ ∘ *bcxab-x-A.P*$_1$
AxB.tuple (*bcxab.P*$_0$ ∘ *bcxab-x-A.P*$_1$) *bcxab-x-A.P*$_0$]
Func-Unfunc-bc.F.simulation-axioms Func-abxA.simulation-axioms
bcxab-x-A.P$_1$*.simulation-axioms bcxab-x-A.P*$_0$*.simulation-axioms*
bcxab.P$_1$*.simulation-axioms bcxab.P*$_0$*.simulation-axioms*
simulation-comp simulation-tuple
by *auto*
also have ... = ⟨⟨*Func b c* ∘ (*bcxab.P*$_1$ ∘ *bcxab-x-A.P*$_1$),
⟨⟨*Func a b* ∘ (*bcxab.P*$_0$ ∘ *bcxab-x-A.P*$_1$),
bcxab-x-A.P$_0$⟩⟩ ⟩⟩
using *comp-product-simulation-tuple*
[*of EXP a b AB.resid Func a b Rts a Rts a A.map*
bcxab-x-A.resid bcxab.P$_0$ ∘ *bcxab-x-A.P*$_1$ *bcxab-x-A.P*$_0$]
comp-identity-simulation [*of bcxab-x-A.resid Rts a bcxab-x-A.P*$_0$]
A.simulation-axioms Func-Unfunc-ab.F.simulation-axioms
bcxab.P$_0$*.simulation-axioms bcxab-x-A.P*$_0$*.simulation-axioms*
bcxab-x-A.P$_1$*.simulation-axioms*
simulation-comp [*of - - bcxab-x-A.P*$_1$ *- bcxab.P*$_0$]
by *simp*
also have ... = ⟨⟨*BCxAB.P*$_1$ ∘ *BCxAB-x-A.P*$_1$ ∘ *Func-bcxFunc-ab-x-A.map*,
⟨⟨*BCxAB.P*$_0$ ∘ *BCxAB-x-A.P*$_1$ ∘ *Func-bcxFunc-ab-x-A.map*,
BCxAB-x-A.P$_0$ ∘ *Func-bcxFunc-ab-x-A.map*⟩⟩ ⟩⟩
proof −
have *Func b c* ∘ (*bcxab.P*$_1$ ∘ *bcxab-x-A.P*$_1$) =
BCxAB.P$_1$ ∘ *BCxAB-x-A.P*$_1$ ∘ *Func-bcxFunc-ab-x-A.map*
using *Func-Unfunc-bc.F.extensional BCxAB-x-A.P*$_1$*.extensional*
bcxab-x-A.map-def bcxab-x-A.P$_1$*-def bcxab.P*$_1$*-def*
Func-bcxFunc-ab-x-A.map-def Func-bcxFunc-ab.map-def
BCxAB-x-A.P$_1$*-def BCxAB.P*$_1$*-def*
by *auto*
moreover have *Func a b* ∘ (*bcxab.P*$_0$ ∘ *bcxab-x-A.P*$_1$) =
(*BCxAB.P*$_0$ ∘ *BCxAB-x-A.P*$_1$) ∘ *Func-bcxFunc-ab-x-A.map*
using *BCxAB.P*$_0$*-def BCxAB-x-A.P*$_1$*-def bcxab.P*$_0$*-def bcxab-x-A.P*$_1$*-def*
Func-bcxFunc-ab-x-A.map-def Func-bcxFunc-ab.map-def
Func-bcxFunc-ab-x-A.extensional Func-Unfunc-ab.F.extensional
by *auto*
moreover have *bcxab-x-A.P*$_0$ =
BCxAB-x-A.P$_0$ ∘ *Func-bcxFunc-ab-x-A.map*
using *BCxAB.P*$_0$*-def BCxAB-x-A.P*$_0$*-def bcxab-x-A.P*$_0$*-def*
Func-bcxFunc-ab-x-A.map-def Func-bcxFunc-ab.map-def
Func-bcxFunc-ab-x-A.extensional
by *auto*
ultimately show *?thesis* **by** *simp*
qed

```
    also have ... = ⟨⟨BCxAB.P₁ ∘ BCxAB-x-A.P₁,
                      ⟨⟨BCxAB.P₀ ∘ BCxAB-x-A.P₁, BCxAB-x-A.P₀⟩⟩ ⟩⟩ ∘
                      Func-bcxFunc-ab-x-A.map
      by (simp add: comp-pointwise-tuple)
    also have ... = ASSOC.map ∘ Func-bcxFunc-ab-x-A.map
      unfolding ASSOC.map-def by simp
    finally show ?thesis by blast
  qed
  have COMP.E-BC-o-BCxE-AB.map ∘
         (Func-bc-x-Func-abxA.map ∘ Assoc-bc-ab-a.ASSOC.map ∘
           UnpackxA.map) =
       COMP.E-BC-o-BCxE-AB.map ∘
         (ASSOC.map ∘ Func-bcxFunc-ab-x-A.map ∘ UnpackxA.map)
    using 1 by simp
  also have ... = COMP.E-BC-o-BCxE-AB.map ∘ ASSOC.map ∘
                    (Func-bcxFunc-ab-x-A.map ∘ UnpackxA.map)
    by auto
  finally show ?thesis by simp
qed
also have ... = COMP.Currying.E.coext BCxAB.resid
                  (E-BC.map ∘ BCxE-AB.map ∘ ASSOC.map) ∘
                  (Func-bcxFunc-ab.map ∘ Unpack (exp b c) (exp a b))
proof −
  have COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
         (COMP.E-BC-o-BCxE-AB.map ∘ ASSOC.map ∘
           (Func-bcxFunc-ab-x-A.map ∘ UnpackxA.map)) =
       COMP.Currying.E.coext (Rts (exp b c ⊗ exp a b))
         (COMP.E-BC-o-BCxE-AB.map ∘ ASSOC.map ∘
           (product-simulation.map (Rts (exp b c ⊗ exp a b)) (Rts a)
          (Func-bcxFunc-ab.map ∘ Unpack (exp b c) (exp a b)) A.map))
  proof −
    have A.map ∘ A.map = A.map
      using comp-identity-simulation [of Rts a Rts a A.map]
            A.simulation-axioms
      by simp
    hence Func-bcxFunc-ab-x-A.map ∘ UnpackxA.map =
          product-simulation.map (Rts (exp b c ⊗ exp a b)) (Rts a)
            (Func-bcxFunc-ab.map ∘ Unpack (exp b c) (exp a b)) A.map
      using simulation-interchange
            [of Rts (exp b c ⊗ exp a b) bcxab.resid
                Unpack (exp b c) (exp a b)
                Rts a Rts a A.map BCxAB.resid Func-bcxFunc-ab.map
                Rts a A.map]
           Func-bcxFunc-ab.simulation-axioms PU-bcxab.G.simulation-axioms
           A.simulation-axioms
      by simp
    thus ?thesis by simp
  qed
  also have ... = COMP.Currying.E.coext BCxAB.resid
```

```
                (E-BC.map ∘ BCxE-AB.map ∘ ASSOC.map) ∘
                (Func-bcxFunc-ab.map ∘ Unpack (exp b c) (exp a b))
  proof −
    have simulation (Rts (exp b c ⊗ exp a b)) BCxAB.resid
                (Func-bcxFunc-ab.map ∘ Unpack (exp b c) (exp a b))
      using simulation-comp Func-bcxFunc-ab.simulation-axioms
            PU-bcxab.G.simulation-axioms
      by auto
    moreover have simulation BCxAB-x-A.resid (Rts c)
                (COMP.E-BC-o-BCxE-AB.map ∘ ASSOC.map)
      using simulation-comp COMP.E-BC-o-BCxE-AB.simulation-axioms
            ASSOC.simulation-axioms
      by auto
    ultimately show ?thesis
      using COMP.Currying.E.comp-coext-simulation
              [of Rts (exp b c ⊗ exp a b) BCxAB.resid
                  Func-bcxFunc-ab.map ∘ Unpack (exp b c) (exp a b)
                  E-BC.map ∘ BCxE-AB.map ∘ ASSOC.map]
            BCxAB.weakly-extensional-rts-axioms
            bcxab'.weakly-extensional-rts-axioms
      by fastforce
  qed
  finally show ?thesis by blast
qed
also have ... = COMP.map ∘
                (Func-bcxFunc-ab.map ∘ Unpack (exp b c) (exp a b))
  unfolding COMP.map-def by simp
finally show ?thesis by simp
qed
```

We obtain as a corollary an explicit formula for *Map* (*Comp a b c*) in terms of the external compositor *COMP.map* and packing and unpacking isomorphisms.

```
corollary Map-Comp:
shows Map (ECMC.Comp a b c) =
      Unfunc a c ∘ COMP.map ∘
        (Func-bcxFunc-ab.map ∘ Unpack (exp b c) (exp a b))
proof −
  have Map (ECMC.Comp a b c) =
       I (EXP a c) ∘ Map (ECMC.Comp a b c)
    using a b c ECMC.Comp-in-hom arrD(3) [of ECMC.Comp a b c]
          comp-identity-simulation
            [of Rts (exp b c ⊗ exp a b) EXP a c Map (ECMC.Comp a b c)]
    by simp
  also have ... = Unfunc a c ∘ Func a c ∘ Map (ECMC.Comp a b c)
    using a c Unfunc-o-Func by simp
  also have ... = Unfunc a c ∘ (Func a c ∘ Map (ECMC.Comp a b c))
    by auto
  also have ... = Unfunc a c ∘
```

```
                COMP.map ∘
                 (Func-bcxFunc-ab.map ∘ Unpack (exp b c) (exp a b))
    using Func-o-Map-Comp by auto
   finally show ?thesis by blast
  qed

 end

 context rtscat
 begin

  abbreviation EXP
  where EXP ≡ λa b. Rts (exp a b)

  proposition Map-Comp:
  assumes ide a and ide b and ide c
  shows Map (ECMC.Comp a b c) =
        Unfunc a c ∘ COMP.map (Rts a) (Rts b) (Rts c) ∘
         (product-simulation.map (EXP b c) (EXP a b)
           (Func b c) (Func a b) ∘
           Unpack (exp b c) (exp a b))
  proof −
   interpret Comp: Composition
    using assms by unfold-locales
   show ?thesis
    using Comp.Map-Comp by blast
  qed

 end

end
```

4.5 Top-Level Interpretation

```
theory RTSCat-Interp
imports RTSCatx RTSCat
begin
```

The purpose of this section is simply to demonstrate the possibility of making top-level interpretations of locales *rtscatx* and *rtscat*. It is important to do this because some kinds of clashes that occur when the same names are used in multiple sublocales only cause a problem when an attempt is made to instantiate the locale in the top-level name space.

```
 interpretation RTSx: rtscatx ‹TYPE(V)›
 proof −
  interpret V: universe ‹TYPE(V)›
   using V-is-universe by auto
  show rtscatx (TYPE(V)) ..
```

```
  qed

  interpretation RTS: rtscat ‹TYPE(V)›
  proof −
    interpret V: universe ‹TYPE(V)›
      using V-is-universe by auto
    show rtscat (TYPE(V)) ..
  qed

end
```

Chapter 5

RTS-Enriched Categories

5.1 RTS-Enriched Categories

The category **RTS** is cartesian closed, hence monoidal closed. This implies that each hom-set of **RTS** itself carries the structure of an RTS, so that **RTS** becomes a category "enriched in itself". In this section we show that RTS-categories are essentially the same thing as categories enriched in **RTS**, and that the RTS-category $\mathbf{RTS}^\dagger$ is equivalent to the RTS-category determined by **RTS** regarded as a category enriched in itself. Thus, the complete structure of the RTS-category $\mathbf{RTS}^\dagger$ is already determined by its ordinary subcategory **RTS**.

theory *RTSEnrichedCategory*
imports *RTSCatx RTSCat EnrichedCategoryBasics.CartesianClosedMonoidalCategory*
EnrichedCategoryBasics.EnrichedCategory
begin

context *rtscat*
begin

sublocale *CMC*: *cartesian-monoidal-category comp Prod* α ι
using *extends-to-cartesian-monoidal-category*$_{ECC}$ **by** *blast*

The tensor for *elementary-cartesian-closed-monoidal-category* is given by the binary functor *Prod*. This functor is defined in uncurried form, which is consistent with its nature as a functor defined on a product category. However, the tensor *CMC.tensor* defined in *monoidal-category* is a curried version. There might be a way to streamline this, if the various monoidal category locales were changed so that the binary functor used to specify the tensor were given in curried form, but I have not yet attempted to do this. For now, we have two versions of tensor, which we need to show are equal to each other.

lemma *tensor-agreement*:
assumes *arr f* **and** *arr g*
shows *CMC.tensor f g* = *f* ⊗ *g*
by *simp*

The situation with tupling and "duplicators" is similar.

lemma *tuple-agreement*:
assumes *span f g*
shows *CMC.tuple f g* = ⟨*f*, *g*⟩
proof (*intro pr-joint-monic* [*of cod f cod g CMC.tuple f g* ⟨*f*, *g*⟩])
show *ide* (*cod f*) **and** *ide* (*cod g*)
using *assms* **by** *auto*
show *seq* $\mathfrak{p}_0$[*cod f*, *cod g*] (*CMC.tuple f g*)
by (*metis* (*no-types*, *lifting*) *CMC.ECC.seq-pr-tuple* ‹*ide* (*cod f*)›
‹*ide* (*cod g*)› *assms pr-agreement*(*1*))
show $\mathfrak{p}_0$[*cod f*, *cod g*] · *CMC.tuple f g* = $\mathfrak{p}_0$[*cod f*, *cod g*] · ⟨*f*, *g*⟩
using *assms pr-agreement*(*1*−*2*)
by (*metis* (*no-types*, *lifting*) *CMC.ECC.pr-tuple*(*2*) ‹*ide* (*cod f*)›
‹*ide*(*cod g*)› *pr-tuple*(*2*))
show $\mathfrak{p}_1$[*cod f*, *cod g*] · *CMC.tuple f g* = $\mathfrak{p}_1$[*cod f*, *cod g*] · ⟨*f*, *g*⟩
using *assms pr-agreement*(*1*−*2*)
by (*metis* (*no-types*, *lifting*) *CMC.ECC.pr-tuple*(*1*) ‹*ide* (*cod f*)›
‹*ide* (*cod g*)› *pr-tuple*(*1*))
qed

lemma *dup-agreement*:
assumes *arr f*
shows *CMC.dup f* = *dup f*
using *assms tuple-agreement* **by** *simp*

sublocale *elementary-cartesian-closed-monoidal-category*
comp Prod α ι *exp eval curry*
using *extends-to-elementary-cartesian-closed-monoidal-category*$_{ECCC}$
by *blast*

We have a need for the following expansion of associators in terms of tensor and projections. This is actually the definition of the associators given in *category-with-binary-products*, but it could (and perhaps should) be proved as a consequence of the locale assumptions in *elementary-cartesian-category*. Here we already have the fact *assoc-agreement* which expresses that the definition of associators given in *category-with-binary-products* agrees with the version derived from the locale parameters in *cartesian-monoidal-category*, and *prod-eq-tensor*, which expresses that the tensor equals the cartesian product. So we can just use these facts, together with the definition from *elementary-cartesian-category*, to avoid a longer proof.

lemma *assoc-expansion*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *CMC.assoc a b c* =

$\langle \mathfrak{p}_1[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c],\ \langle \mathfrak{p}_0[a,\ b] \cdot \mathfrak{p}_1[a \otimes b,\ c],\ \mathfrak{p}_0[a \otimes b,\ c] \rangle \rangle$
using *assms assoc-def assoc-agreement* **by** *simp*

lemma *extends-to-enriched-category*:
shows *enriched-category comp Prod* α ι
(*Collect ide*) *exp ECMC.Id ECMC.Comp*
using *ECMC.is-enriched-in-itself* **by** *blast*

end

locale *rts-enriched-category* =
universe arr-type +
RTS: *rtscat arr-type* +
enriched-category RTS.comp RTS.Prod RTS.α *RTS.*ι *Obj Hom Id Comp*
for *arr-type* :: $'A$ *itself*
and *Obj* :: $'O$ *set*
and *Hom* :: $'O \Rightarrow 'O \Rightarrow 'A$ *rtscatx.arr*
and *Id* :: $'O \Rightarrow 'A$ *rtscatx.arr*
and *Comp* :: $'O \Rightarrow 'O \Rightarrow 'O \Rightarrow 'A$ *rtscatx.arr*
begin

abbreviation HOM_{EC}
where $HOM_{EC}\ a\ b \equiv RTS.Rts\ (Hom\ a\ b)$

end

locale *hom-rts* =
rts-enriched-category +
fixes *a* :: $'b$
and *b* :: $'b$
assumes *a*: $a \in Obj$
and *b*: $b \in Obj$
begin

sublocale *extensional-rts* ‹$HOM_{EC}\ a\ b$›
using *a b* **by** *force*

sublocale *small-rts* ‹$HOM_{EC}\ a\ b$›
using *a b* **by** *force*

end

locale *rts-enriched-functor* =
RTS: *rtscat arr-type* +
A: *rts-enriched-category arr-type* Obj_A Hom_A Id_A $Comp_A$ +
B: *rts-enriched-category arr-type* Obj_B Hom_B Id_B $Comp_B$ +
enriched-functor RTS.comp RTS.Prod RTS.α *RTS.*ι

for *arr-type* :: *'A itself*
begin

lemma *is-local-simulation*:
assumes $a \in Obj_A$ **and** $b \in Obj_A$
shows *simulation* $(A.HOM_{EC}\ a\ b)\ (B.HOM_{EC}\ (F_o\ a)\ (F_o\ b))$
$(RTS.Map\ (F_a\ a\ b))$
using *assms preserves-Hom* [*of a b*] *RTS.arrD* [*of* F_a *a b*] **by** *auto*

end

locale *fully-faithful-rts-enriched-functor* =
rts-enriched-functor +
fully-faithful-enriched-functor RTS.comp RTS.Prod RTS.α RTS.ι

5.2 RTS-Enriched Categories induce RTS-Categories

Here we show that every RTS-enriched category determines a corresponding RTS-category. This is done by combining the RTS's underlying the homs of the RTS-enriched category, forming a global RTS that provides the vertical structure of the RTS-category. The composition operation of the RTS-enriched category is used to obtain the horizontal structure.

locale *rts-category-of-enriched-category* =
universe arr-type +
RTS: *rtscat arr-type* +
rts-enriched-category arr-type Obj Hom Id Comp
for *arr-type* :: *'A itself*
and *Obj* :: *'O set*
and *Hom* :: *'O* ⇒ *'O* ⇒ *'A rtscatx.arr*
and *Id* :: *'O* ⇒ *'A rtscatx.arr*
and *Comp* :: *'O* ⇒ *'O* ⇒ *'O* ⇒ *'A rtscatx.arr*
begin

notation *RTS.in-hom* («- : - → -»)
notation *RTS.prod* (**infixr** ⊗ *51*)
notation *RTS.one* (**1**)
notation *RTS.assoc* (a[-, -, -])
notation *RTS.lunit* (l[-])
notation *RTS.runit* (r[-])

Here we define the "global RTS", obtained as the disjoint union of all the hom-RTS's. Note that types 'O and 'A are fixed in the current context: type 'O is the type of "objects" of the given RTS-enriched category, and type 'A is the type of the universe that underlies the base category *RTS*.

datatype (*'o*, *'a*) *arr* =
Null
| *MkArr 'o 'o 'a*

```
fun Dom :: ('O, 'A) arr ⇒ 'O
where Dom (MkArr a - -) = a
  | Dom - = undefined

fun Cod :: ('O, 'A) arr ⇒ 'O
where Cod (MkArr - b -) = b
  | Cod - = undefined

fun Trn :: ('O, 'A) arr ⇒ 'A
where Trn (MkArr - - t) = t
  | Trn - = undefined

abbreviation Arr :: ('O, 'A) arr ⇒ bool
where Arr ≡ λt. t ≠ Null ∧ Dom t ∈ Obj ∧ Cod t ∈ Obj ∧
              residuation.arr (HOM_EC (Dom t) (Cod t)) (Trn t)

abbreviation Ide :: ('O, 'A) arr ⇒ bool
where Ide ≡ λt. t ≠ Null ∧ Dom t ∈ Obj ∧ Cod t ∈ Obj ∧
              residuation.ide (HOM_EC (Dom t) (Cod t)) (Trn t)

definition Con :: ('O, 'A) arr ⇒ ('O, 'A) arr ⇒ bool
where Con t u ≡ Arr t ∧ Arr u ∧ Dom t = Dom u ∧ Cod t = Cod u ∧
              residuation.con (HOM_EC (Dom t) (Cod t)) (Trn t) (Trn u)
```

The global residuation is obtained by combining the local residuations of each of the hom-RTS's.

```
fun resid  (infix \ 70)
where resid Null u = Null
  | resid t Null = Null
  | resid t u = (if Con t u
                 then MkArr (Dom t) (Cod t)
                       (HOM_EC (Dom t) (Cod t) (Trn t) (Trn u))
                 else Null)

sublocale V: ResiduatedTransitionSystem.partial-magma resid
  apply unfold-locales
  by (metis Trn.cases resid.simps(1−2))

lemma null-char:
shows V.null = Null
  by (metis V.null-is-zero(2) resid.simps(1))

lemma ConI [intro]:
assumes Arr t and Arr u and Dom t = Dom u and Cod t = Cod u
and residuation.con (HOM_EC (Dom t) (Cod t)) (Trn t) (Trn u)
shows Con t u
  using assms Con-def by simp
```

```
lemma ConE [elim]:
assumes Con t u
and ⟦Arr t; Arr u; Dom t = Dom u; Cod t = Cod u;
     residuation.con (HOM_EC (Dom t) (Cod t)) (Trn t) (Trn u)⟧ ⟹ T
shows T
  using assms Con-def by blast

lemma Con-sym:
assumes Con t u
shows Con u t
  using assms Con-def extensional-rts-def residuation.con-sym
        rts.axioms(1)
  by fastforce

lemma resid-ne-Null-imp-Con:
assumes t \ u ≠ Null
shows Con t u
  using assms resid.elims by metis

sublocale V: residuation resid
proof
  fix t u :: ('O, 'A) arr
  assume tu: t \ u ≠ V.null
  interpret hom: hom-rts arr-type Obj Hom Id Comp ‹Dom t› ‹Cod t›
    using tu null-char resid-ne-Null-imp-Con
    by unfold-locales auto
  show t \ u ≠ V.null ⟹ u \ t ≠ V.null
    using tu null-char Con-sym
    apply (cases t; cases u)
      apply simp-all
    by metis
  show (t \ u) \ (t \ u) ≠ V.null
    using tu null-char hom.arr-resid hom.con-arr-self Con-def
    apply (cases t; cases u)
      apply simp-all
    by metis
  next
  fix t u v :: ('O, 'A) arr
  assume tuv: (v \ t) \ (u \ t) ≠ V.null
  have tu: Con t u
    using tuv null-char Con-sym resid-ne-Null-imp-Con
    by (metis Con-def)
  have tv: Con t v
    using tuv null-char Con-sym resid-ne-Null-imp-Con
    by (metis Con-def)
  interpret hom: hom-rts arr-type Obj Hom Id Comp ‹Dom t› ‹Cod t›
    using tu null-char arr.exhaust resid.simps(1−3)
      apply unfold-locales
    by blast+
```

```
  show (v \ t) \ (u \ t) = (v \ u) \ (t \ u)
  proof −
    have ⋀td tc tt ud uc ut vd vc vt.
           ⟦t = MkArr td tc tt; u = MkArr ud uc ut; v = MkArr vd vc vt⟧
             ⟹ ?thesis
    proof −
      fix td tc tt ud uc ut vd vc vt
      assume t: t = MkArr td tc tt
      assume u: u = MkArr ud uc ut
      assume v: v = MkArr vd vc vt
      show (v \ t) \ (u \ t) = (v \ u) \ (t \ u)
        using t u v tu tv tuv Con-def hom.cube null-char
        apply auto[1]
        by (metis Cod.simps(1) Dom.simps(1) hom.con-sym hom.arr-resid
          hom.arr-resid-iff-con hom.resid-reflects-con)+
    qed
    thus ?thesis
      using tu tv tuv hom.cube hom.arr-resid-iff-con hom.resid-reflects-con
          null-char resid-ne-Null-imp-Con
      apply (cases v, blast)
      apply (cases u, blast)
      apply (cases t, blast)
      by metis
  qed
qed

notation V.con  (infix ⌢ 50)

lemma con-char:
shows t ⌢ u ⟷ Con t u
proof
  show t ⌢ u ⟹ Con t u
    using null-char resid-ne-Null-imp-Con by auto
  show Con t u ⟹ t ⌢ u
    using null-char
    by (cases t; cases u) auto
qed

lemma arr-char:
shows V.arr t ⟷ Arr t
by (metis Con-def RTS.ideD_RTSC V.arr-def con-char extensional-rts.axioms(1)
    ide-Hom residuation.arrE rts.axioms(1))

lemma ide-char:
shows V.ide t ⟷ Ide t
proof (cases V.arr t)
  show ¬ V.arr t ⟹ ?thesis
    by (metis RTS.ideD_RTSC V.arrI V.ide-def arr-char ide-Hom
        residuation.ide-implies-arr rts.axioms(1) small-rts-def)
```

```
  assume t: V.arr t
  interpret hom: hom-rts arr-type Obj Hom Id Comp ‹Dom t› ‹Cod t›
    using t arr-char
    by unfold-locales auto
  show ?thesis
    using t null-char con-char V.ide-def arr-char hom.ide-def
    by (cases t) auto
qed

lemma con-implies-Par:
assumes t ⌢ u
shows Dom t = Dom u and Cod t = Cod u
  using assms con-char by blast+

lemma Dom-resid [simp]:
assumes t ⌢ u
shows Dom (t \ u) = Dom t
  using assms con-char
  by (cases t; cases u) auto

lemma Cod-resid [simp]:
assumes t ⌢ u
shows Cod (t \ u) = Cod t
  using assms con-char
  by (cases t; cases u) auto

lemma Trn-resid [simp]:
assumes t ⌢ u
shows Trn (t \ u) = HOM_EC (Dom t) (Cod t) (Trn t) (Trn u)
  using assms con-char
  by (cases t; cases u) auto
```

Targets of arrows of the global RTS agree with the local versions from which they were derived. The same will be shown for sources below.

```
lemma trg-char:
shows V.trg t = (if V.arr t
                 then MkArr (Dom t) (Cod t)
                        (residuation.trg (HOM_EC (Dom t) (Cod t)) (Trn t))
                 else Null)
proof (cases V.arr t)
  show ¬ V.arr t ⟹ ?thesis
    using V.con-def V.trg-def null-char by auto
  show V.arr t ⟹ ?thesis
    using null-char V.not-arr-null
    apply (cases t)
     apply auto[1]
    by (metis (no-types, lifting) RTS.ideD_RTSC V.arrE V.trg-def
        arr-char con-char ide-Hom resid.simps(3)
        residuation.resid-arr-self rts.axioms(1) small-rts-def)
```

qed

sublocale *rts resid*
proof
show $\bigwedge t.\ V.arr\ t \Longrightarrow V.ide\ (V.trg\ t)$
by (*simp add*: *arr-char extensional-rts.axioms*(*1*) *ide-char*
rts.ide-trg trg-char)
show *1*: $\bigwedge a\ t.\ [\![V.ide\ a;\ t \frown a]\!] \Longrightarrow t \setminus a = t$
proof −
fix *a t*
assume *a*: *V.ide a*
assume *con*: $t \frown a$
have $t \setminus a \neq Null \wedge t \neq Null$
using *con null-char* **by** *auto*
moreover have $Dom\ (t \setminus a) = Dom\ t \wedge Cod\ (t \setminus a) = Cod\ t$
using *a con ide-char con-char Con-def*
by (*metis V.arrE V.arr-resid-iff-con V.con-sym V.cube V.ideE*)
moreover have $Trn\ (t \setminus a) = Trn\ t$
using *a con ide-char con-char small-rts-def Con-def*
by (*metis* (*no-types*, *lifting*) *RTS.ideD*$_{RTSC}$(*1*) *Trn-resid ide-Hom*
rts.resid-arr-ide)
ultimately show $t \setminus a = t$
by (*metis Cod.elims Dom.simps*(*1*) *Trn.simps*(*1*))
qed
show $\bigwedge a\ t.\ [\![V.ide\ a;\ a \frown t]\!] \Longrightarrow V.ide\ (a \setminus t)$
using *ide-char con-char*
by (*metis 1 V.arr-resid V.con-arr-self V.con-sym V.cube V.ideE V.ideI*)
show $\bigwedge t\ u.\ t \frown u \Longrightarrow \exists a.\ V.ide\ a \wedge a \frown t \wedge a \frown u$
proof −
fix *t u*
assume *tu*: $t \frown u$
interpret *hom*: *hom-rts arr-type Obj Hom Id Comp* ‹*Dom t*› ‹*Cod t*›
using *tu con-char arr-char*
by *unfold-locales blast*+
have *1*: *hom.con* (*Trn t*) (*Trn u*)
using *tu con-char* **by** *auto*
obtain α **where** α: $hom.ide\ \alpha \wedge hom.con\ \alpha\ (Trn\ t) \wedge hom.con\ \alpha\ (Trn\ u)$
using *1 hom.con-imp-coinitial-ax* **by** *auto*
have *V.ide* (*MkArr* (*Dom t*) (*Cod t*) α)
using *tu* α *V.con-implies-arr arr-char ide-char* **by** *auto*
moreover have $MkArr\ (Dom\ t)\ (Cod\ t)\ \alpha \frown t \wedge$
$MkArr\ (Dom\ t)\ (Cod\ t)\ \alpha \frown u$
using α *tu con-char hom.ide-implies-arr Con-def* **by** *auto*
ultimately show $\exists a.\ V.ide\ a \wedge V.con\ a\ t \wedge V.con\ a\ u$ **by** *blast*
qed
show $\bigwedge t\ u\ v.\ [\![V.ide\ (t \setminus u);\ u \frown v]\!] \Longrightarrow t \setminus u \frown v \setminus u$
proof −
fix *t u v*
assume *tu*: *V.ide* $(t \setminus u)$

```
    assume uv: u ⌢ v
    have 1: t \ u ≠ V.null
      using tu by auto
    interpret hom: hom-rts arr-type Obj Hom Id Comp ‹Dom t› ‹Cod t›
      using 1 con-char arr-char
      by unfold-locales blast+
    have hom.con (HOM_EC (Dom t) (Cod t) (Trn t) (Trn u))
                 (HOM_EC (Dom t) (Cod t) (Trn v) (Trn u))
    proof −
      have hom.con (Trn t) (Trn u)
        using tu 1 con-char by auto
      have hom.ide (HOM_EC (Dom t) (Cod t) (Trn t) (Trn u))
        using tu 1 ide-char Dom.simps(1) Cod.simps(1) Trn.simps(1) V.conI
        by auto
      moreover have hom.con (Trn u) (Trn v)
        using uv con-char 1 V.conI Con-def by fastforce
      ultimately show ?thesis
        using hom.con-target by blast
    qed
    thus t \ u ⌢ v \ u
      unfolding con-char Con-def
      using tu uv 1 null-char V.arr-resid arr-char V.ide-implies-arr
      apply clarsimp
      apply (intro conjI)
      subgoal by (meson V.con-sym)
      subgoal using V.arr-resid V.con-sym by meson
      by (metis (mono-tags, lifting) Cod-resid Dom-resid Trn-resid
          V.conI V.con-sym con-implies-Par(1−2))+
  qed
qed

lemma is-rts:
shows rts resid
  ..

sublocale V: extensional-rts resid
proof
  fix t u
  assume tu: cong t u
  have 1: t \ u ≠ V.null
    using tu by auto
  interpret hom: hom-rts arr-type Obj Hom Id Comp ‹Dom t› ‹Cod t›
    using 1 con-char arr-char [of t]
    by unfold-locales blast+
  have t ≠ Null ∧ u ≠ Null
    using tu con-char by fastforce
  moreover have Dom t = Dom u ∧ Cod t = Cod u
    using tu 1 con-char by blast
  moreover have Trn t = Trn u
```

```
    by (metis Cod-resid Dom-resid Trn-resid calculation(2)
        hom.extensional ide-char prfx-implies-con tu)
  ultimately show t = u
    by (cases t; cases u) auto
qed

lemma is-extensional-rts:
shows extensional-rts resid
  ..

lemma arr-MkArr [intro]:
assumes a ∈ Obj and b ∈ Obj
and residuation.arr (HOM_EC a b) t
shows V.arr (MkArr a b t)
  using assms arr-char by auto

lemma arr-eqI:
assumes t ≠ V.null and u ≠ V.null
and Dom t = Dom u and Cod t = Cod u and Trn t = Trn u
shows t = u
  using assms null-char
  by (metis Cod.elims Dom.simps(1) Trn.simps(1))

lemma MkArr-Trn:
assumes V.arr t
shows t = MkArr (Dom t) (Cod t) (Trn t)
  using assms null-char V.not-arr-null
  by (intro arr-eqI) auto

lemma src-char:
shows V.src t = (if V.arr t
                 then MkArr (Dom t) (Cod t)
                        (weakly-extensional-rts.src
                          (HOM_EC (Dom t) (Cod t)) (Trn t))
                 else Null)
proof (cases V.arr t)
  show ¬ V.arr t ⟹ ?thesis
    using null-char V.src-def by presburger
  assume t: V.arr t
  show ?thesis
  proof −
    interpret Hom: extensional-rts ‹RTS.Rts (Hom (Dom t) (Cod t))›
      using t ide-Hom arr-char RTS.ide-char RTS.arrD by metis
    have V.ide (MkArr (Dom t) (Cod t) (Hom.src (Trn t)))
      unfolding ide-char
      using t arr-char by auto
    moreover have t ⌢ MkArr (Dom t) (Cod t) (Hom.src (Trn t))
      using t MkArr-Trn con-char Con-def by auto
    ultimately show ?thesis
```

```
    using V.sources-char V.src-in-sources by auto
  qed
qed
```

Here we use the composition operation of the original RTS-enriched category to define horizontal composition of transitions of the global RTS. Note that a pair of transitions (which comprise a transition of a product RTS) must be "packed" into a single transition of the RTS underlying a product object, before the composition operation can be applied.

```
definition hcomp  (infixr ⋆ 53)
where t ⋆ u ≡
      if V.arr t ∧ V.arr u ∧ Dom t = Cod u
      then MkArr (Dom u) (Cod t)
                 (RTS.Map (Comp (Dom u) (Cod u) (Cod t))
                          (RTS.Pack (Hom (Dom t) (Cod t))
                                    (Hom (Dom u) (Cod u))
                                    (Trn t, Trn u)))
           else V.null

lemma arr-hcomp:
assumes V.arr t and V.arr u and Dom t = Cod u
shows V.arr (t ⋆ u)
proof −
  let ?a = Dom u and ?b = Cod u and ?c = Cod t
  have a: ?a ∈ Obj and b: ?b ∈ Obj and c: ?c ∈ Obj
    using assms arr-char by auto
  interpret HOM-ab: hom-rts arr-type Obj Hom Id Comp ?a ?b
    using a b by unfold-locales auto
  interpret HOM-bc: hom-rts arr-type Obj Hom Id Comp ?b ?c
    using b c by unfold-locales auto
  interpret HOM-ac: hom-rts arr-type Obj Hom Id Comp ?a ?c
    using a c by unfold-locales auto
  interpret bcxab: extensional-rts ‹RTS.Rts (Hom ?b ?c ⊗ Hom ?a ?b)›
    using a b c by auto
  interpret BCxAB: product-rts ‹HOM_EC ?b ?c› ‹HOM_EC ?a ?b› ..
  interpret Pack: simulation
                    BCxAB.resid ‹RTS.Rts (Hom ?b ?c ⊗ Hom ?a ?b)›
                    ‹RTS.Pack (Hom ?b ?c) (Hom ?a ?b)›
    using a b c RTS.simulation-Pack by blast
  let ?tu = MkArr ?a ?c
              (RTS.Map (Comp ?a ?b ?c)
                 (RTS.Pack (Hom ?b ?c) (Hom ?a ?b) (Trn t, Trn u)))
  have arr-Trn-u: HOM-ab.arr (Trn u)
    using assms arr-char by blast
  have arr-Trn-t: HOM-bc.arr (Trn t)
    using assms arr-char by simp
  have V.arr ?tu
  proof
    show Dom u ∈ Obj
```

```
      using assms arr-char by auto
    show Cod t ∈ Obj
      using assms arr-char by simp
    show HOM-ac.arr (RTS.Map (Comp ?a ?b ?c)
                     (RTS.Pack (Hom ?b ?c) (Hom ?a ?b)
                       (Trn t, Trn u)))
    proof −
      have HOM-ac.arr (RTS.Map (Comp ?a ?b ?c)
                       (RTS.Pack (Hom ?b ?c) (Hom ?a ?b)
                         (Trn t, Trn u)))
      proof −
        have RTS.in-hom (Comp (Dom u) (Cod u) (Cod t))
                        (Hom ?b ?c ⊗ Hom ?a ?b)
                        (Hom (Dom u) (Cod t))
          using a b c Comp-in-hom by auto
        moreover have bcxab.arr
                        (RTS.Pack (Hom ?b ?c) (Hom ?a ?b) (Trn t, Trn u))
          using a b c arr-Trn-t arr-Trn-u Pack.preserves-reflects-arr
          by auto
        ultimately show ?thesis
          by (metis (mono-tags, lifting) HOM-ac.arrI RTS.in-homE
              bcxab.arrE RTS.arrD(3) simulation.preserves-con)
      qed
      thus ?thesis by simp
    qed
  qed
  thus ?thesis
    using assms hcomp-def by simp
qed

lemma Dom-hcomp [simp]:
assumes V.arr t and V.arr u and Dom t = Cod u
shows Dom (t ⋆ u) = Dom u
  using assms hcomp-def by auto

lemma Cod-hcomp [simp]:
assumes V.arr t and V.arr u and Dom t = Cod u
shows Cod (t ⋆ u) = Cod t
  using assms hcomp-def by auto

lemma Trn-hcomp [simp]:
assumes V.arr t and V.arr u and Dom t = Cod u
shows Trn (t ⋆ u) =
      RTS.Map (Comp (Dom u) (Cod u) (Cod t))
        (RTS.Pack (Hom (Cod u) (Cod t)) (Hom (Dom u) (Cod u))
        (Trn t, Trn u))
  using assms hcomp-def by auto

lemma hcomp-Null [simp]:
```

shows $t \star Null = Null$ **and** $Null \star u = Null$
using *hcomp-def null-char* **by** *fastforce+*

sublocale H: *Category.partial-magma hcomp*
using *hcomp-def*
by (*metis Category.partial-magma.intro V.not-arr-null*)

lemma *H-null-char*:
shows $H.null = V.null$
using *hcomp-def*
by (*metis H.null-eqI V.not-arr-null*)

sublocale H: *partial-composition hcomp* **..**

lemma *H-composable-char*:
shows $t \star u \neq V.null \longleftrightarrow V.arr\ t \wedge V.arr\ u \wedge Dom\ t = Cod\ u$
using *hcomp-def null-char*
by (*cases t*; *cases u*) *auto*

definition *horizontal-unit*
where *horizontal-unit a* $\equiv$
$V.arr\ a \wedge Dom\ a = Cod\ a \wedge$
$(\forall t.\ (V.arr\ t \wedge Dom\ t = Cod\ a \longrightarrow t \star a = t) \wedge$
$(V.arr\ t \wedge Dom\ a = Cod\ t \longrightarrow a \star t = t))$

lemma *H-ide-char*:
shows $H.ide\ a \longleftrightarrow$ *horizontal-unit a*
using *H.ide-def H-composable-char H-null-char horizontal-unit-def*
by *fastforce*

Each $A \in Obj$ determines a corresponding identity for horizontal composition; namely, the transition of $HOM_{EC}\ A\ A$ obtained by evaluating the simulation «$Id\ A : One \to Hom\ A\ A$» at the unique arrow *RTS.One.the-arr* of the underlying one-arrow RTS of *One*.

abbreviation *mkobj*
where *mkobj A* $\equiv$ *MkArr A A* (*RTS.Map* (*Id A*) *RTS.One.the-arr*)

lemma *Id-yields-horiz-ide*:
assumes $A \in Obj$
shows *H.ide* (*mkobj A*)
proof (*unfold H.ide-def*, *intro allI conjI impI*)
interpret *HOM-A-A*: *hom-rts arr-type Obj Hom Id Comp A A*
using *assms* **by** *unfold-locales*
interpret *Id-A*: *simulation* ‹*RTS.Rts* **1**› ‹$HOM_{EC}\ A\ A$› ‹*RTS.Map* (*Id A*)›
using *assms Id-in-hom* [*of A*] *RTS.arrD*(*3*) [*of Id A*] *RTS.unity-agreement*
by *auto*
let *?a* = *mkobj A*
have *HOM-A-A.ide* (*RTS.Map* (*Id A*) *RTS.One.the-arr*)
using *assms Id-A.preserves-ide* $RTS.One.ide\text{-}char_{1RTS}$ **by** *auto*

hence *a*: *V.arr ?a* ∧ *Dom ?a* = *Cod ?a*
using *assms ide-char* **by** *auto*
show *mkobj A* ⋆ *mkobj A* ≠ *H.null*
by (*metis Cod.simps*(*1*) *Dom.simps*(*1*) *HOM-A-A.ide-implies-arr H-null-char*
V.not-arr-null ‹*HOM-A-A.ide* (*RTS.Map* (*Id A*) *RTS.One.the-arr*)›
arr-MkArr arr-hcomp assms)
fix *t*
show *t* ⋆ *?a* ≠ *H.null* ⟹ *t* ⋆ *?a* = *t*
proof −
assume *t* ⋆ *?a* ≠ *H.null*
hence *t*: *V.arr t* ∧ *Dom t* = *Cod ?a*
by (*simp add*: *H-composable-char H-null-char*)
show *t* ⋆ *?a* = *t*
proof −
interpret *HOM-AB*: *hom-rts arr-type Obj Hom Id Comp A* ‹*Cod t*›
using *assms t arr-char*
by *unfold-locales auto*
interpret *HOM-AB*: *simulation*
‹HOM_{EC} *A* (*Cod t*)› ‹HOM_{EC} *A* (*Cod t*)›
‹*RTS.Map* (*Hom A* (*Cod t*))›
using *assms t arr-char RTS.arrD* [*of Hom A* (*Cod t*)] **by** *simp*
interpret *HOM-ABxI*: *product-rts* ‹HOM_{EC} *A* (*Cod t*)› ‹*RTS.One.resid*›
..
interpret *HOM-ABxId-A*: *product-simulation*
‹HOM_{EC} *A* (*Cod t*)› ‹*RTS.Rts* **1**›
‹HOM_{EC} *A* (*Cod t*)› ‹HOM_{EC} *A A*›
‹*RTS.Map* (*Hom A* (*Cod t*))› ‹*RTS.Map* (*Id A*)›
..
interpret *PU*: *inverse-simulations*
‹*RTS.Rts* (*Hom A* (*Cod t*) ⊗ **1**)› *HOM-ABxI.resid*
‹*RTS.Pack* (*Hom A* (*Cod t*)) **1**›
‹*RTS.Unpack* (*Hom A* (*Cod t*)) **1**›
using *assms t arr-char RTS.ide-one*
RTS.inverse-simulations-Pack-Unpack
[*of Hom A* (*Cod t*) *RTS.one*]
by *simp*
have *t* ⋆ *?a* ≠ *Null*
using *a t arr-hcomp null-char arr-char* **by** *blast*
moreover have *t* ≠ *Null*
using *t null-char arr-char* **by** *blast*
moreover have *Dom t* = *Dom* (*hcomp t ?a*)
using *a t hcomp-def* **by** *fastforce*
moreover have *Cod t* = *Cod* (*hcomp t ?a*)
using *a t hcomp-def* **by** *fastforce*
moreover have *Trn t* = *Trn* (*hcomp t ?a*)
proof −
have *Trn* (*t* ⋆ *?a*) =
(*RTS.Map* (*Comp A A* (*Cod t*)) ∘
(*RTS.Pack* (*Hom A* (*Cod t*)) (*Hom A A*) ∘

```
        HOM-ABxId-A.map))
      (Trn t, RTS.One.the-arr)
  using HOM-ABxId-A.map-simp RTS.Map-ide a arr-char t by force
also have ... =
    (RTS.Map (Comp A A (Cod t)) ∘
      (RTS.Map (Hom A (Cod t) ⊗ Id A) ∘
        RTS.Pack (Hom A (Cod t)) 1))
      (Trn t, RTS.One.the-arr)
proof −
  have RTS.Map (Hom A (Cod t) ⊗ Id A) ∘
        RTS.Pack (Hom A (Cod t)) 1 =
      (RTS.Pack (Hom A (Cod t)) (Hom A A) ∘
        HOM-ABxId-A.map ∘
          RTS.Unpack (Hom A (Cod t)) 1) ∘
        RTS.Pack (Hom A (Cod t)) 1
    by (metis (no-types, lifting) HOM-AB.b Id-in-hom RTS.Map-prod
    RTS.ideD(1−3) RTS.in-homE RTS.unity-agreement assms ide-Hom)
  also have ... =
      (RTS.Pack (Hom A (Cod t)) (Hom A A) ∘
        HOM-ABxId-A.map) ∘
          (RTS.Unpack (Hom A (Cod t)) 1 ∘
            RTS.Pack (Hom A (Cod t)) 1)
    by auto
  also have ... = RTS.Pack (Hom A (Cod t)) (Hom A A) ∘
                    (HOM-ABxId-A.map ∘ I HOM-ABxI.resid)
    using PU.inv by auto
  also have ... = RTS.Pack (Hom A (Cod t)) (Hom A A) ∘
                    HOM-ABxId-A.map
    using comp-simulation-identity
          [of HOM-ABxI.resid HOM-ABxId-A.B1xB0.resid
            HOM-ABxId-A.map]
        HOM-ABxId-A.simulation-axioms
    by auto
  finally have RTS.Map (Hom A (Cod t) ⊗ Id A) ∘
                 RTS.Pack (Hom A (Cod t)) 1 =
          RTS.Pack (Hom A (Cod t)) (Hom A A) ∘ HOM-ABxId-A.map
    by blast
  thus ?thesis by simp
qed
also have ... =
    ((RTS.Map (Comp A A (Cod t)) ∘
       RTS.Map (Hom A (Cod t) ⊗ Id A)) ∘
         RTS.Pack (Hom A (Cod t)) 1)
      (Trn t, RTS.One.the-arr)
  by auto
also have ... =
    (RTS.Map (Comp A A (Cod t)) · (Hom A (Cod t) ⊗ Id A)) ∘
      RTS.Pack (Hom A (Cod t)) 1)
      (Trn t, RTS.One.the-arr)
```

```
    by (metis (no-types, lifting) Comp-Hom-Id HOM-AB.b RTS.CMC.arr-runit
          RTS.Map-comp assms ide-Hom prod.sel(1−2))
    also have ... = (RTS.Map r[Hom A (Cod t)] ∘
                     RTS.Pack (Hom A (Cod t)) 1)
                    (Trn t, RTS.One.the-arr)
      using assms t arr-char Comp-Hom-Id
      by (simp add: RTS.runit-agreement)
    also have ... = (HOM-ABxI.P₁ ∘
                     (RTS.Unpack (Hom A (Cod t)) 1 ∘
                       RTS.Pack (Hom A (Cod t)) 1))
                    (Trn t, RTS.One.the-arr)
      using assms t arr-char RTS.Map-runit by auto
    also have ... = HOM-ABxI.P₁ (Trn t, RTS.One.the-arr)
      using assms t arr-char PU.inv HOM-ABxI.P₁.simulation-axioms
            comp-simulation-identity
              [of HOM-ABxI.resid - HOM-ABxI.P₁]
      by simp
    also have ... = Trn t
    using t arr-char HOM-ABxI.P₁-def HOM-ABxI.arr-char RTS.One.arr-char
      by auto
    finally show ?thesis by auto
  qed
  ultimately show t ⋆ ?a = t
    apply (cases t)
    by auto (metis Cod.simps(1) Dom.simps(1) Trn.elims)
 qed
qed
show ?a ⋆ t ≠ H.null ⟹ ?a ⋆ t = t
proof −
 assume ?a ⋆ t ≠ H.null
 hence t: V.arr t ∧ Dom ?a = Cod t
   by (simp add: H-composable-char H-null-char)
 show ?a ⋆ t = t
 proof −
   interpret HOM-BA: hom-rts arr-type Obj Hom Id Comp ‹Dom t› A
     using assms t arr-char
     by unfold-locales auto
   interpret HOM-BA: simulation
                     ‹HOM_EC (Dom t) A› ‹HOM_EC (Dom t) A›
                     ‹RTS.Map (Hom (Dom t) A)›
     using assms t arr-char RTS.arrD(3) [of Hom (Dom t) A] by auto
   interpret IxHOM-BA: product-rts
                       ‹RTS.Rts (RTS.dom 1)›
                       ‹HOM_EC (Dom t) A›
  by (metis HOM-BA.small-rts-axioms Id-A.simulation-axioms RTS.ideD(2)
        RTS.ide-one product-rts.intro simulation-def small-rts-def)
   interpret Id-AxHOM-BA: product-simulation
                          ‹RTS.Rts (RTS.dom 1)› ‹HOM_EC (Dom t) A›
                          ‹HOM_EC A A› ‹HOM_EC (Dom t) A›
```

```
                    ‹RTS.Map (Id A)› ‹RTS.Map (Hom (Dom t) A)›
  by (metis (mono-tags, lifting) HOM-BA.simulation-axioms
      Id-A.simulation-axioms RTS.ideD(2) RTS.ide-one product-rts.intro
      product-simulation-def simulation-def)
interpret PU: inverse-simulations
              ‹RTS.Rts (RTS.dom (𝟭 ⊗ Hom (Dom t) A))›
              IxHOM-BA.resid
              ‹RTS.Pack 𝟭 (Hom (Dom t) A)›
              ‹RTS.Unpack 𝟭 (Hom (Dom t) A)›
  using assms t arr-char RTS.ide-one
        RTS.inverse-simulations-Pack-Unpack [of 𝟭 Hom (Dom t) A]
  by simp
have ?a ⋆ t ≠ Null
  using a t arr-hcomp null-char arr-char by blast
moreover have t ≠ Null
  using t null-char arr-char by blast
moreover have Dom t = Dom (hcomp ?a t)
  using a t hcomp-def by fastforce
moreover have Cod t = Cod (hcomp ?a t)
  using a t hcomp-def by fastforce
moreover have Trn t = Trn (hcomp ?a t)
proof −
  have Trn (?a ⋆ t) =
       RTS.Map (Comp (Dom t) A A)
         (RTS.Pack (Hom A A) (Hom (Dom t) A)
           (RTS.Map (Id A) RTS.One.the-arr, Trn t))
    using a t hcomp-def by simp
  also have ... =
       (RTS.Map (Comp (Dom t) A A) ∘
         (RTS.Pack (Hom A A) (Hom (Dom t) A) ∘
           Id-AxHOM-BA.map))
         (RTS.One.the-arr, Trn t)
    using assms t arr-char RTS.Map-ide Id-AxHOM-BA.map-simp
          Id-A.preserves-reflects-arr RTS.ide-one a
    by auto
  also have ... =
       (RTS.Map (Comp (Dom t) A A) ∘
         (RTS.Map (Id A ⊗ Hom (Dom t) A) ∘
           RTS.Pack 𝟭 (Hom (Dom t) A)))
         (RTS.One.the-arr, Trn t)
  proof −
    have RTS.Map (Id A ⊗ Hom (Dom t) A) ∘
           RTS.Pack 𝟭 (Hom (Dom t) A) =
         (RTS.Pack (Hom A A) (Hom (Dom t) A) ∘
           Id-AxHOM-BA.map ∘
             RTS.Unpack 𝟭 (Hom (Dom t) A)) ∘
           RTS.Pack 𝟭 (Hom (Dom t) A)
      by (metis (no-types, lifting) HOM-BA.a Id-in-hom RTS.Map-prod
          RTS.ide-char RTS.ide-one RTS.in-homE RTS.unity-agreement
```

```
      assms ide-Hom)
  also have ... =
      (RTS.Pack (Hom A A) (Hom (Dom t) A) ∘
        Id-AxHOM-BA.map) ∘
          (RTS.Unpack 1 (Hom (Dom t) A) ∘
            RTS.Pack 1 (Hom (Dom t) A))
    by auto
  also have ... = (RTS.Pack (Hom A A) (Hom (Dom t) A) ∘
                    Id-AxHOM-BA.map) ∘
                      I IxHOM-BA.resid
  proof −
    have RTS.Unpack 1 (Hom (Dom t) A) ∘
           RTS.Pack 1 (Hom (Dom t) A) =
         I IxHOM-BA.resid
      using PU.inv by fastforce
    thus ?thesis by simp
  qed
  also have ... = RTS.Pack (Hom A A) (Hom (Dom t) A) ∘
                    (Id-AxHOM-BA.map ∘ I IxHOM-BA.resid)
    by auto
  also have ... = RTS.Pack (Hom A A) (Hom (Dom t) A) ∘
                    Id-AxHOM-BA.map
    using comp-simulation-identity
          [of IxHOM-BA.resid Id-AxHOM-BA.B1xB0.resid
              Id-AxHOM-BA.map]
        Id-AxHOM-BA.simulation-axioms
    by auto
  finally have RTS.Map (Id A ⊗ Hom (Dom t) A) ∘
                 RTS.Pack 1 (Hom (Dom t) A) =
         RTS.Pack (Hom A A) (Hom (Dom t) A) ∘ Id-AxHOM-BA.map
    by blast
  thus ?thesis by simp
qed
also have ... =
    ((RTS.Map (Comp (Dom t) A A) ∘
       RTS.Map (Id A ⊗ Hom (Dom t) A)) ∘
         RTS.Pack 1 (Hom (Dom t) A))
       (RTS.One.the-arr, Trn t)
  by auto
also have ... =
    (RTS.Map (Comp (Dom t) A A · (Id A ⊗ Hom (Dom t) A)) ∘
      RTS.Pack 1 (Hom (Dom t) A))
      (RTS.One.the-arr, Trn t)
  using assms t Comp-Id-Hom HOM-BA.a
      RTS.Map-comp
        [of Comp (Dom t) A A Id A ⊗ Hom (Dom t) A]
  by auto
also have ... = (RTS.Map 1[Hom (Dom t) A] ∘
                   RTS.Pack 1 (Hom (Dom t) A))
```

```
                    (RTS.One.the-arr, Trn t)
        using assms t arr-char Comp-Id-Hom
        by (simp add: RTS.lunit-agreement)
      also have ... = (IxHOM-BA.P₀ ∘
                    RTS.Unpack RTS.one (Hom (Dom t) A) ∘
                    RTS.Pack 1 (Hom (Dom t) A))
                    (RTS.One.the-arr, Trn t)
        using assms t arr-char RTS.Map-lunit RTS.ide-one by auto
      also have ... = (IxHOM-BA.P₀ ∘
                    (RTS.Unpack 1 (Hom (Dom t) A) ∘
                      RTS.Pack 1 (Hom (Dom t) A)))
                    (RTS.One.the-arr, Trn t)
        by auto
      also have ... = IxHOM-BA.P₀ (RTS.One.the-arr, Trn t)
        using assms t arr-char PU.inv IxHOM-BA.P₀.simulation-axioms
            comp-simulation-identity
              [of IxHOM-BA.resid - IxHOM-BA.P₀]
        by simp
      also have ... = Trn t
        using t arr-char IxHOM-BA.P₀-def IxHOM-BA.arr-char
            Id-A.preserves-reflects-arr RTS.ide-one a
        by auto
      finally show ?thesis by auto
    qed
    ultimately show ?thesis
      apply (cases t)
       apply auto[1]
      by (metis Cod.simps(1) Dom.simps(1) Trn.elims)
  qed
 qed
qed

lemma H-ide-is-V-ide:
assumes H.ide a
shows V.ide a
proof −
 have 1: V.arr a ∧ a = mkobj (Dom a)
   by (metis assms Cod.simps(1) H-ide-char Id-yields-horiz-ide
       horizontal-unit-def arr-char)
 interpret HOM-AA: hom-rts arr-type Obj Hom Id Comp ‹Dom a› ‹Dom a›
   using assms 1 arr-char by unfold-locales simp
 interpret Id-A: simulation ‹RTS.Dom 1› ‹HOM_EC (Dom a) (Dom a)›
               ‹RTS.Map (Id (Dom a))›
   using 1 arr-char Id-in-hom RTS.arrD(3) RTS.ide-one by force
 have V.ide (MkArr (Dom a) (Dom a)
             (RTS.Map (Id (Dom a)) RTS.One.the-arr))
 proof −
   have Trn a = RTS.Map (Id (Dom a)) RTS.One.the-arr
     using 1 by (metis Trn.simps(1))
```

```
    moreover have 2: HOM-AA.ide ...
      using Id-A.preserves-ide RTS.One.ide-char_{1RTS} RTS.ide-one by force
    ultimately show ?thesis
      by (metis 1 Cod.simps(1) arr-char ide-char)
  qed
  thus V.ide a
    using 1 ide-char by metis
qed

lemma H-domains-char:
shows H.domains t = {a. V.arr t ∧ a = mkobj (Dom t)}
proof
  show {a. V.arr t ∧ a = mkobj (Dom t)} ⊆ H.domains t
  proof
    fix a
    assume a: a ∈ {a. V.arr t ∧ a = mkobj (Dom t)}
    have H.ide a
      using a arr-char Id-yields-horiz-ide by force
    thus a ∈ H.domains t
      using a H.domains-def H.ide-def H-composable-char H-null-char
      by force
  qed
  show H.domains t ⊆ {a. V.arr t ∧ a = mkobj (Dom t)}
  proof
    fix a
    assume a: a ∈ H.domains t
    have 1: H.ide a ∧ hcomp t a ≠ H.null
      using a H.domains-def by blast
    have a = mkobj (Dom t)
      using a 1 H.ide-def
      by (metis (no-types, lifting) Dom.simps(1) H-composable-char
          H-null-char Id-yields-horiz-ide arr-char)
    moreover have V.arr t
      using 1 hcomp-def null-char H-null-char by metis
    ultimately show a ∈ {a. V.arr t ∧ a = mkobj (Dom t)}
      by blast
  qed
qed

lemma H-codomains-char:
shows H.codomains t = {a. V.arr t ∧ a = mkobj (Cod t)}
proof
  show {a. V.arr t ∧ a = mkobj (Cod t)} ⊆ H.codomains t
  proof
    fix a
    assume a: a ∈ {a. V.arr t ∧ a = mkobj (Cod t)}
    have H.ide a
      using a arr-char Id-yields-horiz-ide by force
    thus a ∈ H.codomains t
```

```
    using a H.codomains-def H.ide-def H-composable-char H-null-char
    by force
  qed
  show H.codomains t ⊆ {a. V.arr t ∧ a = mkobj (Cod t)}
  proof
    fix a
    assume a: a ∈ H.codomains t
    have 1: H.ide a ∧ hcomp a t ≠ H.null
      using a H.codomains-def by blast
    have a = mkobj (Cod t)
      using a 1 H.ide-def
      by (metis (no-types, lifting) Cod.simps(1) H-composable-char
          H-null-char Id-yields-horiz-ide arr-char)
    moreover have V.arr t
      using 1 hcomp-def null-char H-null-char by metis
    ultimately show a ∈ {a. V.arr t ∧ a = mkobj (Cod t)}
      by blast
  qed
qed

lemma H-arr-char:
shows H.arr t ⟷ t ≠ Null ∧ Dom t ∈ Obj ∧ Cod t ∈ Obj ∧
                 residuation.arr (HOM_EC (Dom t) (Cod t)) (Trn t)
proof
  assume t: H.arr t
  interpret HOM: hom-rts arr-type Obj Hom Id Comp ‹Dom t› ‹Cod t›
    using t
    by unfold-locales
       (auto simp add: H.arr-def H-codomains-char H-domains-char arr-char)
  show t ≠ Null ∧ Dom t ∈ Obj ∧ Cod t ∈ Obj ∧ HOM.arr (Trn t)
    using t H.arr-def H-codomains-char H-domains-char arr-char by auto
  next
  assume t: t ≠ Null ∧ Dom t ∈ Obj ∧ Cod t ∈ Obj ∧
            residuation.arr (HOM_EC (Dom t) (Cod t)) (Trn t)
  have V.arr t
    using t arr-char by blast
  thus H.arr t
    using t H.arr-def H-codomains-char H-domains-char arr-char
          Id-yields-horiz-ide
    by blast
qed

lemma H-seq-char:
shows H.seq t u ⟷ V.arr t ∧ V.arr u ∧ Dom t = Cod u
  by (metis H-arr-char H-composable-char arr-char arr-hcomp null-char)

sublocale H: category hcomp
proof
  show ⋀t u. hcomp t u ≠ H.null ⟹ H.seq t u
```

using *hcomp-def H-seq-char H-null-char null-char*
by *auto argo+*
show $\bigwedge t.\ (H.domains\ t \neq \{\}) = (H.codomains\ t \neq \{\})$
by (*simp add*: *H-codomains-char H-domains-char*)
show $\bigwedge h\ g\ f.\ [\![H.seq\ h\ g;\ H.seq\ (h \star g)\ f]\!] \Longrightarrow H.seq\ g\ f$
using *H-seq-char*
by (*metis Dom.simps*(*1*) *hcomp-def*)
show $\bigwedge h\ g\ f.\ [\![H.seq\ h\ (g \star f);\ H.seq\ g\ f]\!] \Longrightarrow H.seq\ h\ g$
using *H-seq-char*
by (*metis Cod.simps*(*1*) *hcomp-def*)
show $\bigwedge g\ f\ h.\ [\![H.seq\ g\ f;\ H.seq\ h\ g]\!] \Longrightarrow H.seq\ (h \star g)\ f$
using *H-seq-char*
by (*metis Dom.simps*(*1*) *arr-hcomp hcomp-def*)
show $\bigwedge t\ u\ v.\ [\![H.seq\ u\ v;\ H.seq\ t\ u]\!] \Longrightarrow (t \star u) \star v = t \star u \star v$
proof (*intro arr-eqI*)
fix *t u v*
assume *tu*: *H.seq t u* **and** *uv*: *H.seq u v*
show $(t \star u) \star v \neq V.null$
using *tu uv arr-hcomp H-seq-char H-composable-char null-char* **by** *auto*
show $t \star u \star v \neq V.null$
using *tu uv arr-hcomp H-seq-char H-composable-char null-char* **by** *auto*
show $Dom\ ((t \star u) \star v) = Dom\ (t \star u \star v)$
using *tu uv arr-hcomp H-seq-char H-composable-char null-char* **by** *simp*
show $Cod\ ((t \star u) \star v) = Cod\ (t \star u \star v)$
using *tu uv arr-hcomp H-seq-char H-composable-char null-char* **by** *simp*
show $Trn\ ((t \star u) \star v) = Trn\ (t \star u \star v)$
proof −
let *?A* = *Dom v* **and** *?B* = *Cod v* **and** *?C* = *Cod u* **and** *?D* = *Cod t*
have *A*: *?A* ∈ *Obj* **and** *B*: *?B* ∈ *Obj* **and** *C*: *?C* ∈ *Obj* **and** *D*: *?D* ∈ *Obj*
using *tu uv arr-char arr-char* [*of u*] *arr-char* [*of v*] *H-seq-char* **by** *blast+*
interpret *AB*: *hom-rts arr-type Obj Hom Id Comp ?A ?B*
using *A B* **by** *unfold-locales*
interpret *BC*: *hom-rts arr-type Obj Hom Id Comp ?B ?C*
using *B C* **by** *unfold-locales*
interpret *CD*: *hom-rts arr-type Obj Hom Id Comp ?C ?D*
using *C D* **by** *unfold-locales*
interpret *CDxBC*: *product-rts* ‹HOM_{EC} *?C ?D*› ‹HOM_{EC} *?B ?C*› **..**
interpret *BCxAB*: *product-rts* ‹HOM_{EC} *?B ?C*› ‹HOM_{EC} *?A ?B*› **..**
interpret *CDxBC-x-AB*: *product-rts CDxBC.resid* ‹HOM_{EC} *?A ?B*› **..**
interpret *CD-x-BCxAB*: *product-rts* ‹HOM_{EC} *?C ?D*› *BCxAB.resid* **..**
interpret *I-AB*: *simulation*
‹HOM_{EC} *?A ?B*› ‹HOM_{EC} *?A ?B*›
‹*RTS.Map* (*Hom ?A ?B*)›
using *A B* **by** (*metis RTS.arrD*(*3*) *RTS.ideD*(*1*−*3*) *ide-Hom*)
interpret *I-BC*: *simulation*
‹HOM_{EC} *?B ?C*› ‹HOM_{EC} *?B ?C*›
‹*RTS.Map* (*Hom ?B ?C*)›
using *B C* **by** (*metis RTS.arrD*(*3*) *RTS.ideD*(*1*−*3*) *ide-Hom*)
interpret *I-CD*: *simulation*

```
                ‹HOM_EC ?C ?D› ‹HOM_EC ?C ?D›
                ‹RTS.Map (Hom ?C ?D)›
  using C D by (metis RTS.arrD(3) RTS.ideD(1−3) ide-Hom)
interpret I-CDxI-BC: product-simulation
                  ‹HOM_EC ?C ?D› ‹HOM_EC ?B ?C›
                  ‹HOM_EC ?C ?D› ‹HOM_EC ?B ?C›
                  ‹RTS.Map (Hom ?C ?D)› ‹RTS.Map (Hom ?B ?C)›
  ..
interpret PU-BC-AB: inverse-simulations
                  ‹RTS.Dom (Hom ?B ?C ⊗ Hom ?A ?B)›
                  BCxAB.resid
                  ‹RTS.Pack (Hom ?B ?C) (Hom ?A ?B)›
                  ‹RTS.Unpack (Hom ?B ?C) (Hom ?A ?B)›
  using A B C D RTS.inverse-simulations-Pack-Unpack by simp
interpret PU-CD-BC: inverse-simulations
                  ‹RTS.Dom (Hom ?C ?D ⊗ Hom ?B ?C)›
                  CDxBC.resid
                  ‹RTS.Pack (Hom ?C ?D) (Hom ?B ?C)›
                  ‹RTS.Unpack (Hom ?C ?D) (Hom ?B ?C)›
  using A B C D RTS.inverse-simulations-Pack-Unpack by simp
interpret bcxab: extensional-rts ‹RTS.Dom (Hom ?B ?C ⊗ Hom ?A ?B)›
  using A B C D RTS.arrD
      RTS.ide-char [of Hom ?B ?C ⊗ Hom ?A ?B]
  by blast
interpret CD-x-bcxab: product-rts
                  ‹HOM_EC ?C ?D›
                  ‹RTS.Dom (Hom ?B ?C ⊗ Hom ?A ?B)›
  ..
interpret cdxbc: extensional-rts ‹RTS.Dom (Hom ?C ?D ⊗ Hom ?B ?C)›
  using A B C D RTS.arrD
      RTS.ide-char [of Hom ?C ?D ⊗ Hom ?B ?C]
  by blast
interpret cdxbc-x-AB: product-rts
                  ‹RTS.Dom (Hom ?C ?D ⊗ Hom ?B ?C)›
                  ‹HOM_EC ?A ?B›
  ..
interpret PU-cdxbc-x-AB: inverse-simulations
                ‹RTS.Dom ((Hom ?C ?D ⊗ Hom ?B ?C) ⊗ Hom ?A ?B)›
                  cdxbc-x-AB.resid
                 ‹RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B)›
                  ‹RTS.Unpack
                    (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B)›
  using A B C D RTS.inverse-simulations-Pack-Unpack by auto
interpret PU-CD-x-bcxab: inverse-simulations
                    ‹RTS.Dom
                      (Hom ?C ?D ⊗ Hom ?B ?C ⊗ Hom ?A ?B)›
                    CD-x-bcxab.resid
                    ‹RTS.Pack
                      (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B)›
```

```
                    ‹RTS.Unpack
                      (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B)›
  using A B C D RTS.inverse-simulations-Pack-Unpack by auto
interpret I-AB: identity-simulation ‹HOM_EC ?A ?B› ..
interpret U-CD-BC-x-I-AB: product-simulation
                    ‹RTS.Dom (Hom ?C ?D ⊗ Hom ?B ?C)›
                      ‹HOM_EC ?A ?B›
                    CDxBC.resid ‹HOM_EC ?A ?B›
                    ‹RTS.Unpack (Hom ?C ?D) (Hom ?B ?C)›
                    ‹I (HOM_EC ?A ?B)›
  ..
interpret C-CD-BC: simulation
                  ‹RTS.Dom (Hom ?C ?D ⊗ Hom ?B ?C)›
                  ‹HOM_EC ?B ?D›
                  ‹RTS.Map (Comp ?B ?C ?D)›
  using B C D Comp-in-hom [of ?B ?C ?D] arr-char
  by (metis (no-types, lifting) RTS.arrD(3) RTS.ideD(2) RTS.ide-prod
      RTS.in-homE ide-Hom prod.sel(1−2))
interpret C-CD-BC-x-I-AB: product-simulation
                    ‹RTS.Dom (Hom ?C ?D ⊗ Hom ?B ?C)›
                    ‹HOM_EC ?A ?B›
                    ‹HOM_EC ?B ?D› ‹HOM_EC ?A ?B›
                    ‹RTS.Map (Comp ?B ?C ?D)›
                    ‹RTS.Map (Hom ?A ?B)›
   ..
interpret I-CD: identity-simulation ‹HOM_EC ?C ?D› ..
interpret I-CD-x-P-BC-AB: product-simulation
                    ‹HOM_EC ?C ?D› BCxAB.resid
                    ‹HOM_EC ?C ?D›
                    ‹RTS.Dom (Hom ?B ?C ⊗ Hom ?A ?B)›
                    ‹I (HOM_EC ?C ?D)›
                    ‹RTS.Pack (Hom ?B ?C) (Hom ?A ?B)›
  ..
interpret C-BC-AB: simulation
                  ‹RTS.Dom (Hom ?B ?C ⊗ Hom ?A ?B)›
                  ‹HOM_EC ?A ?C›
                  ‹RTS.Map (Comp ?A ?B ?C)›
  using A B C Comp-in-hom [of ?A ?B ?C] arr-char
  by (metis (no-types, lifting) RTS.arrD(3) RTS.ideD(2) RTS.ide-prod
      RTS.in-homE ide-Hom prod.sel(1) prod.sel(2))
interpret I-CD-x-Comp-ABC: product-simulation
                    ‹HOM_EC ?C ?D›
                    ‹RTS.Dom (Hom ?B ?C ⊗ Hom ?A ?B)›
                    ‹HOM_EC ?C ?D› ‹HOM_EC ?A ?C›
                    ‹RTS.Map (Hom ?C ?D)›
                    ‹RTS.Map (Comp ?A ?B ?C)›
  ..
have Trn ((t ⋆ u) ⋆ v) =
     (RTS.Map (Comp ?A ?B ?D)
```

```
      (RTS.Pack (Hom ?B ?D) (Hom ?A ?B)
        (RTS.Map (Comp ?B ?C ?D)
          (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u)),
        Trn v)))
  using tu uv H-seq-char ‹(t ⋆ u) ⋆ v ≠ V.null› hcomp-def by auto
also have ... =
    RTS.Map (Comp ?A ?B ?D)
      (RTS.Pack (Hom ?B ?D) (Hom ?A ?B)
        (RTS.Map (Comp ?B ?C ?D)
          (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u)),
        I (HOM_EC ?A ?B) (Trn v)))
  using H-seq-char arr-char uv by simp
also have ... =
    RTS.Map (Comp ?A ?B ?D)
      ((RTS.Pack (Hom ?B ?D) (Hom ?A ?B) ∘ C-CD-BC-x-I-AB.map)
        (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u), Trn v))
  using A B C D RTS.Map-ide C-CD-BC-x-I-AB.map-simp H-seq-char
      PU-CD-BC.F.preserves-reflects-arr arr-char tu uv
  by force
also have ... =
    RTS.Map (Comp ?A ?B ?D)
      ((RTS.Map (Comp ?B ?C ?D ⊗ Hom ?A ?B) ∘
          RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B))
        (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u), Trn v))
proof −
  have RTS.Map (Comp ?B ?C ?D ⊗ Hom ?A ?B) ∘
         RTS.Pack
           (RTS.prod (Hom ?C ?D) (Hom ?B ?C)) (Hom ?A ?B) =
       RTS.Pack (Hom ?B ?D) (Hom ?A ?B) ∘ C-CD-BC-x-I-AB.map ∘
         (RTS.Unpack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B) ∘
           RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B))
    using A B C D Comp-in-hom [of ?B ?C ?D]
        RTS.Map-prod [of Comp ?B ?C ?D Hom ?A ?B]
    by fastforce
  also have ... =
      RTS.Pack (Hom ?B ?D) (Hom ?A ?B) ∘
        (C-CD-BC-x-I-AB.map ∘ I cdxbc-x-AB.resid)
    using PU-cdxbc-x-AB.inv
    by auto
  also have ... = RTS.Pack (Hom ?B ?D) (Hom ?A ?B) ∘
                    C-CD-BC-x-I-AB.map
    using comp-simulation-identity
          [of cdxbc-x-AB.resid C-CD-BC-x-I-AB.B1xB0.resid
              C-CD-BC-x-I-AB.map]
        C-CD-BC-x-I-AB.simulation-axioms
    by auto
  finally have RTS.Map (Comp ?B ?C ?D ⊗ Hom ?A ?B) ∘
                 RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B) =
               RTS.Pack (Hom ?B ?D) (Hom ?A ?B) ∘ C-CD-BC-x-I-AB.map
```

```
    by blast
  thus ?thesis by simp
qed
also have ... =
      ((RTS.Map (Comp ?A ?B ?D) ∘
          RTS.Map (Comp ?B ?C ?D ⊗ Hom ?A ?B)) ∘
            RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B))
             (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u), Trn v)
  by auto
also have ... =
      (RTS.Map (Comp ?A ?B ?D · (Comp ?B ?C ?D ⊗ Hom ?A ?B)) ∘
          RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B))
            (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u), Trn v)
proof −
  have RTS.seq (Comp ?A ?B ?D) (Comp ?B ?C ?D ⊗ Hom ?A ?B)
    using A B C D Comp-in-hom
    apply (intro RTS.seqI)
     apply auto[3]
  by (metis RTS.ide-char RTS.in-homE RTS.prod-simps(1,3) ide-Hom)+
  thus ?thesis
    using RTS.Map-comp
          [of Comp ?A ?B ?D Comp ?B ?C ?D ⊗ Hom ?A ?B]
    by argo
qed
also have ... =
      (RTS.Map (Comp ?A ?C ?D · (Hom ?C ?D ⊗ Comp ?A ?B ?C) ·
                  a[Hom ?C ?D, Hom ?B ?C, Hom ?A ?B]) ∘
          RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B))
            (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u), Trn v)
  using A B C D Comp-assoc RTS.assoc-agreement by auto
also have ... =
      ((RTS.Map (Comp ?A ?C ?D) ∘
          RTS.Map ((Hom ?C ?D ⊗ Comp ?A ?B ?C) ·
                     a[Hom ?C ?D, Hom ?B ?C, Hom ?A ?B])) ∘
          RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B))
            (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u), Trn v)
proof −
  have RTS.seq
         (Comp ?A ?C ?D)
         ((Hom ?C ?D ⊗ Comp ?A ?B ?C) ·
            a[Hom ?C ?D, Hom ?B ?C, Hom ?A ?B])
    using A B C D
         Comp-in-hom [of ?A ?C ?D] Comp-in-hom [of ?A ?B ?C]
         RTS.assoc-in-hom
    apply (intro RTS.seqI)
       apply auto[4]
    by fastforce
  thus ?thesis
    using RTS.Map-comp
```

```
          [of Comp ?A ?C ?D
             (Hom ?C ?D ⊗ Comp ?A ?B ?C) ·
               a[Hom ?C ?D, Hom ?B ?C, Hom ?A ?B]]
    by argo
qed
also have ... =
     ((RTS.Map (Comp ?A ?C ?D) ∘
       (RTS.Map (Hom ?C ?D ⊗ Comp ?A ?B ?C) ∘
         RTS.Map
           (RTS.assoc (Hom ?C ?D) (Hom ?B ?C) (Hom ?A ?B)))) ∘
        RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B))
         (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u), Trn v)
proof −
  have RTS.seq (Hom ?C ?D ⊗ Comp ?A ?B ?C)
         a[Hom (Cod u) (Cod t), Hom (Cod v) (Cod u),
           Hom (Dom v) (Cod v)]
    using A B C D Comp-in-hom [of ?A ?B ?C] RTS.assoc-simps(1,3)
          RTS.arrI
    by (intro RTS.seqI) auto
  thus ?thesis
    using RTS.Map-comp
          [of Hom ?C ?D ⊗ Comp ?A ?B ?C
              RTS.assoc (Hom ?C ?D) (Hom ?B ?C) (Hom ?A ?B)]
    by auto
qed
also have ... =
     (RTS.Map (Comp ?A ?C ?D) ∘
       RTS.Map (Hom ?C ?D ⊗ Comp ?A ?B ?C))
      (RTS.Map (RTS.assoc (Hom ?C ?D) (Hom ?B ?C) (Hom ?A ?B))
         (RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B)
           (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u),
            Trn v)))
  using comp-assoc by simp
also have ... =
     (RTS.Map (Comp ?A ?C ?D) ∘
       RTS.Map (Hom ?C ?D ⊗ Comp ?A ?B ?C))
       ((RTS.Pack (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B) ∘
          I-CD-x-P-BC-AB.map ∘
           ASSOC.map
             (HOM_EC ?C ?D) (HOM_EC ?B ?C) (HOM_EC ?A ?B) ∘
             U-CD-BC-x-I-AB.map)
          ((RTS.Unpack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B) ∘
             RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B))
             (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u),
              Trn v)))
  using A B C D RTS.Map-assoc by simp
also have ... =
     (RTS.Map (Comp ?A ?C ?D) ∘
       RTS.Map (Hom ?C ?D ⊗ Comp ?A ?B ?C))
```

```
        ((RTS.Pack (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B) ∘
           I-CD-x-P-BC-AB.map ∘
             ASSOC.map
               (HOM_EC ?C ?D) (HOM_EC ?B ?C) (HOM_EC ?A ?B))
          (U-CD-BC-x-I-AB.map
            (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u),
             Trn v)))
proof −
  have RTS.Unpack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B) ∘
         RTS.Pack (Hom ?C ?D ⊗ Hom ?B ?C) (Hom ?A ?B) =
       I cdxbc-x-AB.resid
    using A B C D PU-cdxbc-x-AB.inv by blast
  moreover have I cdxbc-x-AB.resid
                  (RTS.Pack (Hom ?C ?D) (Hom ?B ?C)
                    (Trn t, Trn u), Trn v) =
                 (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u),
                  Trn v)
  proof −
    have CDxBC.arr (Trn t, Trn u)
      using tu uv arr-char [of t] arr-char [of u] H-seq-char by auto
    moreover have AB.arr (Trn v)
      using uv arr-char H-seq-char by simp
    ultimately have cdxbc-x-AB.arr
                      (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u),
                       Trn v)
      using A B C D PU-CD-BC.F.preserves-reflects-arr by fastforce
    thus ?thesis by auto
  qed
  ultimately show ?thesis by simp
qed
also have ... =
     (RTS.Map (Comp ?A ?C ?D) ∘
        RTS.Map (Hom ?C ?D ⊗ Comp ?A ?B ?C))
        ((RTS.Pack (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B) ∘
            I-CD-x-P-BC-AB.map ∘
              ASSOC.map
                (HOM_EC ?C ?D) (HOM_EC ?B ?C) (HOM_EC ?A ?B))
              ((RTS.Unpack (Hom ?C ?D) (Hom ?B ?C) ∘
                  RTS.Pack (Hom ?C ?D) (Hom ?B ?C))
                  (Trn t, Trn u),
                Trn v))
proof −
  have cdxbc-x-AB.arr
         (RTS.Pack (Hom ?C ?D) (Hom ?B ?C) (Trn t, Trn u), Trn v)
    using tu uv arr-char H-seq-char PU-CD-BC.F.preserves-reflects-arr
    by fastforce
  thus ?thesis
    using A B C D U-CD-BC-x-I-AB.map-simp by fastforce
qed
```

```
also have ... =
    (RTS.Map (Comp ?A ?C ?D) ∘
      RTS.Map (Hom ?C ?D ⊗ Comp ?A ?B ?C))
      ((RTS.Pack (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B) ∘
        I-CD-x-P-BC-AB.map)
        (ASSOC.map
          (HOM_EC ?C ?D) (HOM_EC ?B ?C) (HOM_EC ?A ?B)
          ((Trn t, Trn u), Trn v)))
  using tu uv arr-char H-seq-char PU-CD-BC.inv by fastforce
also have ... =
    (RTS.Map (Comp ?A ?C ?D) ∘
      RTS.Map (Hom ?C ?D ⊗ Comp ?A ?B ?C))
      ((RTS.Pack (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B) ∘
        I-CD-x-P-BC-AB.map)
        (Trn t, Trn u, Trn v))
proof −
  interpret A: ASSOC
              ‹HOM_EC ?C ?D› ‹HOM_EC ?B ?C› ‹HOM_EC ?A ?B›
    ..
  have CDxBC-x-AB.arr ((Trn t, Trn u), Trn v)
    using tu uv arr-char H-seq-char by fastforce
  thus ?thesis
    using A.map-eq by simp
qed
also have ... =
    (RTS.Map (Comp ?A ?C ?D)
      ((RTS.Map (Hom ?C ?D ⊗ Comp ?A ?B ?C) ∘
        RTS.Pack (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B))
        (Trn t, RTS.Pack (Hom ?B ?C) (Hom ?A ?B) (Trn u, Trn v))))
  using tu uv arr-char H-seq-char by fastforce
also have ... =
    RTS.Map (Comp ?A ?C ?D)
    ((RTS.Pack (Hom ?C ?D) (Hom ?A ?C) ∘ I-CD-x-Comp-ABC.map)
        (Trn t, RTS.Pack (Hom ?B ?C) (Hom ?A ?B) (Trn u, Trn v)))
proof −
  have RTS.Map (Hom ?C ?D ⊗ Comp ?A ?B ?C) ∘
        RTS.Pack (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B) =
      (RTS.Pack (Hom ?C ?D) (Hom ?A ?C) ∘
        I-CD-x-Comp-ABC.map ∘
        (RTS.Unpack (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B) ∘
          RTS.Pack (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B)))
    using A B C D RTS.Map-prod [of Hom ?C ?D Comp ?A ?B ?C]
        Comp-in-hom [of ?A ?B ?C]
    by fastforce
  also have ... =
      (RTS.Pack (Hom ?C ?D) (Hom ?A ?C) ∘
        (I-CD-x-Comp-ABC.map ∘ I CD-x-bcxab.resid))
    using PU-CD-x-bcxab.inv by auto
  also have ... = RTS.Pack (Hom ?C ?D) (Hom ?A ?C) ∘
```

```
                    I-CD-x-Comp-ABC.map
      using comp-simulation-identity
            [of CD-x-bcxab.resid - I-CD-x-Comp-ABC.map]
          I-CD-x-Comp-ABC.simulation-axioms
      by simp
    finally have RTS.Map (Hom ?C ?D ⊗ Comp ?A ?B ?C) ∘
                  RTS.Pack (Hom ?C ?D) (Hom ?B ?C ⊗ Hom ?A ?B) =
          RTS.Pack (Hom ?C ?D) (Hom ?A ?C) ∘ I-CD-x-Comp-ABC.map
      by simp
    thus ?thesis by simp
  qed
  also have ... =
      RTS.Map (Comp ?A ?C ?D)
        (RTS.Pack (Hom ?C ?D) (Hom ?A ?C)
          (Trn t,
           RTS.Map (Comp ?A ?B ?C)
             (RTS.Pack (Hom ?B ?C) (Hom ?A ?B) (Trn u, Trn v))))
    using A B C D tu uv arr-char H-seq-char RTS.Map-ide
          I-CD-x-Comp-ABC.map-simp
          [of Trn t RTS.Pack (Hom ?B ?C) (Hom ?A ?B) (Trn u, Trn v)]
    by fastforce
  also have ... = Trn (t ⋆ u ⋆ v)
    using tu uv H-seq-char
    apply auto[1]
    using arr-hcomp hcomp-def by auto
  finally show Trn ((t ⋆ u) ⋆ v) = Trn (t ⋆ u ⋆ v)
    by blast
 qed
qed

lemma is-category:
shows category hcomp
  ..

lemma H-dom-char:
shows H.dom =
      (λt. if H.arr t
           then MkArr (Dom t) (Dom t)
                  (RTS.Map (Id (Dom t)) RTS.One.the-arr)
           else V.null)
  using H-domains-char H.dom-in-domains H.has-domain-iff-arr H.dom-def
        H-null-char
  by auto

lemma H-dom-simp:
assumes V.arr t
shows H.dom t = MkArr (Dom t) (Dom t)
                  (RTS.Map (Id (Dom t)) RTS.One.the-arr)
```

using *assms arr-char H-arr-char H-dom-char* **by** *fastforce*

lemma *H-cod-char*:
shows $H.cod =$
$(\lambda t.\ if\ H.arr\ t$
$then\ MkArr\ (Cod\ t)\ (Cod\ t)$
$(RTS.Map\ (Id\ (Cod\ t))\ RTS.One.the\text{-}arr)$
$else\ V.null)$
using *H-codomains-char H.cod-in-codomains H.has-codomain-iff-arr H.cod-def H-null-char*
by *auto*

lemma *H-cod-simp*:
assumes $V.arr\ t$
shows $H.cod\ t = MkArr\ (Cod\ t)\ (Cod\ t)$
$(RTS.Map\ (Id\ (Cod\ t))\ RTS.One.the\text{-}arr)$
using *assms arr-char H-arr-char H-cod-char* **by** *fastforce*

lemma *con-implies-H-par*:
assumes $V.con\ t\ u$
shows $H.par\ t\ u$
using *assms con-char V.con-implies-arr(1−2) H-arr-char H-dom-simp H-cod-simp*
by (*simp add*: *arr-char con-implies-Par(1−2)*)

lemma *H-par-resid*:
assumes $V.con\ t\ u$
shows $H.par\ t\ (resid\ t\ u)$
using *assms con-char V.con-implies-arr(1−2) H-arr-char H-dom-simp H-cod-simp Dom-resid Cod-resid arr-char V.arr-resid*
by (*intro conjI*) *metis+*

lemma *simulation-dom*:
shows *simulation resid resid H.dom*
using *H-dom-char arr-char H-arr-char con-char*
apply *unfold-locales*
apply *auto*[*1*]
apply (*metis* (*no-types*, *lifting*) *Con-def H.arr-dom V.arrE*)
by (*metis* (*no-types*, *lifting*) *H.ide-dom H-ide-is-V-ide H-par-resid V.ideE con-implies-H-par*)

lemma *simulation-cod*:
shows *simulation resid resid H.cod*
using *H-cod-char arr-char H-arr-char con-char*
apply *unfold-locales*
apply *presburger*
apply (*metis* (*no-types*, *lifting*) *H.arr-cod V.arr-def rts-category-of-enriched-category.ConE*

```
      rts-category-of-enriched-category-axioms)
  by (metis (no-types, lifting) H.ide-cod H-ide-is-V-ide
      H-par-resid V.con-implies-arr(2) V.ideE con-implies-Par(2))

sublocale dom: simulation resid resid H.dom
  using simulation-dom by blast
sublocale cod: simulation resid resid H.cod
  using simulation-cod by blast
sublocale RR: fibered-product-rts resid resid resid H.dom H.cod ..

sublocale H: simulation RR.resid resid
             ‹λt. if RR.arr t then fst t ⋆ snd t else V.null›
proof
  let ?C = λt. if RR.arr t then fst t ⋆ snd t else V.null
  show ⋀t. ¬ RR.arr t ⟹ ?C t = V.null
    by simp
  fix t u
  assume tu: RR.con t u
  have arr-t: RR.arr t
    using tu RR.con-implies-arr by blast
  have arr-u: RR.arr u
    using tu RR.con-implies-arr by blast
  have t: V.arr (fst t) ∧ V.arr (snd t) ∧ Dom (fst t) = Cod (snd t)
    by (metis H-cod-simp H-dom-simp RR.arr-char arr.inject arr-t)
  have u: V.arr (fst u) ∧ V.arr (snd u) ∧ Dom (fst u) = Cod (snd u)
    using H-cod-simp H-dom-simp RR.arr-char arr-u by auto
  let ?a = Dom (snd t) and ?b = Cod (snd t) and ?c = Cod (fst t)
  have a: ?a ∈ Obj and b: ?b ∈ Obj and c: ?c ∈ Obj
    using tu RR.con-char RR.con-implies-arr ide-char arr-char by fast+
  interpret AB: hom-rts arr-type Obj Hom Id Comp ?a ?b
    using t u ide-char arr-char
    by unfold-locales auto
  interpret BC: hom-rts arr-type Obj Hom Id Comp ?b ?c
    using t u ide-char arr-char
    by unfold-locales auto
  interpret AC: hom-rts arr-type Obj Hom Id Comp ?a ?c
    using t u ide-char arr-char
    by unfold-locales auto
  interpret BCxAB: product-rts ‹HOM_EC ?b ?c› ‹HOM_EC ?a ?b› ..
  interpret bcxab: extensional-rts
                   ‹RTS.Rts (RTS.dom (Hom ?b ?c ⊗ Hom ?a ?b))›
    using t u ide-char arr-char by auto
  have 1: Dom (snd u) = ?a
    using tu RR.con-char RR.arr-char RR.con-implies-arr RR.con-sym
          H-dom-simp H-cod-simp
    by (meson con-implies-Par(1))
  have 2: Cod (fst u) = ?c
    using tu RR.con-char RR.arr-char RR.con-implies-arr RR.con-sym
          H-dom-simp H-cod-simp
```

```
  by (meson con-implies-Par(2))
have 3: Cod (snd u) = ?b
  using tu RR.con-char RR.arr-char RR.con-implies-arr RR.con-sym
        H-dom-simp H-cod-simp
  by (meson con-implies-Par(2))
have 4: BCxAB.con (Trn (fst t), Trn (snd t)) (Trn (fst u), Trn (snd u))
  using tu RR.con-char BCxAB.con-char con-char t by auto

interpret P: simulation BCxAB.resid
             ‹RTS.Rts (RTS.dom (Hom ?b ?c ⊗ Hom ?a ?b))›
             ‹RTS.Pack (Hom ?b ?c) (Hom ?a ?b)›
  using a b c RTS.simulation-Pack by auto
have 5: bcxab.con
          (RTS.Pack (Hom ?b ?c) (Hom ?a ?b) (Trn (fst t), Trn (snd t)))
          (RTS.Pack (Hom ?b ?c) (Hom ?a ?b) (Trn (fst u), Trn (snd u)))
  using 4 P.preserves-con by simp
interpret Comp: simulation
                  ‹RTS.Rts (RTS.dom  (Hom ?b ?c ⊗ Hom ?a ?b))›
                  ‹HOM_EC ?a ?c›
                  ‹RTS.Map (Comp ?a ?b ?c)›
  using a b c Comp-in-hom arr-char
  by (metis (no-types, lifting) RTS.arrD(3) RTS.ideD(2) RTS.ide-prod
      RTS.in-homE ide-Hom prod.sel(1−2))
show con: ?C t ⌢ ?C u
proof −
  have AC.con (RTS.Map (Comp ?a ?b ?c)
                 (RTS.Pack (Hom ?b ?c) (Hom ?a ?b)
                   (Trn (fst t), Trn (snd t))))
               (RTS.Map (Comp ?a ?b ?c)
                 (RTS.Pack (Hom ?b ?c) (Hom ?a ?b)
                  (Trn (fst u), Trn (snd u))))
    using 5 Comp.preserves-con by blast
  thus ?thesis
    using 1 2 3 AC.con-implies-arr(1−2) Con-def H-composable-char
          a arr-t arr-u c con-char null-char t u
    by auto
qed
show ?C (RR.resid t u) = resid (?C t) (?C u)
proof −
  have resid (?C t) (?C u) =
       MkArr ?a ?c
         (RTS.Map (Comp ?a ?b ?c)
           (RTS.Rts (RTS.dom (Hom ?b ?c ⊗ Hom ?a ?b))
             (RTS.Pack (Hom ?b ?c) (Hom ?a ?b) (Trn (fst t), Trn (snd t)))
            (RTS.Pack (Hom ?b ?c) (Hom ?a ?b) (Trn (fst u), Trn (snd u)))))
    using a b c t u arr-t arr-u 1 2 3 4 5 hcomp-def P.preserves-reflects-arr
          arr-char Comp.preserves-resid con con-char by force
  also have ... = ?C (RR.resid t u)
    using tu t u a b c 1 2 3 4 RR.con-char RR.arr-resid hcomp-def
```

```
        RR.resid-def BCxAB.resid-def P.preserves-resid
    by auto
  finally show ?thesis by simp
 qed
qed

lemma simulation-hcomp:
shows simulation RR.resid resid
        (λt. if RR.arr t then fst t ⋆ snd t else V.null)
 ..

lemma Dom-src [simp]:
assumes V.arr t
shows Dom (V.src t) = Dom t
 using assms con-implies-Par(1) by simp

lemma Dom-trg [simp]:
assumes V.arr t
shows Dom (V.trg t) = Dom t
 using assms V.trg-def by simp

lemma Cod-src [simp]:
assumes V.arr t
shows Cod (V.src t) = Cod t
 using assms con-implies-Par(2) by simp

lemma Cod-trg [simp]:
assumes V.arr t
shows Cod (V.trg t) = Cod t
 using assms V.trg-def by simp

lemma null-coincidence [simp]:
shows H.null = V.null
 using H-null-char by blast

lemma arr-coincidence [simp]:
shows H.arr = V.arr
 using H-arr-char arr-char by blast

lemma dom-src [simp]:
shows H.dom (V.src t) = H.dom t
 using H-dom-char H-arr-char arr-char null-char V.arr-src-iff-arr
 by auto

lemma src-dom [simp]:
shows V.src (H.dom t) = H.dom t
 using H-ide-is-V-ide V.src-def V.src-ide dom.extensional by auto

lemma small-homs:
```

```
shows small (H.hom a b)
proof −
  have ¬ H.ide a ∨ ¬ H.ide b ⟹ ?thesis
  proof −
    assume 1: ¬ H.ide a ∨ ¬ H.ide b
    have H.hom a b = {}
      using 1 H.ide-dom H.ide-cod by blast
    thus ?thesis by auto
  qed
  moreover have ⟦H.ide a; H.ide b⟧ ⟹ ?thesis
  proof −
    assume a: H.ide a and b: H.ide b
    interpret Hom: hom-rts arr-type Obj Hom Id Comp ‹Dom a› ‹Dom b›
      using a b
      by (meson H.ideD(1) H-arr-char hom-rts.intro hom-rts-axioms.intro
          rts-enriched-category-axioms)
    have bij-betw Trn (H.hom a b) (Collect Hom.arr)
    proof (intro bij-betwI)
      show Trn ∈ H.hom a b → Collect Hom.arr
        using a b H-ide-char H-arr-char ide-char arr-char
              H-ide-is-V-ide
        by (auto simp add: H-dom-simp H-cod-simp)
      show (λt. MkArr (Dom a) (Dom b) t) ∈ Collect Hom.arr → H.hom a b
        using a b H.ideD(1−2) H-cod-simp H-dom-char Hom.a Hom.b
              arr-MkArr arr-coincidence
        by auto
      show ⋀x. x ∈ H.hom a b ⟹ MkArr (Dom a) (Dom b) (Trn x) = x
        by (metis CollectD H.ide-in-hom H.seqI' H-seq-char MkArr-Trn a b)
     show ⋀y. y ∈ Collect Hom.arr ⟹ Trn (MkArr (Dom a) (Dom b) y) = y
        using a b by auto
    qed
    hence inj-on Trn (H.hom a b) ∧ Collect Hom.arr = Trn ‘ H.hom a b
      using bij-betw-imp-inj-on bij-betw-imp-surj-on by metis
    thus ?thesis
      using Hom.small small-image-iff by auto
  qed
  ultimately show ?thesis by blast
qed
```

Note that the arrow type of the RTS-category given by the following is $('O, 'A)$ *arr*, where 'A is the type of the universe underlying the category RTS and 'O is the type of objects of the context RTS-enriched category. If we start with an RTS-enriched category having object type 'O, then we construct an RTS-category having arrow type $('O, 'A)$ *arr*, and then we try to go back to an RTS-enriched category, the hom-RTS's will have arrow type $('O, 'A)$ *arr*, not $'A$ as required for them to determine objects of *RTS*. So to show that the passage between RTS-categories and RTS-enriched categories is an equivalence, we will need to be able to reduce the type of the hom-RTS's from $('O, 'A)$ *arr* back to $'A$.

sublocale *rts-category resid hcomp*
using *null-coincidence arr-coincidence small-homs*
by *unfold-locales auto*

proposition *is-rts-category*:
shows *rts-category resid hcomp*
..

end

5.2.1 The Small Case

Given an RTS-enriched category, the corresponding RTS-category R has arrows at a higher type than the arrow type $'A$ of the base category *RTS*. In particular, the arrow type for this category is $('O, 'A)$ *arr*, where $'O$ is the element type of *Obj*. If we want to reconstruct the original RTS-enriched category up to isomorphism, then we need to be able to map this type back down to $'A$, so that we can obtain (via *RTS.MkIde*) an RTS R' with arrow type $'A$, which is isomorphic to the desired RTS-category R. For this to be possible, clearly we need the set *Obj* to be small. However, we also need a way to represent each element of *Obj* uniquely as an element of $'A$. This would be true automatically if we knew that $'A$ were large enough to embed all small sets, but we don't want to tie the definition of the category *RTS* itself to a particular definition of "small". So, here we instead just directly assume the existence of an injection from *Obj* to $'A$.

locale *rts-category-of-small-enriched-category* =
rts-category-of-enriched-category arr-type Obj Hom Id Comp
for *arr-type* :: $'A$ *itself*
and *Obj* :: $'O$ *set*
and *Hom* :: $'O \Rightarrow 'O \Rightarrow 'A$ *rtscatx.arr*
and *Id* :: $'O \Rightarrow 'A$ *rtscatx.arr*
and *Comp* :: $'O \Rightarrow 'O \Rightarrow 'O \Rightarrow 'A$ *rtscatx.arr* +
assumes *small-Obj*: *small Obj*
and *inj-Obj-to-arr*: $\exists \varphi :: 'O \Rightarrow 'A.$ *inj-on* φ *Obj*
begin

We will use R to refer to the RTS constructed from the given enriched category.

abbreviation $R :: ('O, 'A)$ *arr resid*
where $R \equiv$ *resid*

The locale assumptions are sufficient to allow us to uniquely encode each element of *Collect arr* $\cup$ $\{null\}$ as single element of $'A$.

lemma *ex-arrow-injection*:
shows $\exists i :: ('O, 'A)$ *arr* $\Rightarrow 'A.$ *inj-on i* (*Collect arr* $\cup$ $\{null\}$)
proof −
obtain $\varphi :: 'O \Rightarrow 'A$ **where** φ: *inj-on* φ *Obj*

```
    using inj-Obj-to-arr by blast
  let ?p = λt. some-pair (some-pair (φ (Dom t), φ (Cod t)), Trn t)
  have p: inj-on ?p (Collect arr)
    by (metis (mono-tags, lifting) CollectD φ arr-char arr-eqI
        first-conv inj-onD inj-onI null-char second-conv)
  let ?i = λx. some-lift (if arr x then Some (?p x) else None)
  have inj-on ?i (Collect arr ∪ {null})
  proof
    fix x y
    assume x: x ∈ Collect arr ∪ {null} and y: y ∈ Collect arr ∪ {null}
    assume eq: ?i x = ?i y
    show x = y
      using x y eq p inj-some-lift injD inj-on-contraD by fastforce
  qed
  thus ?thesis by auto
qed

lemma bij-betw-Obj-horiz-ide:
shows bij-betw mkobj Obj (Collect H.ide)
  using arr-char Id-yields-horiz-ide H-ide-char horizontal-unit-def
  apply (intro bij-betwI)
    apply auto[3]
  by (metis Dom.simps(1) mem-Collect-eq)

lemma ex-isomorphic-image-rts:
shows ∃R' (UP :: 'A ⇒ ('O, 'A) arr) (DN :: ('O, 'A) arr ⇒ 'A).
        small-rts R' ∧ extensional-rts R' ∧ inverse-simulations R R' UP DN
proof −
  obtain i :: ('O, 'A) arr ⇒ 'A where i: inj-on i (Collect arr ∪ {null})
    using ex-arrow-injection by blast
  interpret R': inj-image-rts i R
    using i by unfold-locales
  interpret R': extensional-rts R'.resid
    using V.extensional-rts-axioms R'.preserves-extensional-rts by blast
  interpret R': small-rts R'.resid
  proof −
    have small (Collect arr)
    proof −
      have small ((Collect H.ide × Collect H.ide) ×
                  (⋃x∈Collect H.ide × Collect H.ide.
                      H.hom (fst x) (snd x)))
      proof −
        have ⋀a b. ⟦H.ide a; H.ide b⟧ ⟹ small (H.hom a b)
          using small-homs by auto
        moreover have small (Collect H.ide × Collect H.ide)
          using small-Obj bij-betw-Obj-horiz-ide
          by (metis (no-types, lifting) bij-betw-imp-surj-on replacement
              small-Sigma)
        ultimately show ?thesis
```

```
      using small-homs by force
    qed
    moreover
    have (λt. ((H.dom t, H.cod t), t)) ∈
                  Collect arr →
                    ((Collect H.ide × Collect H.ide) ×
                      (⋃x∈Collect H.ide × Collect H.ide.
                          H.hom (fst x) (snd x)))
    proof
      fix t
      assume t: t ∈ Collect arr
      have H.dom t ∈ Collect H.ide ∧ H.cod t ∈ Collect H.ide
        using t arr-coincidence H.ide-dom H.ide-cod by simp
      moreover have t ∈ H.hom (H.dom t) (H.cod t)
        using t arr-coincidence by auto
      ultimately
      show ((H.dom t, H.cod t), t) ∈
             (Collect H.ide × Collect H.ide) ×
               (⋃x∈Collect H.ide × Collect H.ide. H.hom (fst x) (snd x))
        by auto
    qed
    moreover have inj-on (λt. ((H.dom t, H.cod t), t)) (Collect arr)
      by (intro inj-onI) blast
    ultimately show ?thesis
      using small-image-iff
            smaller-than-small
              [of - (λt. ((H.dom t, H.cod t), t)) ‘ Collect arr]
      by blast
  qed
  hence small-rts R
    using small-rts-def rts-axioms small-rts-axioms.intro by auto
  thus small-rts R'.resid
    using R'.preserves-reflects-small-rts by blast
qed
have inverse-simulations R'.resid R R'.map_ext R'.map'_ext
  using R'.inverse-simulations-axioms by auto
thus ?thesis
  using R'.rts-axioms R'.extensional-rts-axioms R'.small-rts-axioms
        inverse-simulations-sym
  by meson
qed
```

We now choose some RTS with the properties asserted by the previous lemma, along with the invertible simulations that relate it to R.

```
definition R' :: 'A resid
where R' ≡ SOME R'. ∃ UP DN. small-rts R' ∧ extensional-rts R' ∧
                             inverse-simulations resid R' UP DN

definition UP :: 'A ⇒ ('O, 'A) arr
```

where $UP \equiv SOME\ UP.\ \exists\, DN.\ small\text{-}rts\ R' \wedge extensional\text{-}rts\ R' \wedge$
$inverse\text{-}simulations\ resid\ R'\ UP\ DN$

definition $DN :: ('O, 'A)\ arr \Rightarrow 'A$
where $DN \equiv SOME\ DN.\ small\text{-}rts\ R' \wedge extensional\text{-}rts\ R' \wedge$
$inverse\text{-}simulations\ resid\ R'\ UP\ DN$

lemma *R′-prop*:
shows $\exists\, UP\ DN.\ small\text{-}rts\ R' \wedge extensional\text{-}rts\ R' \wedge$
$inverse\text{-}simulations\ R\ R'\ UP\ DN$
unfolding *R′-def*
using *small-Obj ex-isomorphic-image-rts*
someI-ex
$[of\ \lambda R'.\ \exists\, UP\ DN.\ small\text{-}rts\ R' \wedge extensional\text{-}rts\ R' \wedge$
$inverse\text{-}simulations\ R\ R'\ UP\ DN]$
by *auto*

sublocale *R′*: *extensional-rts R′*
using *R′-prop* **by** *simp*
sublocale *R′*: *small-rts R′*
using *R′-prop* **by** *simp*

lemma *extensional-rts-R′*:
shows *extensional-rts R′*
..

lemma *small-rts-R′*:
shows *small-rts R′*
..

sublocale *UP-DN*: *inverse-simulations R R′ UP DN*
using *small-Obj R′-prop UP-def DN-def*
someI-ex $[of\ \lambda UP.\ \exists\, DN.\ inverse\text{-}simulations\ resid\ R'\ UP\ DN]$
someI-ex $[of\ \lambda DN.\ inverse\text{-}simulations\ resid\ R'\ UP\ DN]$
by *auto*

lemma *inverse-simulations-UP-DN*:
shows *inverse-simulations resid R′ UP DN*
..

lemma *R′-src-char*:
shows $R'.src = DN \circ src \circ UP$
proof −
have $\bigwedge t.\ DN\ (UP\ (R'.src\ t)) = DN\ (src\ (UP\ t))$
by (*metis H.dom-null R′.con-arr-src*(*2*) *R′.ide-src R′.not-arr-null R′.src-def*
UP-DN.F.extensional UP-DN.F.preserves-con UP-DN.F.preserves-ide
null-coincidence src-dom V.src-eqI)
moreover have $\bigwedge t.\ DN\ (UP\ (R'.src\ t)) = R'.src\ t$
using *R′.arr-src-iff-arr R′.src-def UP-DN.inv*

```
    by (metis (no-types, lifting) comp-apply)
  ultimately show ?thesis by auto
qed

lemma R'-trg-char:
shows R'.trg = DN ∘ trg ∘ UP
proof −
  have ⋀t. DN (UP (R'.trg t)) = DN (trg (UP t))
    by (metis R'.arr-trg-iff-arr UP-DN.F.extensional UP-DN.F.preserves-trg
        V.null-is-zero(2) V.trg-def)
  moreover have ⋀t. DN (UP (R'.trg t)) = R'.trg t
    using R'.arr-trg-iff-arr R'.trg-def UP-DN.inv
    by (metis (no-types, lifting) R'.src-def R'.src-trg comp-apply)
  ultimately show ?thesis by auto
qed
```

We transport the horizontal composition ($\star$) to R' via the isomorphisms UP and DN.

```
abbreviation hcomp' :: 'A resid  (infixr ⋆´ 53)
where t ⋆´ u ≡ DN (UP t ⋆ UP u)

interpretation H': Category.partial-magma hcomp'
  by unfold-locales
     (metis H-composable-char R'.not-arr-null UP-DN.F.extensional
      UP-DN.F.preserves-reflects-arr UP-DN.G.extensional)

lemma H'-null-char:
shows H'.null = DN null
  using arr-coincidence
  by (metis H'.null-is-zero(2) R'.not-arr-null UP-DN.F.extensional
      hcomp-Null(2) null-char)

interpretation H': partial-composition ‹λt u. DN (hcomp (UP t) (UP u))› ..

lemma H'-ide-char:
shows H'.ide t ⟷ H.ide (UP t)
proof
  have 1: ⋀f. ⟦arr f; Dom f = Cod (UP t); t ⋆´ t ≠ DN null;
               ⋀t u. (t ⋆ u ≠ null) = (arr t ∧ arr u ∧ Dom t = Cod u);
               ∀f. (f ⋆´ t ≠ DN null ⟶ f ⋆´ t = f) ∧
                   (t ⋆´ f ≠ DN null ⟶ t ⋆´ f = f)⟧
                 ⟹ f ⋆ UP t = f
    by (metis (no-types, lifting) UP-DN.G.preserves-reflects-arr
        UP-DN.inv'-simp arr-hcomp)
  have 2: ⋀f. ⟦arr f; Dom (UP t) = Cod f; t ⋆´ t ≠ DN null;
               ⋀t u. (t ⋆ u ≠ null) = (arr t ∧ arr u ∧ Dom t = Cod u);
               ∀f. (f ⋆´ t ≠ DN null ⟶ f ⋆´ t = f) ∧
                   (t ⋆´ f ≠ DN null ⟶ t ⋆´ f = f)⟧
                 ⟹ UP t ⋆ f = f
```

```
    by (metis (no-types, lifting) UP-DN.G.preserves-reflects-arr
        UP-DN.inv'-simp arr-hcomp)
  show H'.ide t ⟹ obj (UP t)
    unfolding H'.ide-def H.ide-def
    using H'-null-char H-composable-char
    apply auto[1]
     apply (metis UP-DN.F.preserves-reflects-arr)
     apply metis
    using 1 2 by blast+
  show obj (UP t) ⟹ H'.ide t
    unfolding H'.ide-def H.ide-def
    apply (auto simp add: H'-null-char H-composable-char)[1]
     apply (metis H-composable-char UP-DN.F.extensional UP-DN.inv-simp)
    by (metis H-composable-char UP-DN.F.extensional UP-DN.inv-simp)
qed

lemma H'-domains-char:
shows H'.domains t = DN ‘ H.domains (UP t)
proof −
  have {a. H.ide (UP a) ∧ t ⋆´ a ≠ DN null} =
       DN ‘ {a. H.ide a ∧ UP t ⋆ a ≠ null}
  proof
    show {a. H.ide (UP a) ∧ t ⋆´ a ≠ DN null} ⊆
         DN ‘ {a. H.ide a ∧ UP t ⋆ a ≠ null}
    proof
      fix a
      assume a: a ∈ {a. H.ide (UP a) ∧ t ⋆´ a ≠ DN null}
      have 1: H.ide (UP a) ∧ UP t ⋆ UP a ≠ null
        using a by auto
      moreover have a = DN (UP a)
        using a 1
        by (metis (no-types, opaque-lifting) H-composable-char
            UP-DN.F.preserves-reflects-arr UP-DN.inv comp-apply)
      ultimately show a ∈ DN ‘ {a. H.ide a ∧ UP t ⋆ a ≠ null} by blast
    qed
    show DN ‘ {a. H.ide a ∧ UP t ⋆ a ≠ null} ⊆
         {a. H.ide (UP a) ∧ t ⋆´ a ≠ DN null}
    proof
      fix a
      assume a: a ∈ DN ‘ {a. H.ide a ∧ UP t ⋆ a ≠ null}
      obtain UPa
      where UPa: a = DN UPa ∧ UPa ∈ {a. H.ide a ∧ UP t ⋆ a ≠ null}
        using a by blast
      have UPa = UP a
        using UPa H-composable-char UP-DN.inv' comp-apply by auto
      thus a ∈ {a. H.ide (UP a) ∧ DN (UP t ⋆ UP a) ≠ DN null}
        using UPa null-coincidence
        by (metis (mono-tags, lifting) H.ext UP-DN.G.preserves-reflects-arr
            arr-coincidence mem-Collect-eq V.not-arr-null)
```

```
    qed
  qed
  thus ?thesis
    unfolding H'.domains-def H.domains-def
    using H'-ide-char H'-null-char null-coincidence by simp
qed

lemma H'-codomains-char:
shows H'.codomains t = DN ‘ H.codomains (UP t)
proof −
  have {b. H.ide (UP b) ∧ b ⋆´ t ≠ DN null} =
        DN ‘ {b. H.ide b ∧ b ⋆ UP t ≠ null}
  proof
    show {b. H.ide (UP b) ∧ b ⋆´ t ≠ DN null} ⊆
          DN ‘ {b. H.ide b ∧ b ⋆ UP t ≠ null}
    proof
      fix b
      assume b: b ∈ {b. H.ide (UP b) ∧ b ⋆´ t ≠ DN null}
      have DN (UP b) ∈ DN ‘ {b. H.ide b ∧ b ⋆ UP t ≠ null}
        using b by auto
      moreover have DN (UP b) = b
        using b
        by (metis (no-types, lifting) H'.ide-def H'-ide-char
            H.comp-ide-self mem-Collect-eq)
      ultimately show b ∈ DN ‘ {b. H.ide b ∧ b ⋆ UP t ≠ null} by auto
    qed
    show DN ‘ {b. H.ide b ∧ b ⋆ UP t ≠ null} ⊆
          {b. H.ide (UP b) ∧ b ⋆´ t ≠ DN null}
    proof
      fix b
      assume b: b ∈ DN ‘ {b. H.ide b ∧ b ⋆ UP t ≠ null}
      obtain UPb
      where UPb: b = DN UPb ∧ UPb ∈ {b. H.ide b ∧ b ⋆ UP t ≠ null}
        using b by blast
      have UPb = UP b
        using UPb H-composable-char UP-DN.inv' comp-apply by auto
      thus b ∈ {b. H.ide (UP b) ∧ DN (UP b ⋆ UP t) ≠ DN null}
        using UPb null-coincidence arr-coincidence
        by (metis (mono-tags, lifting) H.ext UP-DN.G.preserves-reflects-arr
            mem-Collect-eq V.not-arr-null)
    qed
  qed
  thus ?thesis
    unfolding H'.codomains-def H.codomains-def
    using H'-ide-char H'-null-char null-coincidence by simp
qed

lemma H'-arr-char:
shows H'.arr t = H.arr (UP t)
```

```
  unfolding H'.arr-def H.arr-def
  using H'-domains-char H'-codomains-char by auto

lemma H'-seq-char:
shows H'.seq t u ⟷ H.seq (UP t) (UP u)
  by (simp add: H'-arr-char)

sublocale H': category hcomp'
proof
  show ⋀g f. g ⋆' f ≠ H'.null ⟹ H'.seq g f
    using H'-null-char H'-seq-char UP-DN.G.extensional by auto
  show ⋀f. (H'.domains f ≠ {}) = (H'.codomains f ≠ {})
   using H'-domains-char H'-codomains-char H.has-domain-iff-has-codomain
    by simp
  show ⋀h g f. ⟦H'.seq h g; H'.seq (DN (UP h ⋆ UP g)) f⟧ ⟹ H'.seq g f
    by (metis H'-seq-char H.match-1 UP-DN.inv'-simp arr-coincidence)
  show ⋀h g f. ⟦H'.seq h (g ⋆' f); H'.seq g f⟧ ⟹ H'.seq h g
    using H'-seq-char H-seq-char by auto
  show ⋀g f h. ⟦H'.seq g f; H'.seq h g⟧ ⟹ H'.seq (h ⋆' g) f
    using H'-arr-char H-seq-char by auto
  show ⋀g f h. ⟦H'.seq g f; H'.seq h g⟧ ⟹ (h ⋆' g) ⋆' f = h ⋆' g ⋆' f
    using H'-seq-char H.comp-assoc UP-DN.inv' by auto
qed

lemma hcomp'-is-category:
shows category hcomp'
  ..

lemma H'-dom-char:
shows H'.dom = DN ∘ H.dom ∘ UP
proof
  fix t
  show H'.dom t = (DN ∘ H.dom ∘ UP) t
  proof (cases arr (UP t))
    show ¬ arr (UP t) ⟹ ?thesis
      by (metis H'.dom-def H'.domains-char H'-arr-char H'-null-char
          H.dom-null H-arr-char UP-DN.F.extensional
          UP-DN.F.preserves-reflects-arr arr-char comp-def
          null-coincidence)
    assume t: arr (UP t)
    have (DN ∘ H.dom ∘ UP) t = DN (H.dom (UP t))
      using t by auto
    also have ... = H'.dom t
      using t H'-domains-char H'-arr-char arr-coincidence H.dom-in-domains
            H.has-domain-iff-arr
      by (intro H'.dom-eqI') auto
    finally show ?thesis by auto
  qed
qed
```

```
lemma H'-cod-char:
shows H'.cod = DN ∘ H.cod ∘ UP
proof
  fix t
  show H'.cod t = (DN ∘ H.cod ∘ UP) t
  proof (cases arr (UP t))
    show ¬ arr (UP t) ⟹ ?thesis
      by (metis H'.cod-def H'.codomains-char H'-arr-char H'-null-char
          H.cod-null H-arr-char UP-DN.F.extensional
          UP-DN.F.preserves-reflects-arr arr-char comp-def
          null-coincidence)
    assume t: arr (UP t)
    have (DN ∘ H.cod ∘ UP) t = DN (H.cod (UP t))
      using t by auto
    also have ... = H'.cod t
      using t H'-codomains-char H'-arr-char arr-coincidence
            H.cod-in-codomains H.has-codomain-iff-arr
      by (intro H'.cod-eqI') auto
    finally show ?thesis by auto
  qed
qed

lemma null'-coincidence [simp]:
shows H'.null = R'.null
  by (simp add: H'-null-char UP-DN.G.extensional)

lemma arr'-coincidence [simp]:
shows H'.arr = R'.arr
  using H'-arr-char UP-DN.F.preserves-reflects-arr arr-coincidence by auto

lemma H'-hom-char:
shows H'.hom a b = DN ‘ H.hom (UP a) (UP b)
proof
  show H'.hom a b ⊆ DN ‘ H.hom (UP a) (UP b)
  proof
    fix t
    assume t: t ∈ H'.hom a b
    have UP t ∈ H.hom (UP a) (UP b)
    proof
      have a: V.ide (UP a)
        using t arr'-coincidence H'-ide-char UP-DN.F.preserves-ide
        by (metis H'.arr-dom H'.dom-dom H'.ide-char' H'.in-homE
            H-ide-is-V-ide mem-Collect-eq)
      have b: V.ide (UP b)
        using t arr'-coincidence H'-ide-char UP-DN.F.preserves-ide
        by (metis H'.arr-cod H'.cod-cod H'.ide-char' H'.in-homE
            H-ide-is-V-ide mem-Collect-eq)
      show H.in-hom (UP t) (UP a) (UP b)
```

```
    proof
      show 1: H.arr (UP t)
        using t arr-coincidence arr'-coincidence
            UP-DN.F.preserves-reflects-arr
        by auto
      show H.dom (UP t) = UP a
      proof −
        have 2: DN (H.dom (UP t)) = a
          using t a 1 H'-dom-char by auto
        also have ... = DN (UP a)
          using t a UP-DN.inv
          by (metis (no-types, lifting) UP-DN.F.preserves-reflects-arr
              comp-apply V.ide-implies-arr)
        finally have DN (H.dom (UP t)) = DN (UP a) by blast
        thus ?thesis
          by (metis 1 2 H.arr-dom-iff-arr UP-DN.inv'
              arr-coincidence comp-apply)
      qed
      show H.cod (UP t) = UP b
      proof −
        have 2: DN (H.cod (UP t)) = b
          using t b 1 arr-coincidence H'-cod-char by auto
        also have ... = DN (UP b)
          using t b UP-DN.inv
          by (metis (no-types, lifting) UP-DN.F.preserves-reflects-arr
              comp-apply V.ide-implies-arr)
        finally have DN (H.cod (UP t)) = DN (UP b) by blast
        thus ?thesis
          by (metis 1 2 H.arr-cod-iff-arr UP-DN.inv'
              arr-coincidence comp-apply)
      qed
    qed
  qed
  moreover have DN (UP t) = t
    using t UP-DN.inv
    by (metis (no-types, lifting) H'.in-homE arr'-coincidence
        comp-apply mem-Collect-eq)
  ultimately show t ∈ DN ‘ H.hom (UP a) (UP b)
    by (simp add: rev-image-eqI)
qed
show DN ‘ H.hom (UP a) (UP b) ⊆ H'.hom a b
proof
  fix t'
  assume t': t' ∈ DN ‘ H.hom (UP a) (UP b)
  obtain t where t: t ∈ H.hom (UP a) (UP b) ∧ t' = DN t
    using t' by blast
  have DN t ∈ H'.hom a b
  proof
    show H'.in-hom (DN t) a b
```

```
      proof
        show H'.arr (DN t)
          using t H'-arr-char arr-coincidence by fastforce
        show H'.dom (DN t) = a
          using t H'-dom-char
          by (metis (no-types, lifting) Fun.comp-def H'.ide-char'
              H'-ide-char H.ide-dom H.in-homE H-arr-char UP-DN.inv
              UP-DN.inv' arr'-coincidence arr-char mem-Collect-eq)
        show H'.cod (DN t) = b
          using t H'-cod-char
          by (metis (no-types, lifting) Fun.comp-def H'.ide-char'
              H'-ide-char H.ide-cod H.in-homE H-arr-char
              UP-DN.inv UP-DN.inv' arr'-coincidence arr-char
              mem-Collect-eq)
      qed
    qed
    thus t' ∈ H'.hom a b
      using t by blast
  qed
qed

interpretation dom': simulation R' R' H'.dom
  using H'-dom-char simulation-comp simulation-dom
        UP-DN.F.simulation-axioms UP-DN.G.simulation-axioms
  by auto

interpretation cod': simulation R' R' H'.cod
  using H'-cod-char simulation-comp simulation-cod
        UP-DN.F.simulation-axioms UP-DN.G.simulation-axioms
  by auto

lemma R'-con-char:
shows R'.con t u ⟷ V.con (UP t) (UP u)
  by (metis UP-DN.F.preserves-con UP-DN.F.preserves-reflects-arr
      UP-DN.G.preserves-con UP-DN.inv comp-apply
      residuation.con-implies-arr(1-2) V.residuation-axioms)

sublocale R'R': fibered-product-rts R' R' R' H'.dom H'.cod ..

sublocale H': simulation R'R'.resid R'
                ‹λt. if R'R'.arr t then fst t ⋆´ snd t else R'.null›
proof
  show ⋀t. ¬ R'R'.arr t ⟹
             (if R'R'.arr t then fst t ⋆´ snd t else R'.null) = R'.null
    by auto
  fix t u
  assume tu: R'R'.con t u
  show 1: R'.con (if R'R'.arr t then fst t ⋆´ snd t else R'.null)
                 (if R'R'.arr u then fst u ⋆´ snd u else R'.null)
```

```
proof −
  have UP (fst t ⋆´ snd t) = UP (fst t) ⋆ UP (snd t) ∧
       UP (fst u ⋆´ snd u) = UP (fst u) ⋆ UP (snd u)
    using tu arr-coincidence null-coincidence UP-DN.inv′ H.ext
    apply auto[1]
    by (metis (no-types, lifting) UP-DN.F.extensional
        UP-DN.G.preserves-reflects-arr UP-DN.inv′-simp)+
  moreover have UP (fst t) ⋆ UP (snd t) ⌢ UP (fst u) ⋆ UP (snd u)
  proof −
    have RR.con (UP (fst t), UP (snd t)) (UP (fst u), UP (snd u))
      by (metis H′.seqI H′-seq-char H.seqE R′R′.arr-char R′R′.con-char
          R′R′.residuation-axioms R′-con-char RR.con-char arr′-coincidence
          fst-conv residuation.con-implies-arr(1−2) snd-conv tu)
    thus ?thesis
      using H.preserves-con RR.con-implies-arr(1−2) by force
  qed
  ultimately show ?thesis
    using tu
    by (simp add: R′R′.con-implies-arr(1) R′R′.con-implies-arr(2))
qed
show (if R′R′.arr (R′R′.resid t u)
      then fst (R′R′.resid t u) ⋆´ snd (R′R′.resid t u)
      else R′.null) =
     R′ (if R′R′.arr t then fst t ⋆´ snd t else R′.null)
       (if R′R′.arr u then fst u ⋆´ snd u else R′.null)
proof −
  have fst (R′R′.resid t u) ⋆´ snd (R′R′.resid t u) =
       R′ (fst t ⋆´ snd t) (fst u ⋆´ snd u)
  proof −
    have UP (fst (R′R′.resid t u) ⋆´ snd (R′R′.resid t u)) =
         UP (R′ (fst t ⋆´ snd t) (fst u ⋆´ snd u))
    proof −
      have UP (fst (R′R′.resid t u) ⋆´ snd (R′R′.resid t u)) =
           UP (R′ (fst t) (fst u) ⋆´ R′ (snd t) (snd u))
        using tu R′R′.con-char R′R′.resid-def by auto
      also have ... = UP (R′ (fst t) (fst u)) ⋆ UP (R′ (snd t) (snd u))
        using tu
        by (metis H.ext UP-DN.inv′ arr-coincidence comp-apply
            null-coincidence)
      also have ... = resid (UP (fst t)) (UP (fst u)) ⋆
                      resid (UP (snd t)) (UP (snd u))
        using R′R′.con-char UP-DN.F.preserves-resid tu by presburger
      also have ... = (UP (fst t) ⋆ UP (snd t)) \ (UP (fst u) ⋆ UP (snd u))
        using tu 1 H.preserves-resid H.seqE R′.con-implies-arr(1−2)
              R′.not-arr-null R′R′.con-char R′-con-char RR.arr-char
              RR.arr-resid-iff-con RR.con-char RR.resid-def
              UP-DN.G.extensional
        by (metis (no-types, lifting) H′.seqI H′-arr-char H′-seq-char
            hpar-arr-resid resid-hcomp(2))
```

```
        also have ... = UP (fst t ⋆´ snd t) \ UP (fst u ⋆´ snd u)
        proof −
          have UP (fst t ⋆´ snd t) = UP (fst t) ⋆ UP (snd t) ∧
               UP (fst u ⋆´ snd u) = UP (fst u) ⋆ UP (snd u)
            using H'.seqI R'R'.arr-char R'R'.con-implies-arr(1−2) tu by auto
          thus ?thesis
            using tu H.ext UP-DN.inv' arr-coincidence comp-apply
                  null-coincidence
            by auto
        qed
        also have ... = UP (R' (fst t ⋆´ snd t) (fst u ⋆´ snd u))
          using 1 R'R'.con-implies-arr(1) R'R'.con-implies-arr(2) tu by auto
        finally show ?thesis by blast
      qed
      thus ?thesis
        by (metis (no-types, lifting) H'.seqI R'R'.arr-char R'R'.arr-resid
          UP-DN.F.preserves-reflects-arr UP-DN.inv-simp arr'-coincidence tu)
    qed
    thus ?thesis
      using tu 1 H.preserves-resid by auto
  qed
qed

proposition is-locally-small-rts-category:
shows locally-small-rts-category R' hcomp'
proof
  show H'.null = R'.null
    by (simp add: H'-null-char UP-DN.G.extensional)
  show H'.arr = R'.arr
   using H'-arr-char UP-DN.F.preserves-reflects-arr arr-coincidence by auto
  show ⋀t. R'.src (H'.dom t) = H'.dom t
    using R'-src-char H'-dom-char R'.arr-src-iff-arr UP-DN.G.extensional
          UP-DN.G.preserves-reflects-arr
    apply auto[1]
    by (metis (no-types, lifting) R'.not-arr-null UP-DN.inv'-simp src-dom)
  show ⋀a b. small (H'.hom a b)
    using H'-hom-char small-homs by simp
qed

end
```

5.2.2 Functoriality

```
locale rts-functor-of-enriched-functor =
  universe arr-type +
  RTS: rtscat arr-type +
  A: rts-enriched-category arr-type Obj_A Hom_A Id_A Comp_A +
  B: rts-enriched-category arr-type Obj_B Hom_B Id_B Comp_B +
  EF: rts-enriched-functor
```

```
      Obj_A Hom_A Id_A Comp_A Obj_B Hom_B Id_B Comp_B F_o F_a
for Obj_A :: 'a set
and Hom_A :: 'a ⇒ 'a ⇒ 'A rtscatx.arr
and Id_A :: 'a ⇒ 'A rtscatx.arr
and Comp_A :: 'a ⇒ 'a ⇒ 'a ⇒ 'A rtscatx.arr
and Obj_B :: 'b set
and Hom_B :: 'b ⇒ 'b ⇒ 'A rtscatx.arr
and Id_B :: 'b ⇒ 'A rtscatx.arr
and Comp_B :: 'b ⇒ 'b ⇒ 'b ⇒ 'A rtscatx.arr
and F_o :: 'a ⇒ 'b
and F_a :: 'a ⇒ 'a ⇒ 'A rtscatx.arr
begin

  interpretation A: rts-category-of-enriched-category
                  arr-type Obj_A Hom_A Id_A Comp_A
    ..
  interpretation B: rts-category-of-enriched-category
                  arr-type Obj_B Hom_B Id_B Comp_B
    ..

  definition F
  where F t ≡ if residuation.arr A.resid t
              then B.MkArr (F_o (A.Dom t)) (F_o (A.Cod t))
                           (RTS.Map (F_a (A.Dom t) (A.Cod t)) (A.Trn t))
              else ResiduatedTransitionSystem.partial-magma.null B.resid

  lemma preserves-arr:
  assumes A.H.arr f
  shows B.H.arr (F f)
  proof −
    let ?a = A.Dom f
    let ?b = A.Cod f
    show 1: B.H.arr (F f)
    proof −
      have B.arr (F f)
        unfolding F-def
        using assms A.arr-char B.arr-MkArr A.arr-coincidence
          B.arr-coincidence
        apply (simp, intro B.arr-MkArr)
          apply blast
         apply blast
        using EF.is-local-simulation simulation.preserves-reflects-arr
        by metis
      thus ?thesis by auto
    qed
  qed

  sublocale rts-functor A.resid A.hcomp B.resid B.hcomp F
  proof
```

```
show ⋀f. ¬ A.H.arr f ⟹ F f = B.H.null
  using F-def A.arr-coincidence B.null-coincidence by simp
show 1: ⋀f. A.H.arr f ⟹ B.H.arr (F f)
  using preserves-arr by simp
fix f
assume f: A.H.arr f
have 0: A.arr (A.MkArr (A.Dom f) (A.Dom f)
                (RTS.Map (Id_A (A.Dom f)) RTS.One.the-arr))
  using f A.arr-char A.arr-coincidence A.Id-in-hom RTS.Map-ide
  by (metis (no-types, lifting) A.H-dom-char A.dom.preserves-reflects-arr)
have 1: B.arr (B.MkArr (F_o (A.Dom f)) (F_o (A.Cod f))
                (RTS.Map (F_a (A.Dom f) (A.Cod f)) (A.Trn f)))
  using f 1 F-def B.H-dom-char B.arr-char B.null-char B.arr-coincidence
        B.null-coincidence
  by (intro B.arr-MkArr) auto
have 2: A.arr (A.MkArr (A.Cod f) (A.Cod f)
                (RTS.Map (Id_A (A.Cod f)) RTS.One.the-arr))
  using f A.arr-char A.arr-coincidence A.H.ideD(1)
        A.Id-yields-horiz-ide
  by force
show B.H.dom (F f) = F (A.H.dom f)
proof (intro B.arr-eqI)
  show B.H.dom (F f) ≠ B.null
    using f 1 F-def B.H-dom-char B.null-char by auto
  show F (A.H.dom f) ≠ B.null
    using f 0 F-def A.H-dom-char B.null-char by auto
  show B.Dom (B.H.dom (F f)) = B.Dom (F (A.H.dom f))
    using f 0 1 F-def A.H-dom-char B.H-dom-char by simp
  show B.Cod (B.H.dom (F f)) = B.Cod (F (A.H.dom f))
    using f 0 1 F-def A.H-dom-char B.H-dom-char by simp
  show B.Trn (B.H.dom (F f)) = B.Trn (F (A.H.dom f))
  proof −
    have B.Trn (F (A.H.dom f)) =
          RTS.Map (F_a (A.Dom f) (A.Dom f))
            (RTS.Map (Id_A (A.Dom f)) RTS.One.the-arr)
      using f 0 F-def A.H-dom-char B.H-dom-char EF.preserves-Id
            RTS.Map-comp
      by auto
    also have ... = RTS.Map (F_a (A.Dom f) (A.Dom f) · Id_A (A.Dom f))
                      RTS.One.the-arr
      using f RTS.Map-comp A.Id-in-hom
            EF.preserves-Hom [of A.Dom f A.Dom f]
            EF.preserves-Obj [of A.Dom f] comp-apply
      apply auto[1]
      using A.arr-char [of f] by fastforce
    also have ... = RTS.Map (Id_B (F_o (A.Dom f))) RTS.One.the-arr
      using f 0 1 EF.preserves-Id A.arr-char by simp
    also have ... = B.Trn (B.H.dom (F f))
      using f 0 1 F-def by (simp add: B.H-dom-simp B.H-cod-simp)
```

```
    finally show ?thesis by simp
  qed
qed
show B.H.cod (F f) = F (A.H.cod f)
proof (intro B.arr-eqI)
  show B.H.cod (F f) ≠ B.null
    using f 1 F-def B.H-cod-char B.null-char by auto
  show F (A.H.cod f) ≠ B.null
    using f 2 F-def A.H-cod-char B.null-char by auto
  show B.Dom (B.H.cod (F f)) = B.Dom (F (A.H.cod f))
    using f 2 F-def A.H-cod-char B.H-cod-char B.null-char
          B.cod.extensional ‹B.H.cod (F f) ≠ B.null›
    by fastforce
  show B.Cod (B.H.cod (F f)) = B.Cod (F (A.H.cod f))
    using f 2 F-def A.H-cod-char B.H-cod-char B.null-char
          B.cod.extensional ‹B.H.cod (F f) ≠ B.null›
    by fastforce
  show B.Trn (B.H.cod (F f)) = B.Trn (F (A.H.cod f))
  proof −
    have B.Trn (F (A.H.cod f)) =
         RTS.Map (F_a (A.Cod f) (A.Cod f))
           (RTS.Map (Id_A (A.Cod f)) RTS.One.the-arr)
      using f 2 F-def A.H-cod-char B.H-cod-char EF.preserves-Id
            RTS.Map-comp
      by auto
    also have ... =
                RTS.Map (F_a (A.Cod f) (A.Cod f) · Id_A (A.Cod f))
                  RTS.One.the-arr
      using f 0 1 2 A.Id-in-hom
            EF.preserves-Hom [of A.Cod f A.Cod f]
            EF.preserves-Obj [of A.Cod f] comp-apply RTS.Map-comp
      apply auto[1]
      using A.arr-char [of f] by fastforce
    also have ... = RTS.Map (Id_B (F_o (A.Cod f))) RTS.One.the-arr
      using f 0 1 EF.preserves-Id A.arr-char by simp
    also have ... = B.Trn (B.H.cod (F f))
      using f 0 1 F-def by (auto simp add: B.H-dom-simp B.H-cod-simp)
    finally show ?thesis by simp
  qed
qed
next
fix f g
assume fg: A.H.seq g f
show F (A.hcomp g f) = B.hcomp (F g) (F f)
proof (intro B.arr-eqI)
  show F (A.hcomp g f) ≠ B.null
    using fg F-def B.null-char by auto
  have 2: B.Dom (F g) = B.Cod (F f)
    using fg preserves-arr F-def A.H-seq-char by auto
```

```
show 3: B.hcomp (F g) (F f) ≠ B.null
  using fg 2 preserves-arr B.null-char B.arr-hcomp [of F g F f]
        A.arr-coincidence B.arr-coincidence A.H-seq-char
        B.V.not-arr-null
  by auto
show B.Dom (F (A.hcomp g f)) = B.Dom (B.hcomp (F g) (F f))
  using fg F-def 3 A.H-seq-char B.H-composable-char by auto
show B.Cod (F (A.hcomp g f)) = B.Cod (B.hcomp (F g) (F f))
  using fg F-def 3 A.H-seq-char B.H-composable-char by auto
show B.Trn (F (A.hcomp g f)) = B.Trn (B.hcomp (F g) (F f))
proof −
  have B.Trn (F (A.hcomp g f)) =
       RTS.Map (F_a (A.Dom f) (A.Cod g))
               (RTS.Map (Comp_A (A.Dom f) (A.Cod f) (A.Cod g))
                        (RTS.Pack (Hom_A (A.Cod f) (A.Cod g))
                                  (Hom_A (A.Dom f) (A.Cod f))
                                  (A.Trn g, A.Trn f)))
    using fg F-def A.H-seq-char by auto
  also have ... = (RTS.Map (F_a (A.Dom f) (A.Cod g)) ∘
                   RTS.Map (Comp_A (A.Dom f) (A.Cod f) (A.Cod g)))
                  (RTS.Pack
                     (Hom_A (A.Cod f) (A.Cod g))
                     (Hom_A (A.Dom f) (A.Cod f))
                     (A.Trn g, A.Trn f))
    using fg A.H-seq-char A.Comp-in-hom EF.preserves-Hom
          RTS.Map-comp
    by auto
  also have ... = RTS.Map (F_a (A.Dom f) (A.Cod g) ·
                           Comp_A (A.Dom f) (A.Cod f) (A.Cod g))
                  (RTS.Pack (Hom_A (A.Cod f) (A.Cod g))
                            (Hom_A (A.Dom f) (A.Cod f))
                            (A.Trn g, A.Trn f))
    using fg A.H-seq-char comp-apply A.Comp-in-hom EF.preserves-Hom
          RTS.Map-comp
    apply auto[1]
    by (metis (no-types, lifting) A.arr-char RTS.seqI' comp-apply)
  also have ... = RTS.Map
                  (Comp_B (F_o (A.Dom f)) (F_o (A.Cod f)) (F_o (A.Cod g)) ·
                     (F_a (A.Cod f) (A.Cod g) ⊗ F_a (A.Dom f) (A.Cod f)))
                  (RTS.Pack
                     (Hom_A (A.Cod f) (A.Cod g))
                     (Hom_A (A.Dom f) (A.Cod f))
                     (A.Trn g, A.Trn f))
    using fg A.H-seq-char A.arr-char EF.preserves-Comp by auto
  also have ... = (RTS.Map
                  (Comp_B (F_o (A.Dom f)) (F_o (A.Cod f)) (F_o (A.Cod g))) ∘
                     RTS.Map
                       (F_a (A.Cod f) (A.Cod g) ⊗ F_a (A.Dom f) (A.Cod f)))
                  (RTS.Pack
```

```
                (Hom_A (A.Cod f) (A.Cod g))
                (Hom_A (A.Dom f) (A.Cod f))
                (A.Trn g, A.Trn f))
proof −
  have RTS.seq
         (Comp_B (F_o (A.Dom f)) (F_o (A.Cod f)) (F_o (A.Cod g)))
         (F_a (A.Cod f) (A.Cod g) ⊗ F_a (A.Dom f) (A.Cod f))
    using fg B.Comp-in-hom EF.preserves-Obj
    by (metis A.Comp-in-hom A.H.seqE A.H-arr-char EF.preserves-Comp
        EF.preserves-Hom RTS.arrI RTS.seqI' RTS.tensor-agreement)
  thus ?thesis
    using RTS.Map-comp
          [of Comp_B (F_o (A.Dom f)) (F_o (A.Cod f)) (F_o (A.Cod g))
              F_a (A.Cod f) (A.Cod g) ⊗ F_a (A.Dom f) (A.Cod f)]
    by argo
qed
also have ... = RTS.Map
                (Comp_B (F_o (A.Dom f)) (F_o (A.Cod f)) (F_o (A.Cod g)))
                (RTS.Map
                  (F_a (A.Cod f) (A.Cod g) ⊗ F_a (A.Dom f) (A.Cod f))
                  (RTS.Pack
                    (Hom_A (A.Cod f) (A.Cod g))
                    (Hom_A (A.Dom f) (A.Cod f))
                    (A.Trn g, A.Trn f)))
  by simp
also have ... = RTS.Map
                (Comp_B (F_o (A.Dom f)) (F_o (A.Cod f)) (F_o (A.Cod g)))
                ((RTS.Pack
                   (RTS.cod (F_a (A.Cod f) (A.Cod g)))
                   (RTS.cod (F_a (A.Dom f) (A.Cod f))) ∘
                   product-simulation.map
                     (A.HOM_EC (A.Cod f) (A.Cod g))
                     (A.HOM_EC (A.Dom f) (A.Cod f))
                     (RTS.Map (F_a (A.Cod f) (A.Cod g)))
                     (RTS.Map (F_a (A.Dom f) (A.Cod f))) ∘
                     RTS.Unpack
                       (RTS.dom (F_a (A.Cod f) (A.Cod g)))
                       (RTS.dom (F_a (A.Dom f) (A.Cod f))))
                  (RTS.Pack
                    (Hom_A (A.Cod f) (A.Cod g))
                    (Hom_A (A.Dom f) (A.Cod f))
                    (A.Trn g, A.Trn f)))
  using fg A.H-seq-char EF.preserves-Hom RTS.Map-prod
  by (metis (no-types, lifting) A.arr-char RTS.in-homE)
also have ... = RTS.Map
                (Comp_B (F_o (A.Dom f)) (F_o (A.Cod f)) (F_o (A.Cod g)))
                (RTS.Pack
                  (Hom_B (F_o (A.Cod f)) (F_o (A.Cod g)))
                  (Hom_B (F_o (A.Dom f)) (F_o (A.Cod f))))
```

```
                    (product-simulation.map
                      (A.HOM_EC (A.Cod f) (A.Cod g))
                      (A.HOM_EC (A.Dom f) (A.Cod f))
                      (RTS.Map (F_a (A.Cod f) (A.Cod g)))
                      (RTS.Map (F_a (A.Dom f) (A.Cod f)))
                      (RTS.Unpack
                        (Hom_A (A.Cod f) (A.Cod g))
                        (Hom_A (A.Dom f) (A.Cod f))
                        (RTS.Pack
                          (Hom_A (A.Cod f) (A.Cod g))
                          (Hom_A (A.Dom f) (A.Cod f))
                          (A.Trn g, A.Trn f)))))
proof −
  have RTS.dom (F_a (A.Dom f) (A.Cod f)) =
       Hom_A (A.Dom f) (A.Cod f)
    using fg A.H-seq-char A.arr-char
          EF.preserves-Hom [of A.Dom f A.Cod f]
    by auto
  moreover have RTS.cod (F_a (A.Dom f) (A.Cod f)) =
                Hom_B (F_o (A.Dom f)) (F_o (A.Cod f))
    using fg A.H-seq-char A.arr-char
          EF.preserves-Hom [of A.Dom f A.Cod f]
    by auto
  ultimately show ?thesis
    using fg A.H-seq-char A.arr-char
          EF.preserves-Hom [of A.Cod f A.Cod g]
          EF.preserves-Hom [of A.Dom f A.Cod f]
    by auto
qed
also have ... = RTS.Map
                  (Comp_B (F_o (A.Dom f)) (F_o (A.Cod f)) (F_o (A.Cod g)))
                  (RTS.Pack
                    (Hom_B (F_o (A.Cod f)) (F_o (A.Cod g)))
                    (Hom_B (F_o (A.Dom f)) (F_o (A.Cod f)))
                    (product-simulation.map
                      (A.HOM_EC (A.Cod f) (A.Cod g))
                      (A.HOM_EC (A.Dom f) (A.Cod f))
                      (RTS.Map (F_a (A.Cod f) (A.Cod g)))
                      (RTS.Map (F_a (A.Dom f) (A.Cod f)))
                      (A.Trn g, A.Trn f)))
proof −
  interpret HOM: extensional-rts ‹A.HOM_EC (A.Cod f) (A.Cod g)›
    using fg A.H-seq-char A.arr-char A.ide-Hom
    by (metis (no-types, lifting) EF.preserves-Hom RTS.arrD(1)
        RTS.in-homE)
  interpret HOM': extensional-rts ‹A.HOM_EC (A.Dom f) (A.Cod f)›
    using fg A.H-seq-char A.arr-char A.ide-Hom
    by (metis (no-types, lifting) EF.preserves-Hom RTS.arrD(1)
        RTS.in-homE)
```

```
    interpret HOMxHOM': product-rts
                        ‹A.HOM_EC (A.Cod f) (A.Cod g)›
                        ‹A.HOM_EC (A.Dom f) (A.Cod f)›
      ..
    show ?thesis
      using A.H-arr-char A.H-seq-char fg by force
  qed
  also have ... = RTS.Map
                  (Comp_B (F_o (A.Dom f)) (F_o (A.Cod f)) (F_o (A.Cod g)))
                  (RTS.Pack
                     (Hom_B (F_o (A.Cod f)) (F_o (A.Cod g)))
                     (Hom_B (F_o (A.Dom f)) (F_o (A.Cod f)))
                     (RTS.Map (F_a (A.Cod f) (A.Cod g)) (A.Trn g),
                      RTS.Map (F_a (A.Dom f) (A.Cod f)) (A.Trn f)))
  proof −
    interpret F: simulation
                 ‹A.HOM_EC (A.Cod f) (A.Cod g)›
                 ‹B.HOM_EC (F_o (A.Cod f)) (F_o (A.Cod g))›
                 ‹RTS.Map (F_a (A.Cod f) (A.Cod g))›
      using fg A.H-seq-char A.arr-char EF.preserves-Hom
      by (meson EF.is-local-simulation)
    interpret F': simulation
                  ‹A.HOM_EC (A.Dom f) (A.Cod f)›
                  ‹B.HOM_EC (F_o (A.Dom f)) (F_o (A.Cod f))›
                  ‹RTS.Map (F_a (A.Dom f) (A.Cod f))›
      using fg A.H-seq-char A.arr-char EF.preserves-Hom
      by (meson EF.is-local-simulation)
    interpret FxF': product-simulation
                    ‹A.HOM_EC (A.Cod f) (A.Cod g)›
                    ‹A.HOM_EC (A.Dom f) (A.Cod f)›
                    ‹B.HOM_EC (F_o (A.Cod f)) (F_o (A.Cod g))›
                    ‹B.HOM_EC (F_o (A.Dom f)) (F_o (A.Cod f))›
                    ‹RTS.Map (F_a (A.Cod f) (A.Cod g))›
                    ‹RTS.Map (F_a (A.Dom f) (A.Cod f))›
      ..
    show ?thesis
      by (metis FxF'.map-simp fg A.H-seq-char A.arr-char)
  qed
  also have ... = B.Trn (B.hcomp (F g) (F f))
    using fg 3 F-def A.H-seq-char B.Trn-hcomp B.H-composable-char
    by force
  finally show ?thesis by blast
 qed
qed
next
show ⋀t. ¬ A.arr t ⟹ F t = B.null
 unfolding F-def by simp
show ⋀t u. A.V.con t u ⟹ B.V.con (F t) (F u)
 unfolding F-def
```

```
    using A.V.con-implies-arr B.con-char EF.preserves-Obj
          EF.preserves-Hom A.arr-char A.con-char
    apply auto[1]
    apply (intro B.ConI conjI)
          apply auto[11]
    by (metis A.ConE EF.is-local-simulation
        simulation.preserves-reflects-arr simulation.preserves-con)+
  show ⋀t u. A.V.con t u ⟹ F (A.resid t u) = B.resid (F t) (F u)
    unfolding F-def
    using A.V.con-implies-arr B.con-char EF.preserves-Obj EF.preserves-Hom
          A.arr-char B.null-char A.con-implies-Par
    apply auto[1]
      apply (metis (mono-tags, lifting) A.ConE A.resid-ne-Null-imp-Con
        EF.is-local-simulation simulation.preserves-resid)
     apply (metis (mono-tags, lifting) F-def
       ‹⋀u t. A.V.con t u ⟹ B.V.con (F t) (F u)›)
    apply (meson A.V.arr-resid)
   using A.V.arr-resid by force
qed

lemma is-rts-functor:
shows rts-functor A.resid A.hcomp B.resid B.hcomp F
  ..

end
```

5.3 RTS-Categories induce RTS-Enriched Categories

Here we show that an RTS-category induces a corresponding RTS-enriched category. In order to perform this construction, we will need to have a universe to use as the arrow type of the base category **RTS**. In order to avoid introducing a fixed universe, at this point we assume one is given as a parameter.

```
locale enriched-category-of-rts-category =
  universe arr-type +
  locally-small-rts-category resid hcomp
for arr-type :: 'A itself
and resid :: 'A resid  (infix \ 70)
and hcomp :: 'A comp   (infixr ⋆ 53)
begin

  sublocale RTS: rtscat arr-type ..


  no-notation V.comp      (infixr · 55)
  no-notation H.in-hom    («- : - → -»)
  no-notation RTS.prod    (infixr ⊗ 51)
```

notation *RTS.in-hom* («- : - → -»)
notation *RTS.CMC.tensor* (**infixr** ⊗ *51*)
notation *RTS.CMC.unity* (**1**)
notation *RTS.CMC.assoc* (a[-, -, -])
notation *RTS.CMC.lunit* (l[-])
notation *RTS.CMC.runit* (r[-])

abbreviation *Obj*
where *Obj* ≡ *Collect H.ide*

definition *Hom*
where *Hom a b* ≡
  if a ∈ *Obj* ∧ *b* ∈ *Obj then RTS.mkide* (*HOM a b*) *else RTS.null*

definition *Id*
where *Id a* ≡
  RTS.mkarr RTS.One.resid (*RTS.Rts* (*Hom a a*))
    (λ*t*. *if RTS.One.arr t*
      then a
      else ResiduatedTransitionSystem.partial-magma.null (*HOM a a*))

definition *Comp*
where *Comp a b c* ≡
  RTS.mkarr
    (*RTS.Rts* (*Hom b c* ⊗ *Hom a b*))
    (*RTS.Rts* (*Hom a c*))
    (λ*t*. (λ*x*. *fst x* ⋆ *snd x*) (*RTS.Unpack* (*Hom b c*) (*Hom a b*) *t*))

lemma *ide-Hom* [*intro*, *simp*]:
assumes *a* ∈ *Obj* **and** *b* ∈ *Obj*
shows *RTS.ide* (*Hom a b*)
proof −
  interpret *Hom*: *sub-rts resid* ‹λ*t*. *H.in-hom t a b*›
    using *assms sub-rts-HOM* **by** *blast*
  interpret *Hom*: *extensional-rts Hom.resid*
    using *Hom.preserves-extensional-rts V.extensional-rts-axioms* **by** *blast*
  interpret *Hom*: *small-rts Hom.resid*
    using *assms Hom.arr-char small-homs*
    apply *unfold-locales*
    by (*metis Collect-cong mem-Collect-eq*)
  show *RTS.ide* (*Hom a b*)
    unfolding *Hom-def*
    using *assms RTS.ide-mkide Hom.rts-axioms Hom.small-rts-axioms*
      Hom.extensional-rts-axioms
    by *auto*
qed

```
lemma
assumes a ∈ Obj and b ∈ Obj
shows HOM-null-char: ResiduatedTransitionSystem.partial-magma.null
                       (RTS.Rts (Hom a b)) =
                     null
and HOM-arr-char:
      residuation.arr (RTS.Rts (Hom a b)) t ⟷ H.in-hom t a b
proof −
  interpret Hom: sub-rts resid ‹λt. H.in-hom t a b›
    using assms sub-rts-HOM by blast
  show ResiduatedTransitionSystem.partial-magma.null
         (RTS.Rts (Hom a b)) =
       null
    using assms Hom-def RTS.bij-mkide(3) Hom.null-char by auto
  show residuation.arr (RTS.Rts (Hom a b)) t ⟷ H.in-hom t a b
    unfolding Hom-def
    using assms arr-coincidence Hom.arr-char RTS.bij-mkide(3) by simp
qed

lemma Id-in-hom [intro]:
assumes a ∈ Obj
shows «Id a : 1 → Hom a a»
proof −
  interpret Hom: sub-rts resid ‹λt. H.in-hom t a a›
    using assms sub-rts-HOM by blast
  interpret Hom: extensional-rts Hom.resid
    using Hom.preserves-extensional-rts V.extensional-rts-axioms by blast
  interpret Hom: small-rts Hom.resid
    using assms Hom.arr-char small-homs
    apply unfold-locales
    by (metis Collect-cong mem-Collect-eq)
  interpret I: simulation RTS.One.resid ‹Hom.resid›
                 ‹λt. if RTS.One.arr t then a else Hom.null›
  proof
    show ⋀t. ¬ RTS.One.arr t ⟹
               (if RTS.One.arr t then a else Hom.null) = Hom.null
      by simp
    show 1: ⋀t u. RTS.One.con t u ⟹
                    Hom.con (if RTS.One.arr t then a else Hom.null)
                            (if RTS.One.arr u then a else Hom.null)
      using H.ide-in-hom Hom.arr-char RTS.One.con-implies-arr(1−2) assms
      by auto
    show ⋀t u. RTS.One.con t u ⟹
                 (if RTS.One.arr (t \1 u) then a else Hom.null) =
                 Hom.resid
                   (if RTS.One.arr t then a else Hom.null)
                   (if RTS.One.arr u then a else Hom.null)
      using H.ide-in-hom Hom.resid-def RTS.One.con-implies-arr(1−2)
```

```
        V.resid-arr-ide assms obj-implies-sta
    by force
  qed
  show «Id a : 𝟭 → Hom a a»
  proof
    show 1: RTS.arr (Id a)
    proof (unfold Id-def, intro RTS.arr-mkarr)
      show extensional-rts RTS.One.resid ∧ small-rts RTS.One.resid
        using RTS.One.is-extensional-rts RTS.One.small-rts-axioms by auto
      show extensional-rts (RTS.Rts (Hom a a)) ∧
           small-rts (RTS.Rts (Hom a a))
        using assms by auto
      show simulation (\₁) (RTS.Rts (Hom a a))
           (λt. if RTS.One.arr t then a else Hom.null)
        unfolding Hom-def
        using assms RTS.bij-mkide(3) I.simulation-axioms by auto
    qed
    show RTS.dom (Id a) = 𝟭
      using 1 Id-def by (simp add: RTS.unity-agreement)
    show RTS.cod (Id a) = Hom a a
      using 1 Id-def assms by force
  qed
qed

lemma Id-simps [simp]:
assumes a ∈ Obj
shows RTS.arr (Id a)
and RTS.dom (Id a) = 𝟭
and RTS.cod (Id a) = Hom a a
  using assms Id-in-hom RTS.unity-agreement by auto

lemma Comp-in-hom [intro, simp]:
assumes a ∈ Obj and b ∈ Obj and c ∈ Obj
shows «Comp a b c : Hom b c ⊗ Hom a b → Hom a c»
proof (unfold Comp-def, intro RTS.in-homI)
  show 0: RTS.arr (RTS.mkarr
                   (RTS.Rts (Hom b c ⊗ Hom a b))
                   (RTS.Rts (Hom a c))
                   (λt. fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
                        snd (RTS.Unpack (Hom b c) (Hom a b) t)))
  proof (intro RTS.arr-mkarr)
    show extensional-rts (RTS.Rts (Hom b c ⊗ Hom a b)) ∧
         small-rts (RTS.Rts (Hom b c ⊗ Hom a b))
      using assms by auto
    show extensional-rts (RTS.Rts (Hom a c)) ∧
         small-rts (RTS.Rts (Hom a c))
      using assms by auto
    show simulation (RTS.Rts (Hom b c ⊗ Hom a b)) (RTS.Rts (Hom a c))
         (λt. fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
```

```
        snd (RTS.Unpack (Hom b c) (Hom a b) t))
proof −
  interpret ac: extensional-rts ‹RTS.Rts (Hom a c)›
    using assms by auto
  interpret bc: extensional-rts ‹RTS.Rts (Hom b c)›
    using assms by simp
  interpret ab: extensional-rts ‹RTS.Rts (Hom a b)›
    using assms by simp
  interpret HOM-ab: sub-rts resid ‹λt. H.in-hom t a b›
    using assms sub-rts-HOM by blast
  interpret HOM-bc: sub-rts resid ‹λt. H.in-hom t b c›
    using assms sub-rts-HOM by blast
  interpret HOM-ac: sub-rts resid ‹λt. H.in-hom t a c›
    using assms sub-rts-HOM by blast
  interpret bcxab: extensional-rts ‹RTS.Rts (Hom b c ⊗ Hom a b)›
    using assms by auto
  interpret bcXab: product-rts ‹RTS.Rts (Hom b c)› ‹RTS.Rts (Hom a b)›
    ..
  interpret U: simulation
                ‹RTS.Rts (Hom b c ⊗ Hom a b)› bcXab.resid
                ‹RTS.Unpack (Hom b c) (Hom a b)›
    using assms RTS.simulation-Unpack by simp
  show ?thesis
  proof
    show ⋀t. ¬ bcxab.arr t ⟹
                  fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
                    snd (RTS.Unpack (Hom b c) (Hom a b) t) =
                  ac.null
      using assms H.null-is-zero(2) HOM-null-char U.simulation-axioms
            simulation.extensional
      by fastforce
    fix t u
    assume tu: bcxab.con t u
    have 1: HOM-ab.con = ab.con ∧ HOM-bc.con = bc.con ∧
              HOM-ac.con = ac.con
      using assms Hom-def arr-coincidence null-coincidence
            RTS.bij-mkide(3)
      by auto
    have 2: H.in-hom (fst (RTS.Unpack (Hom b c) (Hom a b) t)) b c ∧
            H.in-hom (snd (RTS.Unpack (Hom b c) (Hom a b) t)) a b ∧
            H.in-hom (fst (RTS.Unpack (Hom b c) (Hom a b) u)) b c ∧
            H.in-hom (snd (RTS.Unpack (Hom b c) (Hom a b) u)) a b
      using assms tu bcxab.con-implies-arr Hom-def
            U.preserves-reflects-arr bcXab.arr-char HOM-ab.arr-char
            HOM-bc.arr-char RTS.bij-mkide(3)
      by auto
    hence 3: H.in-hom (fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
                         snd (RTS.Unpack (Hom b c) (Hom a b) t))
               a c ∧
```

```
        H.in-hom (fst (RTS.Unpack (Hom b c) (Hom a b) u) ⋆
                  snd (RTS.Unpack (Hom b c) (Hom a b) u))
          a c
  using assms arr-coincidence by blast
have 4: V.con (fst (RTS.Unpack (Hom b c) (Hom a b) t))
          (fst (RTS.Unpack (Hom b c) (Hom a b) u)) ∧
        V.con (snd (RTS.Unpack (Hom b c) (Hom a b) t))
          (snd (RTS.Unpack (Hom b c) (Hom a b) u))
  using tu 1 2 3 bcXab.con-char HOM-bc.con-char HOM-ab.con-char
        U.preserves-con
  by auto
hence 5: VV.con
          (RTS.Unpack (Hom b c) (Hom a b) t)
          (RTS.Unpack (Hom b c) (Hom a b) u)
  using 2 3 bcXab.con-char VV.con-char Hom-def arr-coincidence
        null-coincidence
  by fast
hence 6: fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
           snd (RTS.Unpack (Hom b c) (Hom a b) t) ⌢
         fst (RTS.Unpack (Hom b c) (Hom a b) u) ⋆
           snd (RTS.Unpack (Hom b c) (Hom a b) u)
  using H.preserves-con VV.con-implies-arr by auto
thus ac.con
       (fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
          snd (RTS.Unpack (Hom b c) (Hom a b) t))
       (fst (RTS.Unpack (Hom b c) (Hom a b) u) ⋆
          snd (RTS.Unpack (Hom b c) (Hom a b) u))
  using 1 3 HOM-ac.con-char by simp
show fst (RTS.Unpack (Hom b c) (Hom a b)
            (RTS.Rts (Hom b c ⊗ Hom a b) t u)) ⋆
       snd (RTS.Unpack (Hom b c) (Hom a b)
              (RTS.Rts (Hom b c ⊗ Hom a b) t u)) =
     RTS.Rts (Hom a c)
       (fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
          snd (RTS.Unpack (Hom b c) (Hom a b) t))
       (fst (RTS.Unpack (Hom b c) (Hom a b) u) ⋆
          snd (RTS.Unpack (Hom b c) (Hom a b) u))
proof −
  have RTS.Rts (Hom a c)
         (fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
            snd (RTS.Unpack (Hom b c) (Hom a b) t))
         (fst (RTS.Unpack (Hom b c) (Hom a b) u) ⋆
            snd (RTS.Unpack (Hom b c) (Hom a b) u)) =
       (fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
          snd (RTS.Unpack (Hom b c) (Hom a b) t)) \
       (fst (RTS.Unpack (Hom b c) (Hom a b) u) ⋆
          snd (RTS.Unpack (Hom b c) (Hom a b) u))
    using assms 3 6 Hom-def HOM-ac.resid-def RTS.bij-mkide(3)
    by simp
```

```
  also have ... = fst (VV.resid
                         (RTS.Unpack (Hom b c) (Hom a b) t)
                         (RTS.Unpack (Hom b c) (Hom a b) u)) ⋆
                  snd (VV.resid
                         (RTS.Unpack (Hom b c) (Hom a b) t)
                         (RTS.Unpack (Hom b c) (Hom a b) u))
    using 5 VV.con-implies-arr H.preserves-resid by simp
  also have ... = fst (bcXab.resid
                         (RTS.Unpack (Hom b c) (Hom a b) t)
                         (RTS.Unpack (Hom b c) (Hom a b) u)) ⋆
                  snd (bcXab.resid
                         (RTS.Unpack (Hom b c) (Hom a b) t)
                         (RTS.Unpack (Hom b c) (Hom a b) u))
  proof −
    have RTS.Rts (Hom b c)
           (fst (RTS.Unpack (Hom b c) (Hom a b) t))
           (fst (RTS.Unpack (Hom b c) (Hom a b) u)) =
         fst (RTS.Unpack (Hom b c) (Hom a b) t) \
           fst (RTS.Unpack (Hom b c) (Hom a b) u)
      using 2 Hom-def RTS.bij-mkide(3)
            HOM-bc.resid-def
            [of fst (RTS.Unpack (Hom b c) (Hom a b) t)
                fst (RTS.Unpack (Hom b c) (Hom a b) u)]
      apply auto[1]
      by auto
    moreover have RTS.Rts (Hom a b)
                    (snd (RTS.Unpack (Hom b c) (Hom a b) t))
                    (snd (RTS.Unpack (Hom b c) (Hom a b) u)) =
                  snd (RTS.Unpack (Hom b c) (Hom a b) t) \
                    snd (RTS.Unpack (Hom b c) (Hom a b) u)
      using 2 Hom-def RTS.bij-mkide(3)
            HOM-ab.resid-def
            [of snd (RTS.Unpack (Hom b c) (Hom a b) t)
                snd (RTS.Unpack (Hom b c) (Hom a b) u)]
      apply auto[1]
      by auto
    ultimately show ?thesis
      using tu 2 4 bcXab.resid-def bcXab.con-char VV.resid-def
            U.preserves-con
      apply auto[1]
      by fastforce+
  qed
  also have ... = fst (RTS.Unpack (Hom b c) (Hom a b)
                         (RTS.Rts (Hom b c ⊗ Hom a b) t u)) ⋆
                  snd (RTS.Unpack (Hom b c) (Hom a b)
                         (RTS.Rts (Hom b c ⊗ Hom a b) t u))
    using tu U.preserves-resid by simp
  finally show ?thesis by simp
qed
```

```
    qed
  qed
 qed
 show RTS.dom
       (RTS.mkarr (RTS.Rts (Hom b c ⊗ Hom a b)) (RTS.Rts (Hom a c))
                  (λt. fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
                       snd (RTS.Unpack (Hom b c) (Hom a b) t))) =
      Hom b c ⊗ Hom a b
   using assms 0 by auto
 show RTS.cod
       (RTS.mkarr (RTS.Rts (Hom b c ⊗ Hom a b)) (RTS.Rts (Hom a c))
                  (λt. fst (RTS.Unpack (Hom b c) (Hom a b) t) ⋆
                       snd (RTS.Unpack (Hom b c) (Hom a b) t))) =
      Hom a c
   using assms 0 by auto
qed

lemma Comp-simps [simp]:
assumes a ∈ Obj and b ∈ Obj and c ∈ Obj
shows RTS.arr (Comp a b c)
and RTS.dom (Comp a b c) = Hom b c ⊗ Hom a b
and RTS.cod (Comp a b c) = Hom a c
 using assms Comp-in-hom RTS.in-homE by auto

lemma Map-Comp-Pack:
assumes a ∈ Obj and b ∈ Obj and c ∈ Obj
and residuation.arr
      (product-rts.resid (RTS.Rts (Hom b c)) (RTS.Rts (Hom a b))) x
shows RTS.Map (Comp a b c) (RTS.Pack (Hom b c) (Hom a b) x) =
      fst x ⋆ snd x
 using assms Comp-def RTS.bij-mkarr(3) by simp

sublocale rts-enriched-category arr-type Obj Hom Id Comp
proof
 show ⋀a b. ⟦a ∈ Obj; b ∈ Obj⟧ ⟹ RTS.ide (Hom a b)
   by blast
 show ⋀a. a ∈ Obj ⟹ «Id a : 𝟭 → Hom a a»
   using Id-in-hom RTS.unity-agreement by auto
 show ⋀a b c. ⟦a ∈ Obj; b ∈ Obj; c ∈ Obj⟧ ⟹
                «Comp a b c : Hom b c ⊗ Hom a b → Hom a c»
   using Comp-in-hom by auto

 fix a b
 assume a: a ∈ Obj and b: b ∈ Obj
 interpret ab: extensional-rts ‹RTS.Rts (Hom a b)›
   using a b by simp
 interpret aa: extensional-rts ‹RTS.Rts (Hom a a)›
   using a by simp
 interpret bb: extensional-rts ‹RTS.Rts (Hom b b)›
```

using *b* **by** *simp*
interpret *abXaa*: *product-rts* ‹*RTS.Rts* (*Hom a b*)› ‹*RTS.Rts* (*Hom a a*)› **..**
interpret *bbXab*: *product-rts* ‹*RTS.Rts* (*Hom b b*)› ‹*RTS.Rts* (*Hom a b*)› **..**
interpret *ab*: *simulation* ‹*RTS.Rts* (*Hom a b*)› ‹*RTS.Rts* (*Hom a b*)›
‹*RTS.Map* (*Hom a b*)›
using *a b ide-Hom RTS.arrD*
by (*metis* (*no-types*, *lifting*) *RTS.ide-char*)
interpret *I*: *simulation RTS.One.resid* ‹*RTS.Rts* (*Hom a a*)›
‹*RTS.Map* (*Id a*)›
using *a ide-Hom Id-in-hom RTS.ide-char*
by (*metis* (*no-types*, *lifting*) *RTS.Rts-one RTS.arrD*(*3*) *RTS.in-homE*
RTS.unity-agreement)
interpret *Ib*: *simulation RTS.One.resid* ‹*RTS.Rts* (*Hom b b*)›
‹*RTS.Map* (*Id b*)›
using *b ide-Hom Id-in-hom RTS.ide-char*
by (*metis* (*no-types*, *lifting*) *RTS.Rts-one RTS.arrD*(*3*) *RTS.in-homE*
RTS.unity-agreement)
interpret *abXI*: *product-simulation*
‹*RTS.Rts* (*Hom a b*)› *RTS.One.resid*
‹*RTS.Rts* (*Hom a b*)› ‹*RTS.Rts*(*Hom a a*)›
‹*RTS.Map* (*Hom a b*)› ‹*RTS.Map* (*Id a*)›
..
interpret *IXab*: *product-simulation*
RTS.One.resid ‹*RTS.Rts* (*Hom a b*)›
‹*RTS.Rts* (*Hom b b*)› ‹*RTS.Rts* (*Hom a b*)›
‹*RTS.Map* (*Id b*)› ‹*RTS.Map* (*Hom a b*)›
..
interpret *abxone*: *extensional-rts* ‹*RTS.Rts* (*Hom a b* ⊗ **1**)›
using *a b* **by** *auto*
interpret *onexab*: *extensional-rts* ‹*RTS.Rts* (**1** ⊗ *Hom a b*)›
using *a b* **by** *auto*
interpret *PU-abXI*: *inverse-simulations*
‹*RTS.Rts* (*Hom a b* ⊗ **1**)› ‹*abXI.A1xA0.resid*›
‹*RTS.Pack* (*Hom a b*) **1**› ‹*RTS.Unpack* (*Hom a b*) **1**›
using *a b RTS.ide-one*
RTS.inverse-simulations-Pack-Unpack [*of Hom a b* **1**]
by *simp*
interpret *PU-IXab*: *inverse-simulations*
‹*RTS.Rts* (**1** ⊗ *Hom a b*)› ‹*IXab.A1xA0.resid*›
‹*RTS.Pack* **1** (*Hom a b*)› ‹*RTS.Unpack* **1** (*Hom a b*)›
using *a b RTS.ide-one*
RTS.inverse-simulations-Pack-Unpack [*of* **1** *Hom a b*]
by *simp*
interpret *PU-abXaa*: *inverse-simulations*
‹*RTS.Rts* (*Hom a b* ⊗ *Hom a a*)› *abXaa.resid*
‹*RTS.Pack* (*Hom a b*) (*Hom a a*)›
‹*RTS.Unpack* (*Hom a b*) (*Hom a a*)›
using *a b RTS.ide-one*
RTS.inverse-simulations-Pack-Unpack [*of Hom a b Hom a a*]

```
  by simp
interpret PU-bbXab: inverse-simulations
                    ‹RTS.Rts (Hom b b ⊗ Hom a b)› bbXab.resid
                    ‹RTS.Pack (Hom b b) (Hom a b)›
                    ‹RTS.Unpack (Hom b b) (Hom a b)›
  using a b RTS.ide-one
        RTS.inverse-simulations-Pack-Unpack [of Hom b b Hom a b]
  by simp

show Comp a a b · (Hom a b ⊗ Id a) = r[Hom a b]
proof (intro RTS.arr-eqI)
  show 1: RTS.par (Comp a a b · (Hom a b ⊗ Id a)) r[Hom a b]
    using a b Id-in-hom by fastforce+
  show RTS.Map (Comp a a b · (Hom a b ⊗ Id a)) = RTS.Map r[Hom a b]
  proof −
    have RTS.Map (Comp a a b · (Hom a b ⊗ Id a)) =
         RTS.Map (Comp a a b) ∘ RTS.Map (Hom a b ⊗ Id a)
      using a b 1 Comp-in-hom Id-in-hom RTS.Map-comp by blast
    also have ... = (λt. fst ((RTS.Unpack (Hom a b) (Hom a a) ∘
                                 RTS.Pack (Hom a b) (Hom a a) ∘
                                   abXI.map ∘
                                     RTS.Unpack (Hom a b) 1) t)
                       ⋆
                       snd ((RTS.Unpack (Hom a b) (Hom a a) ∘
                               RTS.Pack (Hom a b) (Hom a a) ∘
                                 abXI.map ∘
                                   RTS.Unpack (Hom a b) 1) t))
      using a b ide-Hom Id-in-hom Comp-in-hom Comp-def RTS.Map-prod
            RTS.Map-mkarr RTS.tensor-agreement RTS.bij-mkarr(3)
            RTS.unity-agreement
      by auto
    also have ... = (λt. fst ((I abXaa.resid ∘ abXI.map ∘
                                 RTS.Unpack (Hom a b) 1) t) ⋆
                       snd ((I abXaa.resid ∘ abXI.map ∘
                               RTS.Unpack (Hom a b) 1) t))
      using PU-abXaa.inv by auto
    also have ... = RTS.Map r[Hom a b]
    proof
      fix t
      show fst ((I abXaa.resid ∘ abXI.map ∘
                   RTS.Unpack (Hom a b) 1) t) ⋆
           snd ((I abXaa.resid ∘ abXI.map ∘
                   RTS.Unpack (Hom a b) 1) t) =
           RTS.Map r[Hom a b] t
      proof (cases abxone.arr t)
        show ¬ abxone.arr t ⟹ ?thesis
        proof −
          assume t: ¬ abxone.arr t
          have fst ((I abXaa.resid ∘ abXI.map ∘
```

```
            RTS.Unpack (Hom a b) 1) t) ⋆
        snd ((I abXaa.resid ∘ abXI.map ∘
            RTS.Unpack (Hom a b) 1) t) =
        null
    using a b t 1 PU-abXI.G.extensional abXI.extensional
      abXI.A1xA0.P₁.extensional H.null-is-zero(2)
      HOM-null-char null-coincidence
    by simp
  also have ... = RTS.Map r[Hom a b] t
  proof −
    interpret R: simulation ‹RTS.Rts (Hom a b ⊗ 1)›
      ‹RTS.Rts (Hom a b)› ‹RTS.Map r[Hom a b]›
      using a b 1 ide-Hom RTS.arrD(3)
      by (metis (no-types, lifting) RTS.CMC.cod-runit
          RTS.CMC.dom-runit)
    show ?thesis
      using a b t R.extensional HOM-null-char by simp
  qed
  finally show ?thesis by blast
qed
assume t: abxone.arr t
have (I abXaa.resid ∘ abXI.map ∘ RTS.Unpack (Hom a b) 1) t =
     (RTS.Map r[Hom a b] t, a)
proof −
  have (I abXaa.resid ∘ abXI.map ∘ RTS.Unpack (Hom a b) 1) t =
       I abXaa.resid (abXI.map (RTS.Unpack (Hom a b) 1 t))
    by auto
  also have ... = abXI.map (RTS.Unpack (Hom a b) 1 t)
    using a b t abXI.preserves-reflects-arr
          PU-abXI.G.preserves-reflects-arr
    by simp
  also have ... = (fst (RTS.Unpack (Hom a b) 1 t),
                   RTS.Map (Id a)
                     (snd (RTS.Unpack (Hom a b) 1 t)))
    using a b t PU-abXI.G.preserves-reflects-arr
          RTS.Map-ide
          abXI.map-simp [of fst (RTS.Unpack (Hom a b) 1 t)
                         snd (RTS.Unpack (Hom a b) 1 t)]
    by auto
  also have ... = (fst (RTS.Unpack (Hom a b) 1 t), a)
    unfolding Id-def
    using a b t PU-abXI.G.preserves-reflects-arr
          Id-def RTS.bij-mkarr(3) RTS.ide-one
          RTS.One.is-extensional-rts RTS.One.small-rts-axioms
    by auto
  also have ... = (RTS.Map r[Hom a b] t, a)
    using a b t RTS.runit-agreement RTS.Map-runit
          abXI.A1xA0.P₁-def RTS.unity-agreement
          PU-abXI.G.preserves-reflects-arr
```

```
        by auto
      finally show ?thesis by blast
    qed
    moreover have H.in-hom (RTS.Map r[Hom a b] t) a b
      using a b t RTS.Map-runit PU-abXI.G.preserves-reflects-arr
          HOM-arr-char
            [of a b (abXI.A1xA0.P₁ ∘ RTS.Unpack (Hom a b) 1) t]
          RTS.runit-agreement RTS.unity-agreement
      by auto
    ultimately show ?thesis
      using a b H.comp-arr-dom by auto
  qed
 qed
 finally show ?thesis by blast
qed

show Comp a b b · (Id b ⊗ Hom a b) = l[Hom a b]
proof (intro RTS.arr-eqI)
 show 1: RTS.par (Comp a b b · (Id b ⊗ Hom a b)) l[Hom a b]
  using a b Id-in-hom [of b] by auto
 show RTS.Map (Comp a b b · (Id b ⊗ Hom a b)) = RTS.Map l[Hom a b]
 proof −
  have RTS.Map (Comp a b b · (Id b ⊗ Hom a b)) =
       RTS.Map (Comp a b b) ∘ RTS.Map (Id b ⊗ Hom a b)
    using a b 1 Comp-in-hom Id-in-hom RTS.Map-comp by blast
  also have ... = (λt. fst (RTS.Unpack (Hom b b) (Hom a b) t) ⋆
                        snd (RTS.Unpack (Hom b b) (Hom a b) t)) ∘
                  (RTS.Pack (Hom b b) (Hom a b) ∘
                    IXab.map ∘
                      RTS.Unpack 1 (Hom a b))
    using a b ide-Hom Id-in-hom Comp-in-hom [of a b b]
        Comp-def RTS.Map-prod RTS.Map-mkarr RTS.tensor-agreement
        RTS.unity-agreement
    by auto
  also have ... = (λt. fst ((RTS.Unpack (Hom b b) (Hom a b) ∘
                             RTS.Pack (Hom b b) (Hom a b) ∘
                               IXab.map ∘
                                 RTS.Unpack 1 (Hom a b)) t)
                    ⋆
                    snd ((RTS.Unpack (Hom b b) (Hom a b) ∘
                             RTS.Pack (Hom b b) (Hom a b) ∘
                               IXab.map ∘
                                 RTS.Unpack 1 (Hom a b)) t))
    by auto
  also have ... = (λt. fst ((I bbXab.resid ∘ IXab.map ∘
                             RTS.Unpack 1 (Hom a b)) t) ⋆
                      snd ((I bbXab.resid ∘ IXab.map ∘
                             RTS.Unpack 1 (Hom a b)) t))
```

```
  using PU-bbXab.inv by auto
also have ... = RTS.Map ⫷[Hom a b]
proof
  fix t
  show fst ((I bbXab.resid ∘ IXab.map ∘
                RTS.Unpack 𝟭 (Hom a b)) t) ⋆
         snd ((I bbXab.resid ∘ IXab.map ∘
                 RTS.Unpack 𝟭 (Hom a b)) t) =
       RTS.Map ⫷[Hom a b] t
  proof (cases onexab.arr t)
    show ¬ onexab.arr t ⟹ ?thesis
    proof −
      assume t: ¬ onexab.arr t
     have fst ((I bbXab.resid ∘ IXab.map ∘ RTS.Unpack 𝟭 (Hom a b)) t) ⋆
          snd ((I bbXab.resid ∘ IXab.map ∘ RTS.Unpack 𝟭 (Hom a b)) t) =
            null
        using a b t 1 PU-IXab.G.extensional IXab.extensional
              IXab.A1xA0.P₀.extensional HOM-null-char
        apply auto[1]
        by (metis H.null-is-zero(2) null-coincidence)+
      also have ... = RTS.Map ⫷[Hom a b] t
      proof −
        interpret L: simulation
                     ‹RTS.Rts (𝟭 ⊗ Hom a b)› ‹RTS.Rts (Hom a b)›
                     ‹RTS.Map ⫷[Hom a b]›
          using a b t 1 RTS.arrD(3) [of ⫷[Hom a b]] by force
        show ?thesis
          using t L.extensional HOM-null-char a b by force
      qed
      finally show ?thesis by blast
    qed
    assume t: onexab.arr t
    have (I bbXab.resid ∘ IXab.map ∘ RTS.Unpack 𝟭 (Hom a b)) t =
         (b, RTS.Map ⫷[Hom a b] t)
    proof −
      have (I bbXab.resid ∘ IXab.map ∘ RTS.Unpack 𝟭 (Hom a b)) t =
           I bbXab.resid (IXab.map (RTS.Unpack 𝟭 (Hom a b) t))
        by auto
      also have ... = IXab.map (RTS.Unpack 𝟭 (Hom a b) t)
    using a b t IXab.preserves-reflects-arr PU-IXab.G.preserves-reflects-arr
        by simp
     also have ... = (RTS.Map (Id b) (fst (RTS.Unpack 𝟭 (Hom a b) t)),
                      snd (RTS.Unpack 𝟭 (Hom a b) t))
        using a b t PU-IXab.G.preserves-reflects-arr RTS.Map-ide
              IXab.map-simp
              [of fst (RTS.Unpack 𝟭 (Hom a b) t)
                  snd (RTS.Unpack 𝟭 (Hom a b) t)]
        by auto
      also have ... = (b, snd (RTS.Unpack 𝟭 (Hom a b) t))
```

```
        unfolding Id-def
        using a b t PU-IXab.G.preserves-reflects-arr Id-def
              RTS.One.is-extensional-rts
              RTS.One.small-rts-axioms RTS.bij-mkarr(3)
        by auto
      also have ... = (b, RTS.Map l[Hom a b] t)
        using a b t RTS.lunit-agreement RTS.Map-lunit
              IXab.A1xA0.P₀-def RTS.unity-agreement
              PU-IXab.G.preserves-reflects-arr
        by auto
      finally show ?thesis by blast
    qed
    moreover have H.in-hom (RTS.Map l[Hom a b] t) a b
      using a b t RTS.Map-lunit
            RTS.lunit-agreement RTS.unity-agreement
            PU-IXab.G.preserves-reflects-arr
            HOM-arr-char
              [of a b (IXab.A1xA0.P₀ ∘ RTS.Unpack 1 (Hom a b)) t]
      by auto
    ultimately show ?thesis
      using a b H.comp-cod-arr by auto
  qed
 qed
 finally show ?thesis by blast
qed
qed
next
show ⋀a b c d. ⟦a ∈ Obj; b ∈ Obj; c ∈ Obj; d ∈ Obj⟧
                ⟹ Comp a b d · (Comp b c d ⊗ Hom a b) =
                   Comp a c d · (Hom c d ⊗ Comp a b c) ·
                     a[Hom c d, Hom b c, Hom a b]
proof −
 fix a b c d
 assume a: a ∈ Obj and b: b ∈ Obj and c: c ∈ Obj and d: d ∈ Obj
 interpret ab: extensional-rts ‹RTS.Rts (Hom a b)›
   using a b by fastforce
 interpret bc: extensional-rts ‹RTS.Rts (Hom b c)›
   using b c by fastforce
 interpret cd: extensional-rts ‹RTS.Rts (Hom c d)›
   using c d by fastforce
 interpret ac: extensional-rts ‹RTS.Rts (Hom a c)›
   using a c by fastforce
 interpret bd: extensional-rts ‹RTS.Rts (Hom b d)›
   using b d by fastforce
 interpret bcxab: extensional-rts
                    ‹RTS.Rts (RTS.dom (Hom b c ⊗ Hom a b))›
   using a b c by auto
 interpret cdxbc: rts ‹RTS.Rts (RTS.dom (Hom c d ⊗ Hom b c))›
   using b c d ide-Hom RTS.ide-char RTS.arrD extensional-rts-def
```

by (*metis RTS.CMC.tensor-preserves-ide*)
interpret *cdxbc-x-ab*:
rts ‹*RTS.Rts* (*RTS.dom* ((*Hom c d* ⊗ *Hom b c*) ⊗ *Hom a b*))›
using *a b c d extensional-rts-def* **by** *fastforce*
interpret *cd-X-bcxab*: *product-rts*
‹*RTS.Rts* (*Hom c d*)›
‹*RTS.Rts* (*RTS.dom* (*Hom b c* ⊗ *Hom a b*))›
..
interpret *cdxbc-X-ab*: *product-rts*
‹*RTS.Rts* (*RTS.dom* (*Hom c d* ⊗ *Hom b c*))›
‹*RTS.Rts* (*Hom a b*)›
..
interpret *bcXab*: *product-rts* ‹*RTS.Rts* (*Hom b c*)› ‹*RTS.Rts* (*Hom a b*)› ..
interpret *cdXbc*: *product-rts* ‹*RTS.Rts* (*Hom c d*)› ‹*RTS.Rts* (*Hom b c*)› ..
interpret *cd-X-bcXab*: *product-rts* ‹*RTS.Rts* (*Hom c d*)› *bcXab.resid* ..
interpret *cdXbc-X-ab*: *product-rts cdXbc.resid* ‹*RTS.Rts* (*Hom a b*)› ..
interpret *Icd*: *simulation* ‹*RTS.Rts* (*Hom c d*)› ‹*RTS.Rts* (*Hom c d*)›
‹*RTS.Map* (*Hom c d*)›
by (*metis RTS.arrD*(*3*) *RTS.ide-char c d ide-Hom*)
interpret *Iab*: *simulation* ‹*RTS.Rts* (*Hom a b*)› ‹*RTS.Rts* (*Hom a b*)›
‹*RTS.Map* (*Hom a b*)›
by (*metis RTS.arrD*(*3*) *RTS.ideD*(*1*−*3*) *a b ide-Hom*)
interpret *Cabc*: *simulation*
‹*RTS.Rts* (*RTS.dom* (*Hom b c* ⊗ *Hom a b*))›
‹*RTS.Rts* (*Hom a c*)›
‹*RTS.Map* (*Comp a b c*)›
by (*metis* (*no-types*, *lifting*) *Comp-in-hom*
RTS.CMC.tensor-preserves-ide RTS.arrD(*3*) *RTS.ideD*(*2*)
RTS.in-homE a b c ide-Hom)
interpret *Cbcd*: *simulation*
‹*RTS.Rts* (*RTS.dom* (*Hom c d* ⊗ *Hom b c*))›
‹*RTS.Rts* (*Hom b d*)›
‹*RTS.Map* (*Comp b c d*)›
by (*metis* (*no-types*, *lifting*) *Comp-in-hom*
RTS.CMC.tensor-preserves-ide RTS.arrD(*3*) *RTS.ideD*(*2*)
RTS.in-homE b c d ide-Hom)
interpret *Cabd*: *simulation*
‹*RTS.Rts* (*RTS.dom* (*Hom b d* ⊗ *Hom a b*))›
‹*RTS.Rts* (*Hom a d*)›
‹*RTS.Map* (*Comp a b d*)›
by (*metis* (*no-types*, *lifting*) *Comp-in-hom*
RTS.CMC.tensor-preserves-ide RTS.arrD(*3*) *RTS.ideD*(*2*)
RTS.in-homE a b d ide-Hom)
interpret *IcdXCabc*: *product-simulation*
‹*RTS.Rts* (*Hom c d*)›
‹*RTS.Rts* (*RTS.dom* (*Hom b c* ⊗ *Hom a b*))›
‹*RTS.Rts* (*Hom c d*)› ‹*RTS.Rts* (*Hom a c*)›
‹*RTS.Map* (*Hom c d*)› ‹*RTS.Map* (*Comp a b c*)›
..

interpret *CbcdXIab*: *product-simulation*
‹*RTS.Rts* (*RTS.dom* (*Hom c d* ⊗ *Hom b c*))›
‹*RTS.Rts* (*Hom a b*)›
‹*RTS.Rts* (*Hom b d*)› ‹*RTS.Rts* (*Hom a b*)›
‹*RTS.Map* (*Comp b c d*)› ‹*RTS.Map* (*Hom a b*)›
..
interpret *PU-bcXab*: *inverse-simulations*
‹*RTS.Rts* (*RTS.dom* (*Hom b c* ⊗ *Hom a b*))›
bcXab.resid
‹*RTS.Pack* (*Hom b c*) (*Hom a b*)›
‹*RTS.Unpack* (*Hom b c*) (*Hom a b*)›
using *a b c RTS.inverse-simulations-Pack-Unpack* **by** *simp*
interpret *PU-bdXab*: *inverse-simulations*
‹*RTS.Rts* (*RTS.dom* (*Hom b d* ⊗ *Hom a b*))›
CbcdXIab.B1xB0.resid
‹*RTS.Pack* (*Hom b d*) (*Hom a b*)›
‹*RTS.Unpack* (*Hom b d*) (*Hom a b*)›
using *a b c d RTS.inverse-simulations-Pack-Unpack* **by** *simp*
interpret *PU-cdXac*: *inverse-simulations*
‹*RTS.Rts* (*RTS.dom* (*Hom c d* ⊗ *Hom a c*))›
IcdXCabc.B1xB0.resid
‹*RTS.Pack* (*Hom c d*) (*Hom a c*)›
‹*RTS.Unpack* (*Hom c d*) (*Hom a c*)›
using *a c d RTS.inverse-simulations-Pack-Unpack* **by** *simp*
interpret *PU-cdXbc*: *inverse-simulations*
‹*RTS.Rts* (*RTS.dom* (*Hom c d* ⊗ *Hom b c*))›
cdXbc.resid
‹*RTS.Pack* (*Hom c d*) (*Hom b c*)›
‹*RTS.Unpack* (*Hom c d*) (*Hom b c*)›
using *b c d RTS.inverse-simulations-Pack-Unpack* **by** *simp*
interpret *PU-cd-X-bcxab*: *inverse-simulations*
‹*RTS.Rts*
(*RTS.dom* (*Hom c d* ⊗ *Hom b c* ⊗ *Hom a b*))›
cd-X-bcxab.resid
‹*RTS.Pack* (*Hom c d*) (*Hom b c* ⊗ *Hom a b*)›
‹*RTS.Unpack* (*Hom c d*) (*Hom b c* ⊗ *Hom a b*)›
using *a b c d RTS.inverse-simulations-Pack-Unpack* **by** *simp*
interpret *PU-cdxbc-X-ab*: *inverse-simulations*
‹*RTS.Rts*
(*RTS.dom* ((*Hom c d* ⊗ *Hom b c*) ⊗ *Hom a b*))›
cdxbc-X-ab.resid
‹*RTS.Pack* (*Hom c d* ⊗ *Hom b c*) (*Hom a b*)›
‹*RTS.Unpack* (*Hom c d* ⊗ *Hom b c*) (*Hom a b*)›
using *a b c d RTS.inverse-simulations-Pack-Unpack* **by** *simp*
interpret *Ucdxbc-X-Iab*: *product-simulation*
‹*RTS.Rts* (*RTS.dom* (*Hom c d* ⊗ *Hom b c*))›
‹*RTS.Rts* (*Hom a b*)›
cdXbc.resid ‹*RTS.Rts* (*Hom a b*)›
‹*RTS.Unpack* (*Hom c d*) (*Hom b c*)›

```
                    ‹RTS.Map (Hom a b)›
  ..
interpret Icd-X-Pbcxab: product-simulation
                    ‹RTS.Rts (Hom c d)› bcXab.resid
                    ‹RTS.Rts (Hom c d)›
                    ‹RTS.Rts (RTS.dom (Hom b c ⊗ Hom a b))›
                  ‹RTS.Map (Hom c d)› ‹RTS.Pack (Hom b c) (Hom a b)›
  ..
show Comp a b d · (Comp b c d ⊗ Hom a b) =
     Comp a c d · (Hom c d ⊗ Comp a b c) · a[Hom c d, Hom b c, Hom a b]
proof (intro RTS.arr-eqI)
  show 1: RTS.par (Comp a b d · (Comp b c d ⊗ Hom a b))
                  (Comp a c d · (Hom c d ⊗ Comp a b c) ·
                    a[Hom c d, Hom b c, Hom a b])
    using a b c d by auto
  show RTS.Map (Comp a b d · (Comp b c d ⊗ Hom a b)) =
       RTS.Map (Comp a c d · (Hom c d ⊗ Comp a b c) ·
                  a[Hom c d, Hom b c, Hom a b])
  proof
    fix x
    show RTS.Map (Comp a b d · (Comp b c d ⊗ Hom a b)) x =
         RTS.Map (Comp a c d · (Hom c d ⊗ Comp a b c) ·
                    a[Hom c d, Hom b c, Hom a b]) x
    proof (cases cdxbc-x-ab.arr x)
      assume x: ¬ cdxbc-x-ab.arr x
      show ?thesis
      proof −
        interpret L: simulation
                    ‹RTS.Dom ((Hom c d ⊗ Hom b c) ⊗ Hom a b)›
                    ‹RTS.Rts (Hom a d)›
                    ‹RTS.Map (Comp a b d · (Comp b c d ⊗ Hom a b))›
          using a b c d 1
                RTS.arrD(3) [of Comp a b d · (Comp b c d ⊗ Hom a b)]
          by auto
        interpret R: simulation
                  ‹RTS.Rts (RTS.dom ((Hom c d ⊗ Hom b c) ⊗ Hom a b))›
                    ‹RTS.Rts (Hom a d)›
                    ‹RTS.Map (Comp a c d · (Hom c d ⊗ Comp a b c) ·
                                a[Hom c d, Hom b c, Hom a b])›
          using a b c d 1
                RTS.arrD(3)
                  [of Comp a c d · (Hom c d ⊗ Comp a b c) ·
                        a[Hom c d, Hom b c, Hom a b]]
          by auto
        show ?thesis
          using x L.extensional R.extensional by simp
      qed
      next
      assume x: cdxbc-x-ab.arr x
```

```
let ?w = RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x
let ?x = RTS.Unpack (Hom c d) (Hom b c) (fst ?w)
let ?y = snd ?w
have fst-x: cd.arr (fst ?x)
  using cdXbc.arr-char cdxbc-X-ab.arr-char x by blast
have snd-x: bc.arr (snd ?x)
  using cdXbc.arr-char cdxbc-X-ab.arr-char x by blast
have snd-w: ab.arr (snd ?w)
  using cdxbc-X-ab.arr-char x by blast
show ?thesis
proof −
  have RTS.Map (Comp a b d · (Comp b c d ⊗ Hom a b)) x =
       RTS.Map (Comp a b d) (RTS.Map (Comp b c d ⊗ Hom a b) x)
    using a b c d RTS.Map-comp by fastforce
  also have ... = RTS.Map (Comp a b d)
                    (RTS.Pack (Hom b d) (Hom a b)
                      (CbcdXIab.map
                      (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x)))
    using RTS.Map-prod a b c d by auto
  also have ... = (λx. fst x ⋆ snd x)
                    (CbcdXIab.map
                      (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x))
    using a b d x CbcdXIab.preserves-reflects-arr
          PU-cdxbc-X-ab.G.preserves-reflects-arr
          Map-Comp-Pack
            [of a b d CbcdXIab.map
                      (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x)]
    by blast
  also have ... = (λx. fst x ⋆ snd x)
                    (RTS.Map (Comp b c d)
                      (fst
                      (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x)),
                  snd (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x))
    using a b x RTS.Map-ide cdxbc-X-ab.arr-char
          CbcdXIab.map-simp
            [of fst (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x)
                snd (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x)]
          PU-cdxbc-X-ab.G.preserves-reflects-arr [of x]
    by auto
  also have ... = (λx. fst x ⋆ snd x)
                    (RTS.Map (Comp b c d)
                      (I (RTS.Rts (RTS.dom (Hom c d ⊗ Hom b c)))
                        (fst
                          (RTS.Unpack
                            (Hom c d ⊗ Hom b c) (Hom a b) x))),
                  snd (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x))
    using x cdxbc-X-ab.arr-char comp-apply
          PU-cdxbc-X-ab.G.preserves-reflects-arr [of x]
    by simp
```

also have ... = $(\lambda x.\ fst\ x \star snd\ x)$
$(RTS.Map\ (Comp\ b\ c\ d)$
$((RTS.Pack\ (Hom\ c\ d)\ (Hom\ b\ c) \circ$
$RTS.Unpack\ (Hom\ c\ d)\ (Hom\ b\ c))$
$(fst$
$(RTS.Unpack$
$(Hom\ c\ d \otimes Hom\ b\ c)\ (Hom\ a\ b)\ x))),$
$snd\ (RTS.Unpack\ (Hom\ c\ d \otimes Hom\ b\ c)\ (Hom\ a\ b)\ x))$
using *PU-cdXbc.inv′* **by** *simp*
also have ... = $(\lambda x.\ fst\ x \star snd\ x)$
$(RTS.Map\ (Comp\ b\ c\ d)$
$(RTS.Pack\ (Hom\ c\ d)\ (Hom\ b\ c)$
$(RTS.Unpack\ (Hom\ c\ d)\ (Hom\ b\ c)$
$(fst$
$(RTS.Unpack$
$(Hom\ c\ d \otimes Hom\ b\ c)\ (Hom\ a\ b)\ x)))),$
$snd\ (RTS.Unpack\ (Hom\ c\ d \otimes Hom\ b\ c)\ (Hom\ a\ b)\ x))$
by *auto*
also have ... = $(\lambda x.\ (fst\ (fst\ x) \star snd\ (fst\ x)) \star snd\ x)$
$((\lambda y.\ (RTS.Unpack\ (Hom\ c\ d)\ (Hom\ b\ c)\ (fst\ y),$
$snd\ y))$
$(RTS.Unpack\ (Hom\ c\ d \otimes Hom\ b\ c)\ (Hom\ a\ b)\ x))$
using *b c d x PU-cdxbc-X-ab.G.preserves-reflects-arr* [*of x*]
cdxbc-X-ab.arr-char
[*of RTS.Unpack* $(Hom\ c\ d \otimes Hom\ b\ c)\ (Hom\ a\ b)\ x$]
Map-Comp-Pack
[*of b c d*
$RTS.Unpack\ (Hom\ c\ d)\ (Hom\ b\ c)$
$(fst\ (RTS.Unpack\ (Hom\ c\ d \otimes Hom\ b\ c)\ (Hom\ a\ b)\ x))$]
by *fastforce*
finally have *LHS*: *RTS.Map*
$(Comp\ a\ b\ d \cdot (Comp\ b\ c\ d \otimes Hom\ a\ b))\ x =$
$(\lambda x.\ (fst\ (fst\ x) \star snd\ (fst\ x)) \star snd\ x)$
$((\lambda y.\ (RTS.Unpack\ (Hom\ c\ d)\ (Hom\ b\ c)\ (fst\ y),$
$snd\ y))$
$(RTS.Unpack\ (Hom\ c\ d \otimes Hom\ b\ c)\ (Hom\ a\ b)\ x))$
by *blast*

have $RTS.Map\ (Comp\ a\ c\ d \cdot (Hom\ c\ d \otimes Comp\ a\ b\ c) \cdot$
$\mathrm{a}[Hom\ c\ d,\ Hom\ b\ c,\ Hom\ a\ b])\ x =$
$RTS.Map\ (Comp\ a\ c\ d)$
$(RTS.Map\ (Hom\ c\ d \otimes Comp\ a\ b\ c)$
$(RTS.Map\ \mathrm{a}[Hom\ c\ d,\ Hom\ b\ c,\ Hom\ a\ b]\ x))$
using *a b c d RTS.Map-comp* **by** *fastforce*
also have ... = $RTS.Map\ (Comp\ a\ c\ d)$
$(RTS.Map\ (Hom\ c\ d \otimes Comp\ a\ b\ c)$
$(RTS.Pack\ (Hom\ c\ d)\ (Hom\ b\ c \otimes Hom\ a\ b)$
$(product\text{-}simulation.map$
$(RTS.Rts\ (Hom\ c\ d))\ bcXab.resid$

```
                    (I (RTS.Rts (Hom c d)))
                    (RTS.Pack (Hom b c) (Hom a b))
                    (ASSOC.map (RTS.Rts (Hom c d))
                               (RTS.Rts (Hom b c))
                               (RTS.Rts (Hom a b))
                      ((product-simulation.map
                          (RTS.Rts (Hom c d ⊗ Hom b c))
                          (RTS.Rts (Hom a b))
                          (RTS.Unpack (Hom c d) (Hom b c))
                          (I (RTS.Rts (Hom a b)))
                          (RTS.Unpack
                            (Hom c d ⊗ Hom b c)
                            (Hom a b) x)))))))
  using a b c d RTS.tensor-agreement RTS.Map-assoc
  by (auto simp add: RTS.comp-arr-ide)
also have ... = RTS.Map (Comp a c d)
                  (RTS.Map (Hom c d ⊗ Comp a b c)
                    (RTS.Pack (Hom c d) (Hom b c ⊗ Hom a b)
                      (product-simulation.map
                        (RTS.Rts (Hom c d)) bcXab.resid
                        (I (RTS.Rts (Hom c d)))
                        (RTS.Pack (Hom b c) (Hom a b))
                        (ASSOC.map (RTS.Rts (Hom c d))
                                   (RTS.Rts (Hom b c))
                                   (RTS.Rts (Hom a b))
                          (((λx. (RTS.Unpack
                                    (Hom c d) (Hom b c) (fst x),
                                  RTS.Map (Hom a b) (snd x)))
                            (RTS.Unpack
                              (Hom c d ⊗ Hom b c)
                              (Hom a b) x)))))))
  using a b c d x RTS.Map-ide
        PU-cdxbc-X-ab.G.preserves-reflects-arr [of x]
        cdxbc-X-ab.arr-char
        Ucdxbc-X-Iab.map-simp
          [of fst (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x)
              snd (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x)]
  by simp
also have ... = RTS.Map (Comp a c d)
                  (RTS.Map (Hom c d ⊗ Comp a b c)
                    (RTS.Pack (Hom c d) (Hom b c ⊗ Hom a b)
                      (product-simulation.map
                        (RTS.Rts (Hom c d)) bcXab.resid
                        (I (RTS.Rts (Hom c d)))
                        (RTS.Pack (Hom b c) (Hom a b))
                        (fst (RTS.Unpack (Hom c d) (Hom b c)
                          (fst
                            (RTS.Unpack
                              (Hom c d ⊗ Hom b c) (Hom a b) x))),
```

```
                    snd (RTS.Unpack (Hom c d) (Hom b c)
                     (fst
                        (RTS.Unpack
                           (Hom c d ⊗ Hom b c) (Hom a b) x))),
                    snd
                     (RTS.Unpack
                        (Hom c d ⊗ Hom b c) (Hom a b) x)))))
proof −
  interpret A: ASSOC ‹RTS.Rts (Hom c d)› ‹RTS.Rts (Hom b c)›
               ‹RTS.Rts (Hom a b)›
    ..
  show ?thesis
    using a b c d x cdxbc-X-ab.arr-char cdXbc.arr-char
          PU-cdxbc-X-ab.G.preserves-reflects-arr
          PU-cdXbc.G.preserves-reflects-arr
          A.map-eq RTS.Map-ide
    by auto
qed
also have ... = RTS.Map (Comp a c d)
                 (RTS.Map (Hom c d ⊗ Comp a b c)
                   (RTS.Pack (Hom c d) (Hom b c ⊗ Hom a b)
                     (fst ?x,
                      RTS.Pack (Hom b c) (Hom a b)
                       (snd ?x, snd ?w))))
   using c d x RTS.Map-ide Icd-X-Pbcxab.map-simp
         cdxbc-X-ab.arr-char
         PU-cdxbc-X-ab.G.preserves-reflects-arr [of x]
         PU-cdXbc.G.preserves-reflects-arr
          [of fst (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x)]
   by auto
also have ... = RTS.Map (Comp a c d)
                 (RTS.Pack (Hom c d) (Hom a c)
                    (IcdXCabc.map
                       ((RTS.Unpack
                           (Hom c d) (Hom b c ⊗ Hom a b) ∘
                           RTS.Pack (Hom c d) (Hom b c ⊗ Hom a b))
                         (fst ?x,
                          RTS.Pack (Hom b c) (Hom a b)
                           (snd ?x, snd ?w)))))
  using a b c d x RTS.Map-prod by auto
also have ... = RTS.Map (Comp a c d)
                 (RTS.Pack (Hom c d) (Hom a c)
                    (IcdXCabc.map
                       (fst ?x,
                        RTS.Pack (Hom b c) (Hom a b)
                         (snd ?x, snd ?w))))
proof −
  have cd-X-bcxab.arr
        (fst ?x, RTS.Pack (Hom b c) (Hom a b) (snd ?x, snd ?w))
```

```
      using fst-x snd-x snd-w PU-bcXab.F.preserves-reflects-arr
      by fastforce
    thus ?thesis
      using PU-cd-X-bcxab.inv by auto
  qed
  also have ... = RTS.Map (Comp a c d)
                    (RTS.Pack (Hom c d) (Hom a c)
                      (RTS.Map (Hom c d) (fst ?x),
                       RTS.Map (Comp a b c)
                         (RTS.Pack (Hom b c) (Hom a b)
                           (snd ?x, snd ?w))))
    using a b c d x fst-x snd-x snd-w
          IcdXCabc.map-simp
          [of fst ?x
              RTS.Pack (Hom b c) (Hom a b) (snd ?x, snd ?w)]
    by fastforce
  also have ... = fst ?x ⋆ (snd ?x ⋆ snd ?w)
  proof −
    have ac.arr (snd ?x ⋆ snd ?w)
      using a b c snd-w snd-x HOM-arr-char arr-coincidence
      by auto
    thus ?thesis
      using a b c d fst-x RTS.Map-ide snd-x snd-w Map-Comp-Pack
      by auto
  qed
  also have ... = (λx. (fst (fst x) ⋆ snd (fst x)) ⋆ snd x)
                    ((λy. (RTS.Unpack (Hom c d) (Hom b c) (fst y),
                           snd y))
                      (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x))
    using H.comp-assoc by simp
  finally have RHS: RTS.Map
                      (Comp a c d · (Hom c d ⊗ Comp a b c) ·
                        a[Hom c d, Hom b c, Hom a b]) x =
                    (λx. (fst (fst x) ⋆ snd (fst x)) ⋆ snd x)
                      ((λy. (RTS.Unpack (Hom c d) (Hom b c) (fst y),
                             snd y))
                      (RTS.Unpack (Hom c d ⊗ Hom b c) (Hom a b) x))
    by blast
  show ?thesis using LHS RHS by auto
 qed
qed
qed
qed
qed
qed

proposition is-rts-enriched-category:
shows rts-enriched-category Obj Hom Id Comp
  ..
```

```
  lemma HOM-agreement:
  assumes H.ide a and H.ide b
  shows HOM_EC a b = HOM a b
    using assms Hom-def RTS.bij-mkide(3) by auto

end
```

5.3.1 Functoriality

If we are to construct an enriched functor from a given RTS-functor F, then we need a base category **RTS** that is large enough to provide objects for all the required hom-RTS's. So the arrow type of this category will need to embed the arrow types of both the domain A and the codomain B RTS of the given RTS-functor F. Here I have assumed that both of these arrow types are in fact the same type $'A$ and in addition that $'A$ is a universe, so that it supports the construction of the cartesian closed base category **RTS**. At the cost of having to deal with coercions, we could more generally just assume injections from the arrow types of A and B into a common universe $'C$, but we haven't bothered to do that.

```
  locale enriched-functor-of-rts-functor =
    universe arr-type +
    RTS: rtscat arr-type +
    A: locally-small-rts-category resid_A comp_A +
    B: locally-small-rts-category resid_B comp_B +
    F: rts-functor resid_A comp_A resid_B comp_B F
  for arr-type :: 'A itself
  and resid_A :: 'A resid  (infix \_A 70)
  and comp_A :: 'A comp    (infixr ⋆_A 53)
  and resid_B :: 'A resid  (infix \_B 70)
  and comp_B :: 'A comp    (infixr ⋆_B 53)
  and F :: 'A ⇒ 'A
  begin

    interpretation A: enriched-category-of-rts-category arr-type resid_A comp_A ..
    interpretation B: enriched-category-of-rts-category arr-type resid_B comp_B ..



    definition F_o
    where F_o a ≡ if A.H.ide a then F a else B.null

    definition F_a
    where F_a a b ≡ if A.H.ide a ∧ A.H.ide b
                    then RTS.mkarr (A.HOM_EC a b) (B.HOM_EC (F_o a) (F_o b))
                         (λt. if residuation.arr (A.HOM_EC a b) t
                              then F t
                              else ResiduatedTransitionSystem.partial-magma.null
```

```
                                (B.HOM_EC (F_o a) (F_o b)))
            else RTS.null

lemma sub-rts-resid-eq:
assumes a ∈ A.Obj and b ∈ A.Obj
shows sub-rts.resid resid_A (λt. A.H.in-hom t a b) = A.HOM_EC a b
and sub-rts.resid resid_B (λt. B.H.in-hom t (F_o a) (F_o b)) =
    B.HOM_EC (F_o a) (F_o b)
proof −
  have 1: ⋀a. a ∈ Collect A.H.ide ⟹ F_o a ∈ Collect B.H.ide
    unfolding F_o-def by simp
  interpret DOM: sub-rts resid_A ‹λt. A.H.in-hom t a b›
    using assms A.sub-rts-HOM by metis
  interpret COD: sub-rts resid_B ‹λt. B.H.in-hom t (F_o a) (F_o b)›
    using assms 1 B.sub-rts-HOM by metis
  show DOM.resid = A.HOM_EC a b
    using assms DOM.resid-def A.Hom-def RTS.bij-mkide(3) by simp
  show COD.resid = B.HOM_EC (F_o a) (F_o b)
    using assms 1 B.Hom-def COD.resid-def RTS.bij-mkide(3) by auto
qed

sublocale rts-enriched-functor
          ‹Collect A.H.ide› A.Hom A.Id A.Comp
          ‹Collect B.H.ide› B.Hom B.Id B.Comp
          F_o F_a
proof
  show 1: ⋀a. a ∈ Collect A.H.ide ⟹ F_o a ∈ Collect B.H.ide
    unfolding F_o-def by simp
 show ⋀a b. a ∉ Collect A.H.ide ∨ b ∉ Collect A.H.ide ⟹ F_a a b = RTS.null
    unfolding F_a-def by auto
  show 2: ⋀a b. ⟦a ∈ Collect A.H.ide; b ∈ Collect A.H.ide⟧ ⟹
                «F_a a b : A.Hom a b → B.Hom (F_o a) (F_o b)»
  proof −
    fix a b
    assume a: a ∈ Collect A.H.ide and b: b ∈ Collect A.H.ide
    interpret DOM: sub-rts resid_A ‹λt. A.H.in-hom t a b›
      using a b A.sub-rts-HOM by metis
    interpret COD: sub-rts resid_B ‹λt. B.H.in-hom t (F_o a) (F_o b)›
      using a b 1 B.sub-rts-HOM by metis
    interpret F_a-ab: simulation DOM.resid COD.resid
                      ‹λt. if DOM.arr t then F t else COD.null›
    proof
      show ⋀t. ¬ DOM.arr t ⟹
               (if DOM.arr t then F t else COD.null) = COD.null
        by simp
      show ⋀t u. DOM.con t u ⟹
                 COD.con (if DOM.arr t then F t else COD.null)
                         (if DOM.arr u then F u else COD.null)
        using COD.con-char DOM.arr-char DOM.con-char F_o-def a b by auto
```

```
    show ⋀t u. DOM.con t u ⟹
                  (if DOM.arr (DOM.resid t u) then F (DOM.resid t u)
                   else COD.null) =
                  COD.resid (if DOM.arr t then F t else COD.null)
                            (if DOM.arr u then F u else COD.null)
      using a b 1 F_o-def COD.con-char DOM.arr-char DOM.con-char
            DOM.resid-def COD.resid-def DOM.resid-closed
      by auto
  qed
  show «F_a a b : A.Hom a b → B.Hom (F_o a) (F_o b)»
  proof
    have 2: residuation.arr (A.HOM_EC a b) = DOM.arr
      using a b DOM.arr-char A.Hom-def RTS.bij-mkide(3) by simp
    have 3: ResiduatedTransitionSystem.partial-magma.null
              (B.HOM_EC (F_o a) (F_o b)) =
            COD.null
      using a b 1 COD.null-char B.Hom-def RTS.bij-mkide(3) by auto
    show 4: RTS.arr (F_a a b)
    proof (intro RTS.arrI_RTSC)
      show extensional-rts DOM.resid ∧ small-rts DOM.resid
        using a b A.V.extensional-rts-axioms DOM.preserves-extensional-rts
              sub-rts-resid-eq
        by auto
      show extensional-rts COD.resid ∧ small-rts COD.resid
        using a b 1 B.V.extensional-rts-axioms COD.preserves-extensional-rts
              sub-rts-resid-eq(2)
        by auto
      have (λt. if DOM.arr t then F t
                else ResiduatedTransitionSystem.partial-magma.null
                       (B.HOM_EC (F_o a) (F_o b))) =
           (λt. if DOM.arr t then F t else COD.null)
        using a b 3 sub-rts-resid-eq F_a-def RTS.Map-mkarr
        by auto
      thus F_a a b ∈ RTS.mkarr DOM.resid COD.resid ‘
                       Collect (simulation DOM.resid COD.resid)
        unfolding F_a-def
        using a b 1 3 sub-rts-resid-eq F_a-ab.simulation-axioms
              RTS.Map-mkarr
        by auto
    qed
    show RTS.dom (F_a a b) = A.Hom a b
      using 4 F_a-def by fastforce
    show RTS.cod (F_a a b) = B.Hom (F_o a) (F_o b)
      using 4 F_o-def F_a-def by fastforce
  qed
qed
show ⋀a. a ∈ Collect A.H.ide ⟹ F_a a a · A.Id a = B.Id (F_o a)
proof −
  fix a
```

assume *a*: $a \in Collect\ A.H.ide$
interpret HOM_A: *sub-rts* $resid_A$ ‹$\lambda t.\ A.H.in\text{-}hom\ t\ a\ a$›
using *a A.sub-rts-HOM* **by** *metis*
interpret HOM_B: *sub-rts* $resid_B$ ‹$\lambda t.\ B.H.in\text{-}hom\ t\ (F_o\ a)\ (F_o\ a)$›
using *a 1 B.sub-rts-HOM* **by** *metis*
have *3*: $RTS.arr\ (F_a\ a\ a \cdot A.Id\ a)$
using *a 2 A.arr-coincidence A.Id-in-hom* **by** *auto*
have *4*: $F_a\ a\ a \cdot A.Id\ a =$
$RTS.mkarr\ (A.HOM_{EC}\ a\ a)\ (B.HOM_{EC}\ (F\ a)\ (F\ a))$
$(\lambda t.\ if\ HOM_A.arr\ t\ then\ F\ t\ else\ HOM_B.null) \cdot$
$RTS.mkarr\ RTS.One.resid\ (A.HOM_{EC}\ a\ a)$
$(\lambda t.\ if\ RTS.One.arr\ t\ then\ a\ else\ HOM_A.null)$
proof −
have $A.HOM_{EC}\ a\ a = HOM_A.resid$
using *a* HOM_A*.resid-def A.Hom-def RTS.bij-mkide*(*3*) **by** *simp*
moreover have $B.HOM_{EC}\ (F_o\ a)\ (F_o\ a) = HOM_B.resid$
using *a 1* HOM_B*.resid-def B.Hom-def* [*of* F_o *a* F_o *a*] *RTS.bij-mkide*(*3*)
by *simp*
ultimately show *?thesis*
unfolding F_o*-def* F_a*-def A.Id-def B.Id-def*
using *a A.arr-coincidence A.Hom-def B.Hom-def*
apply *auto*[*1*]
using F_o*-def* ‹$B.HOM_{EC}\ (F_o\ a)\ (F_o\ a) = HOM_B.resid$› **by** *presburger*
qed
also have ... = $RTS.mkarr\ RTS.One.resid\ (B.HOM_{EC}\ (F\ a)\ (F\ a))$
$((\lambda t.\ if\ HOM_A.arr\ t\ then\ F\ t\ else\ HOM_B.null) \circ$
$(\lambda t.\ if\ RTS.One.arr\ t\ then\ a\ else\ HOM_A.null))$
using *a 1 3 4 RTS.comp-mkarr* **by** *auto*
also have ... = $RTS.mkarr\ RTS.One.resid\ (B.HOM_{EC}\ (F\ a)\ (F\ a))$
$(\lambda t.\ if\ RTS.One.arr\ t\ then\ F\ a\ else\ HOM_B.null)$
(**is** *?LHS* = *?RHS*)
proof (*intro RTS.arr-eqI*)
interpret *HOM-Fa-Fa*: *hom-rts arr-type B.Obj B.Hom B.Id B.Comp*
‹$F\ a$› ‹$F\ a$›
using *a* **by** *unfold-locales auto*
have *5*: *simulation* $RTS.One.resid\ (B.HOM_{EC}\ (F\ a)\ (F\ a))$
$(\lambda t.\ if\ RTS.One.arr\ t\ then\ F\ a\ else\ HOM_B.null)$
proof −
have $(\lambda t.\ if\ RTS.One.arr\ t\ then\ F\ a$
else ResiduatedTransitionSystem.partial-magma.null
$(B.HOM\ (F\ a)\ (F\ a))) =$
$(\lambda t.\ if\ RTS.One.arr\ t\ then\ F\ a\ else\ HOM_B.null)$
using F_o*-def a* **by** *fastforce*
thus *?thesis*
using *a RTS.bij-mkide*(*3*) *B.Id-in-hom* [*of F a*]
RTS.arrD(*3*) [*of B.Id* (*F a*)]
B.Id-def RTS.Map-mkarr
by *auto*
qed

```
    have 6: extensional-rts (B.HOM_EC (F a) (F a))
      using a by force
    have 7: small-rts (B.HOM_EC (F a) (F a))
      using a by force
    have 8: RTS.arr ?LHS
      using 3 calculation by auto
    have 9: RTS.arr ?RHS
      using 5 6 7 RTS.One.extensional-rts-axioms
            RTS.One.small-rts-axioms
      by auto
    show RTS.par ?LHS ?RHS
      using 8 9 by auto
    show RTS.Map ?LHS = RTS.Map ?RHS
    proof
      fix x
      show RTS.Map ?LHS x = RTS.Map ?RHS x
        using a 8 9 RTS.Map-mkarr HOM_A.arr-char HOM_A.null-char
              A.H.ide-in-hom
        by auto
    qed
  qed
  also have ... = B.Id (F_o a)
    unfolding F_o-def B.Id-def
    using a by auto
  finally show F_a a a · A.Id a = B.Id (F_o a) by blast
qed
show ⋀a b c. ⟦a ∈ A.Obj; b ∈ A.Obj; c ∈ A.Obj⟧
               ⟹ B.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b) =
                  F_a a c · A.Comp a b c
proof −
  fix a b c
  assume a: a ∈ Collect A.H.ide and b: b ∈ Collect A.H.ide
  and c: c ∈ Collect A.H.ide
  interpret HOM_A-ab: sub-rts resid_A ‹λt. A.H.in-hom t a b›
    using a b A.sub-rts-HOM by metis
  interpret HOM_A-ac: sub-rts resid_A ‹λt. A.H.in-hom t a c›
    using a c A.sub-rts-HOM by metis
  interpret HOM_A-bc: sub-rts resid_A ‹λt. A.H.in-hom t b c›
    using b c A.sub-rts-HOM by metis
  interpret HOM_B-ab: sub-rts resid_B ‹λt. B.H.in-hom t (F_o a) (F_o b)›
    using a b 1 B.sub-rts-HOM by metis
  interpret HOM_B-ac: sub-rts resid_B ‹λt. B.H.in-hom t (F_o a) (F_o c)›
    using a c 1 B.sub-rts-HOM by metis
  interpret HOM_B-bc: sub-rts resid_B ‹λt. B.H.in-hom t (F_o b) (F_o c)›
    using b c 1 B.sub-rts-HOM by metis
  interpret F_a-bc: simulation HOM_A-bc.resid HOM_B-bc.resid
                    ‹RTS.Map (F_a b c)›
    using b c 1 2 [of b c] A.Hom-def B.Hom-def RTS.bij-mkide(3)
          RTS.arrD(3) [of F_a b c]
```

by *auto*
interpret F_a*-ab*: *simulation* HOM_A*-ab.resid* HOM_B*-ab.resid*
‹*RTS.Map* (F_a *a b*)›
using *a b 1 2* [*of a b*] *A.Hom-def B.Hom-def RTS.bij-mkide*(*3*)
RTS.arrD(*3*) [*of* F_a *a b*]
by *auto*
interpret F_a*-bc-x-*F_a*-ab*: *product-simulation*
HOM_A*-bc.resid* HOM_A*-ab.resid*
HOM_B*-bc.resid* HOM_B*-ab.resid*
‹*RTS.Map* (F_a *b c*)› ‹*RTS.Map* (F_a *a b*)›
..

interpret *HOM-bc*: *extensional-rts* ‹$A.HOM_{EC}$ *b c*›
using *b c* **by** *simp*
interpret *HOM-ab*: *extensional-rts* ‹$A.HOM_{EC}$ *a b*›
using *a b* **by** *simp*
interpret *HOM-ac*: *extensional-rts* ‹$A.HOM_{EC}$ *a c*›
using *a c* **by** *simp*
interpret *HOM-bc-x-HOM-ab*: *product-rts*
‹$A.HOM_{EC}$ *b c*› ‹$A.HOM_{EC}$ *a b*›
..
interpret *B-HOM-bc*: *extensional-rts* ‹$B.HOM_{EC}$ (F_o *b*) (F_o *c*)›
using *b c 1* **by** *simp*
interpret *B-HOM-ab*: *extensional-rts* ‹$B.HOM_{EC}$ (F_o *a*) (F_o *b*)›
using *a b 1* **by** *simp*
interpret *B-HOM-bc-x-B-HOM-ab*: *product-rts*
‹$B.HOM_{EC}$ (F_o *b*) (F_o *c*)›
‹$B.HOM_{EC}$ (F_o *a*) (F_o *b*)›
..
interpret *U*: *simulation*
‹*RTS.Rts* (*A.Hom b c* ⊗ *A.Hom a b*)›
‹*HOM-bc-x-HOM-ab.resid*›
‹*RTS.Unpack* (*A.Hom b c*) (*A.Hom a b*)›
using *a b c RTS.simulation-Unpack* **by** *simp*

show *B.Comp* (F_o *a*) (F_o *b*) (F_o *c*) · (F_a *b c* ⊗ F_a *a b*) =
F_a *a c* · *A.Comp a b c*
proof (*intro RTS.arr-eqI*)
show *3*: *RTS.par*
(*B.Comp* (F_o *a*) (F_o *b*) (F_o *c*) · (F_a *b c* ⊗ F_a *a b*))
(F_a *a c* · *A.Comp a b c*)
proof (*intro conjI*)
show *RTS.seq* (*B.Comp* (F_o *a*) (F_o *b*) (F_o *c*)) (F_a *b c* ⊗ F_a *a b*)
using *a b c 1 2* [*of b c*] *2* [*of a b*]
RTS.prod-simps [*of* F_a *b c* F_a *a b*]
apply (*intro RTS.seqI*)
apply *auto*[*1*]
by *fastforce*+
show *RTS.seq* (F_a *a c*) (*A.Comp a b c*)

```
    using a b c 1 2 A.Comp-in-hom by blast
  show RTS.dom (B.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) =
       RTS.dom (F_a a c · A.Comp a b c)
    using a b c 1 2 [of b c] 2 [of a b] 2 [of a c] by fastforce
  show RTS.cod (B.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) =
       RTS.cod (F_a a c · A.Comp a b c)
    using a b c 1 2 [of b c] 2 [of a b] 2 [of a c] by fastforce
qed
show RTS.Map (B.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) =
     RTS.Map (F_a a c · A.Comp a b c)
proof −
  have RTS.Map (B.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) =
       RTS.Map (B.Comp (F_o a) (F_o b) (F_o c)) ∘
               RTS.Map (F_a b c ⊗ F_a a b)
    using a b c 1 2 [of b c] 2 [of a b] 2 [of a c] 3 RTS.Map-comp
    by auto
  also have ... = (λt. fst
                       (RTS.Unpack
                          (B.Hom (F_o b) (F_o c))
                          (B.Hom (F_o a) (F_o b)) t) ⋆_B
                       snd
                         (RTS.Unpack
                            (B.Hom (F_o b) (F_o c))
                            (B.Hom (F_o a) (F_o b)) t)) ∘
                  (RTS.Pack (RTS.cod (F_a b c)) (RTS.cod (F_a a b)) ∘
                     F_a-bc-x-F_a-ab.map ∘
                       RTS.Unpack
                         (RTS.dom (F_a b c)) (RTS.dom (F_a a b)))
  proof −
    have RTS.Map (B.Comp (F_o a) (F_o b) (F_o c)) =
         (λt. fst (RTS.Unpack
                     (B.Hom (F_o b) (F_o c)) (B.Hom (F_o a) (F_o b)) t) ⋆_B
             snd (RTS.Unpack
                     (B.Hom (F_o b) (F_o c)) (B.Hom (F_o a) (F_o b)) t))
      using a b c 1 2 B.Comp-def RTS.Map-mkarr
            B.Comp-in-hom [of F_o a F_o b F_o c]
      by auto
    moreover have RTS.Map (F_a b c ⊗ F_a a b) =
                  RTS.Pack (RTS.cod (F_a b c)) (RTS.cod (F_a a b)) ∘
                    F_a-bc-x-F_a-ab.map ∘
                      RTS.Unpack (RTS.dom (F_a b c)) (RTS.dom (F_a a b))
    proof −
      have RTS.Rts (RTS.dom (F_a b c)) = HOM_A-bc.resid
        using b c 2 [of b c] sub-rts-resid-eq by auto
      moreover have RTS.Rts (RTS.dom (F_a a b)) = HOM_A-ab.resid
        using a b 2 [of a b] sub-rts-resid-eq by auto
      ultimately have product-simulation.map
           (RTS.Rts (RTS.dom (F_a b c))) (RTS.Rts (RTS.dom (F_a a b)))
            (RTS.Map (F_a b c)) (RTS.Map (F_a a b)) =
```

```
        F_a-bc-x-F_a-ab.map
    using a b c 1 2 [of b c] 2 [of a b] F_a-def RTS.dom-mkarr by auto
  thus ?thesis
    using a b c 1 2 [of b c] 2 [of a b] A.Hom-def B.Hom-def
        RTS.in-homE RTS.Map-prod [of F_a b c F_a a b]
    by auto
 qed
 ultimately show ?thesis by force
qed
also have ... = (λt. fst
                    (RTS.Unpack
                      (B.Hom (F_o b) (F_o c)) (B.Hom (F_o a) (F_o b))
                       (RTS.Pack
                          (B.Hom (F_o b) (F_o c))
                          (B.Hom (F_o a) (F_o b))
                           (F_a-bc-x-F_a-ab.map
                              (RTS.Unpack
                                 (A.Hom b c) (A.Hom a b) t)))) ⋆_B
                    snd
                      (RTS.Unpack
                         (B.Hom (F_o b) (F_o c)) (B.Hom (F_o a) (F_o b))
                          (RTS.Pack
                             (B.Hom (F_o b) (F_o c))
                             (B.Hom (F_o a) (F_o b))
                             (F_a-bc-x-F_a-ab.map
                                (RTS.Unpack
                                   (A.Hom b c) (A.Hom a b) t)))))
 using a b c 2 [of a b] 2 [of b c] by fastforce
also have ... = (λt. fst (F_a-bc-x-F_a-ab.map
                          (RTS.Unpack (A.Hom b c) (A.Hom a b) t)) ⋆_B
                  snd (F_a-bc-x-F_a-ab.map
                          (RTS.Unpack (A.Hom b c) (A.Hom a b) t)))
proof
 fix t
 interpret PU: inverse-simulations
              ‹RTS.Rts
                 (RTS.dom
                    (B.Hom (F_o b) (F_o c) ⊗ B.Hom (F_o a) (F_o b)))›
              ‹product-rts.resid
               (B.HOM_EC (F_o b) (F_o c)) (B.HOM_EC (F_o a) (F_o b))›
              ‹RTS.Pack
                 (B.Hom (F_o b) (F_o c)) (B.Hom (F_o a) (F_o b))›
              ‹RTS.Unpack
                  (B.Hom (F_o b) (F_o c)) (B.Hom (F_o a) (F_o b))›
   using a b c 1 RTS.inverse-simulations-Pack-Unpack by simp
 show fst
       (RTS.Unpack
          (B.Hom (F_o b) (F_o c)) (B.Hom (F_o a) (F_o b))
          (RTS.Pack (B.Hom (F_o b) (F_o c)) (B.Hom (F_o a) (F_o b))
```

```
                    (F_a-bc-x-F_a-ab.map
                      (RTS.Unpack (A.Hom b c) (A.Hom a b) t))) ⋆_B
                     snd (RTS.Unpack
                           (B.Hom (F_o b) (F_o c)) (B.Hom (F_o a) (F_o b))
                           (RTS.Pack
                             (B.Hom (F_o b) (F_o c)) (B.Hom (F_o a) (F_o b))
                             (F_a-bc-x-F_a-ab.map
                               (RTS.Unpack (A.Hom b c) (A.Hom a b) t)))) =
          fst
            (F_a-bc-x-F_a-ab.map (RTS.Unpack (A.Hom b c) (A.Hom a b) t))
⋆_B
          snd
            (F_a-bc-x-F_a-ab.map (RTS.Unpack (A.Hom b c) (A.Hom a b) t))
        proof (cases U.A.arr t)
          show ¬ U.A.arr t ⟹ ?thesis
            using a b c sub-rts-resid-eq
                  F_a-bc-x-F_a-ab.extensional PU.F.extensional PU.G.extensional
                  U.extensional F_a-bc.extensional F_a-ab.extensional
                  PU.A.not-arr-null B-HOM-bc-x-B-HOM-ab.not-arr-null
            by auto
          assume t: U.A.arr t
          show ?thesis
           using a b c t 1 RTS.Unpack-Pack F_a-bc-x-F_a-ab.preserves-reflects-arr
                 U.preserves-reflects-arr sub-rts-resid-eq
            by auto
        qed
      qed
      also have ... =
              (λt. F (fst (RTS.Unpack (A.Hom b c) (A.Hom a b) t)) ⋆_B
                  F (snd (RTS.Unpack (A.Hom b c) (A.Hom a b) t)))
      proof
        fix t
        show fst
              (F_a-bc-x-F_a-ab.map
                (RTS.Unpack (A.Hom b c) (A.Hom a b) t)) ⋆_B
             snd
              (F_a-bc-x-F_a-ab.map
                (RTS.Unpack (A.Hom b c) (A.Hom a b) t)) =
             F (fst (RTS.Unpack (A.Hom b c) (A.Hom a b) t)) ⋆_B
             F (snd (RTS.Unpack (A.Hom b c) (A.Hom a b) t))
        proof (cases U.A.arr t)
          show ¬ U.A.arr t ⟹ ?thesis
            using a b c U.extensional F_a-bc-x-F_a-ab.extensional F.extensional
                  HOM_A-bc.null-char HOM_A-ab.null-char HOM_B-bc.null-char
                  HOM_B-ab.null-char F_a-bc.extensional F_a-ab.extensional
                  sub-rts-resid-eq HOM_A-bc.not-arr-null HOM_A-ab.not-arr-null
            by auto
          assume t: U.A.arr t
          show ?thesis
```

```
      using a b c t 1 F_a-bc-x-F_a-ab.map-def U.preserves-reflects-arr
            sub-rts-resid-eq B-HOM-ab.extensional-rts-axioms
            B-HOM-bc.extensional-rts-axioms F_a-def
            HOM-ab.extensional-rts-axioms HOM-bc.extensional-rts-axioms
            RTS.bij-mkarr(3)
      by auto
  qed
 qed
also have ... = (λt. F (fst (RTS.Unpack (A.Hom b c) (A.Hom a b) t) ⋆_A
                         snd (RTS.Unpack (A.Hom b c) (A.Hom a b) t)))
 proof
  fix t
  show F (fst (RTS.Unpack (A.Hom b c) (A.Hom a b) t)) ⋆_B
       F (snd (RTS.Unpack (A.Hom b c) (A.Hom a b) t)) =
       F (fst (RTS.Unpack (A.Hom b c) (A.Hom a b) t) ⋆_A
          snd (RTS.Unpack (A.Hom b c) (A.Hom a b) t))
  proof (cases U.A.arr t)
    show ¬ U.A.arr t ⟹ ?thesis
      using a b c U.extensional F.extensional A.HOM-null-char
            A.H.extensional B.H.extensional
            A.H.null-is-zero(2) A.null-coincidence
            B.H.null-is-zero(2) B.null-coincidence
      by auto
    assume t: U.A.arr t
    have A.H.seq (fst (RTS.Unpack (A.Hom b c) (A.Hom a b) t))
                 (snd (RTS.Unpack (A.Hom b c) (A.Hom a b) t))
     using A.HOM-arr-char HOM-bc-x-HOM-ab.arr-char a b c t by blast
    thus ?thesis
      using a b c t U.preserves-reflects-arr F.preserves-comp
      by auto
  qed
 qed
 also have ... = (λt. if HOM-ac.arr t then F t
                      else ResiduatedTransitionSystem.partial-magma.null
                             (B.HOM_EC (F_o a) (F_o c))) ∘
                 (λt. fst (RTS.Unpack (A.Hom b c) (A.Hom a b) t) ⋆_A
                    snd (RTS.Unpack (A.Hom b c) (A.Hom a b) t))
 proof
  fix t
  show F (fst (RTS.Unpack (A.Hom b c) (A.Hom a b) t) ⋆_A
          snd (RTS.Unpack (A.Hom b c) (A.Hom a b) t)) =
       ((λt. if HOM-ac.arr t
             then F t
             else ResiduatedTransitionSystem.partial-magma.null
                    (B.HOM_EC (F_o a) (F_o c))) ∘
        (λt. fst (RTS.Unpack (A.Hom b c) (A.Hom a b) t) ⋆_A
             snd (RTS.Unpack (A.Hom b c) (A.Hom a b) t)))
        t
  proof (cases U.A.arr t)
```

```
        show ¬ U.A.arr t ⟹ ?thesis
          using a b c U.extensional F.extensional A.HOM-null-char
                A.H.null-is-zero(2) A.null-coincidence sub-rts-resid-eq
                HOM_B-ac.null-char
          by auto
        assume t: U.A.arr t
        have HOM-ac.arr (fst (RTS.Unpack (A.Hom b c) (A.Hom a b) t) ⋆_A
                         snd (RTS.Unpack (A.Hom b c) (A.Hom a b) t))
      by (meson A.H.comp-in-homI A.HOM-arr-char HOM-bc-x-HOM-ab.arr-char
            U.preserves-reflects-arr a b c t)
        thus ?thesis
          using a b c t sub-rts-resid-eq F.preserves-reflects-arr
          by auto
      qed
    qed
    also have ... = RTS.Map (F_a a c) ∘ RTS.Map (A.Comp a b c)
      using a b c F_a-def [of a c] A.Comp-def [of a b c] sub-rts-resid-eq
      apply simp
      using 1 RTS.bij-mkarr(3) by force
    also have ... = RTS.Map (F_a a c · A.Comp a b c)
      using RTS.Map-comp
      by (simp add: ‹RTS.par
                      (B.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b))
                      (F_a a c · A.Comp a b c)›)
    finally show ?thesis by blast
  qed
 qed
qed

end
```

5.4 Equivalence of RTS-Enriched Categories and RTS-Categories

We now extend to an equivalence the correspondence between categories enriched in **RTS** and RTS-categories.

5.4.1 RTS-Category to Enriched Category to RTS-Category

```
context enriched-category-of-rts-category
begin

  interpretation RC: rts-category-of-enriched-category arr-type
                   Obj Hom Id Comp ..

  no-notation RTS.prod     (infixr ⊗ 51)
```

interpretation *Trn*: *simulation RC.resid resid*
‹λt. if RC.arr t then RC.Trn t else null›
proof
let *?Trn = λt. if RC.arr t then RC.Trn t else null*
show *⋀t. ¬ RC.arr t ⟹ ?Trn t = null*
by *simp*
fix *t u*
assume *tu*: *RC.V.con t u*
interpret *Hom*: *sub-rts resid ‹λv. H.in-hom v (RC.Dom u) (RC.Cod u)›*
using *sub-rts-HOM* **by** *auto*
interpret *HOM*: *hom-rts arr-type Obj Hom Id Comp*
‹RC.Dom u› ‹RC.Cod u›
using *tu RC.con-char RC.arr-char RC.V.con-implies-arr*
by *unfold-locales blast+*
show *con*: *?Trn t ⌢ ?Trn u*
proof −
have *HOM.con (RC.Trn t) (RC.Trn u) ⟷*
Hom.con (RC.Trn t) (RC.Trn u)
using *tu Hom.con-char Hom-def RC.con-char RTS.bij-mkide(3)* **by** *auto*
thus *?thesis*
using *tu Hom.con-char RC.con-char RC.Con-def* **by** *auto*
qed
show *?Trn (RC.resid t u) = ?Trn t \ ?Trn u*
proof −
have *HOM*$_{EC}$ *(RC.Dom t) (RC.Cod t) (RC.Trn t) (RC.Trn u) =*
RC.Trn t \ RC.Trn u
using *tu con RC.con-char Hom.con-char Hom-def Hom.resid-def RC.arr-char*
RTS.bij-mkide(3)
by *auto*
thus *?thesis*
using *tu con* **by** *auto*
qed
qed

interpretation *MkArr*: *simulation resid RC.resid*
‹λt. if arr t then RC.MkArr (H.dom t) (H.cod t) t
else RC.null›
proof
let *?MkArr = λt. if arr t then RC.MkArr (H.dom t) (H.cod t) t else RC.null*
show *⋀t. ¬ arr t ⟹ ?MkArr t = RC.null*
by *simp*
fix *t u*
assume *tu*: *t ⌢ u*
interpret *Hom*: *sub-rts resid ‹λt. H.in-hom t (H.dom u) (H.cod u)›*
using *tu sub-rts-HOM* [*of H.dom u H.cod u*] *arr-coincidence V.con-implies-arr*
by *auto*
show *con*: *RC.V.con (?MkArr t) (?MkArr u)*
using *tu V.con-implies-arr arr-coincidence RC.con-char HOM-arr-char*
con-implies-hpar Hom.con-char H.in-homI HOM-agreement

```
    by (unfold RC.con-char) auto
  show ?MkArr (t \ u) = RC.resid (?MkArr t) (?MkArr u)
    using tu con arr-coincidence con-implies-hpar RC.con-char HOM-arr-char
      con-implies-hpar Hom.con-char H.in-homI HOM-agreement Hom.resid-def
    apply auto[1]
     apply (metis VV.F.preserves-trg dom-trg V.trg-def)
    by (metis VV.G.preserves-trg cod-trg V.trg-def)
qed

interpretation Trn-MkArr: inverse-simulations resid RC.resid
                          ‹λt. if RC.arr t then RC.Trn t else null›
                          ‹λt. if arr t then RC.MkArr (H.dom t) (H.cod t) t
                               else RC.null›
proof
  let ?Trn = λt. if RC.arr t then RC.Trn t else null
 let ?MkArr = λt. if arr t then RC.MkArr (H.dom t) (H.cod t) t else RC.null
  show ?MkArr ∘ ?Trn = I RC.resid
  proof
    fix t
    show (?MkArr ∘ ?Trn) t = I RC.resid t
     apply auto[1]
     by (metis RC.Cod.simps(1) RC.Dom.simps(1) RC.Trn.simps(1)
         RC.arr.simps(2) RC.arr-char RC.arr-eqI RC.null-char H.in-homE
         HOM-arr-char)
  qed
  show ?Trn ∘ ?MkArr = I resid by auto
qed

lemma inverse-simulations-Trn-MkArr:
shows inverse-simulations resid RC.resid
        (λt. if RC.arr t then RC.Trn t else null)
        (λt. if arr t then RC.MkArr (H.dom t) (H.cod t) t else RC.null)
  ..

interpretation Trn: functor RC.hcomp hcomp
                    ‹λt. if RC.arr t then RC.Trn t else null›
proof
  let ?Trn = λt. if RC.arr t then RC.Trn t else null
  show ⋀f. ¬ RC.H.arr f ⟹ ?Trn f = H.null
    using null-coincidence RC.arr-coincidence by auto
  show 1: ⋀f. RC.H.arr f ⟹ H.arr (?Trn f)
    using RC.arr-coincidence arr-coincidence null-coincidence Trn.extensional
    by (metis Trn.preserves-reflects-arr)
  show ⋀f. RC.H.arr f ⟹ H.dom (?Trn f) = ?Trn (RC.H.dom f)
  proof −
    fix t
    assume t: RC.H.arr t
    have 2: RC.arr (RC.MkArr (RC.Dom t) (RC.Dom t) (RC.Dom t))
      using t RC.arr-coincidence RC.arr-char HOM-arr-char H.ide-in-hom
```

```
    by auto
  show H.dom (?Trn t) = ?Trn (RC.H.dom t)
  proof (intro H.dom-eqI)
    show 3: H.ide (?Trn (RC.H.dom t))
      using t 2 RC.arr-coincidence Id-def RC.arr-char
            RTS.One.arr-char RTS.One.is-extensional-rts
            RTS.One.small-rts-axioms RTS.bij-mkarr(3)
      by (auto simp add: RC.H-dom-simp RC.H-cod-simp)
    show H.seq (?Trn t) (?Trn (RC.H.dom t))
      by (metis 3 H.ide-char H.seqI RC.Dom.simps(1) RC.H-dom-simp
          RC.arr-coincidence Trn.preserves-reflects-arr
          Trn-MkArr.inv-simp arr-coincidence t)
  qed
qed
show ⋀f. RC.H.arr f ⟹ H.cod (?Trn f) = ?Trn (RC.H.cod f)
proof -
  fix t
  assume t: RC.H.arr t
  have 2: RC.arr (RC.MkArr (RC.Cod t) (RC.Cod t) (RC.Cod t))
    using t RC.arr-coincidence RC.arr-char HOM-arr-char H.ide-in-hom
    by auto
  show H.cod (?Trn t) = ?Trn (RC.H.cod t)
  proof (intro H.cod-eqI)
    show 3: H.ide (?Trn (RC.H.cod t))
      using t 2 RC.arr-coincidence Id-def RC.arr-char
            RTS.One.arr-char RTS.One.is-extensional-rts
            RTS.One.small-rts-axioms RTS.bij-mkarr(3)
      by (auto simp add: RC.H-dom-simp RC.H-cod-simp)
    show H.seq (?Trn (RC.H.cod t)) (?Trn t)
      by (metis 1 3 H.ide-char H.seqI RC.Cod.simps(1)
          RC.H-cod-simp Trn.preserves-reflects-arr
          Trn-MkArr.inv-simp arr-coincidence t)
  qed
qed
fix f g
assume fg: RC.H.seq g f
interpret DOM: simulation
              ‹HOM_EC (RC.Dom f) (RC.Cod f)›
              ‹RTS.Rts (RTS.cod (Hom (RC.Dom f) (RC.Cod f)))›
              ‹RTS.Map (Hom (RC.Dom f) (RC.Cod f))›
  using fg ide-Hom RTS.ide-char RTS.arrD
  by (metis (no-types, lifting) RC.H.seqE RC.H-arr-char)
interpret COD: simulation ‹HOM_EC (RC.Cod f) (RC.Cod g)›
              ‹RTS.Rts (RTS.cod (Hom (RC.Cod f) (RC.Cod g)))›
              ‹RTS.Map (Hom (RC.Cod f) (RC.Cod g))›
  using fg ide-Hom RTS.ide-char RTS.arrD
  by (metis (no-types, lifting) RC.H.seqE RC.H-arr-char)
interpret CODxDOM: product-rts
                ‹HOM_EC (RC.Cod f) (RC.Cod g)›
```

```
                ‹HOM_EC (RC.Dom f) (RC.Cod f)›
    ..
    interpret PU: inverse-simulations
                ‹RTS.Rts
                  (RTS.dom
                    (Hom (RC.Cod f) (RC.Cod g) ⊗
                      Hom (RC.Dom f) (RC.Cod f)))›
                CODxDOM.resid
                ‹RTS.Pack
                  (Hom (RC.Cod f) (RC.Cod g))
                  (Hom (RC.Dom f) (RC.Cod f))›
                ‹RTS.Unpack
                  (Hom (RC.Cod f) (RC.Cod g))
                  (Hom (RC.Dom f) (RC.Cod f))›
      using fg RC.H-seq-char RC.arr-char
            RTS.inverse-simulations-Pack-Unpack
      by simp
    have 4: COD.A.arr (RC.Trn g) ∧ DOM.A.arr (RC.Trn f)
      using fg RC.arr-coincidence RC.H-arr-char
      by (elim RC.H.seqE) (auto simp add: RC.H-dom-simp RC.H-cod-simp)
  have RTS.Unpack (Hom (RC.Cod f) (RC.Cod g)) (Hom (RC.Dom f) (RC.Cod
f))
        (RTS.Pack (Hom (RC.Cod f) (RC.Cod g)) (Hom (RC.Dom f) (RC.Cod
f))
            (RC.Trn g, RC.Trn f)) =
        (RC.Trn g, RC.Trn f)
      using PU.inv 4 by auto
    moreover have RC.arr
                  (RC.MkArr (RC.Dom f) (RC.Cod g) (RC.Trn g ⋆ RC.Trn f))
      by (metis (no-types, lifting) 4 RC.H.seqE RC.H-arr-char RC.arr-MkArr
          H.comp-in-homI HOM-arr-char fg)
    ultimately show ?Trn (RC.hcomp g f) = hcomp (?Trn g) (?Trn f)
    using fg RC.hcomp-def Comp-def PU.inv RC.arr-coincidence RTS.Map-mkarr
      apply auto[1]
      using RC.arr-char RTS.bij-mkarr(3) by force
  qed

  interpretation MkArr: functor hcomp RC.hcomp
                    ‹λt. if arr t then RC.MkArr (H.dom t) (H.cod t) t
                         else RC.null›
  proof
    let ?MkArr = λt. if arr t then RC.MkArr (H.dom t) (H.cod t) t
                     else RC.null
    show ⋀f. ¬ H.arr f ⟹ ?MkArr f = RC.H.null
      using arr-coincidence RC.null-coincidence by auto
    show ⋀f. H.arr f ⟹ RC.H.arr (?MkArr f)
      using arr-coincidence RC.arr-coincidence
      by (metis MkArr.preserves-reflects-arr)
    have 1: ⋀f. H.arr f ⟹ RC.arr (RC.MkArr (H.dom f) (H.cod f) f)
```

```
  using MkArr.preserves-reflects-arr arr-coincidence H.in-homI HOM-arr-char
    by (intro RC.arr-MkArr) auto
  thus ⋀f. H.arr f ⟹ RC.H.dom (?MkArr f) = ?MkArr (H.dom f)
   using RC.H-dom-char Id-def RTS.One.arr-char RTS.One.is-extensional-rts
         RTS.One.small-rts-axioms RTS.bij-mkarr(3)
    by auto
  show ⋀f. H.arr f ⟹ RC.H.cod (?MkArr f) = ?MkArr (H.cod f)
   using 1 RC.H-cod-char Id-def RTS.One.arr-char RTS.One.is-extensional-rts
         RTS.One.small-rts-axioms RTS.bij-mkarr(3)
    by auto
  show ⋀g f. H.seq g f ⟹
               ?MkArr (g ⋆ f) = RC.hcomp (?MkArr g) (?MkArr f)
  proof −
    fix f g
    assume fg: H.seq g f
    interpret COD: extensional-rts ‹RTS.Rts (Hom (H.cod f) (H.cod g))›
      using fg ide-Hom [of H.cod f H.cod g] arr-coincidence
      by (metis H.ide-cod H.seqE RTS.ideD_RTSC(1) mem-Collect-eq)
    interpret DOM: extensional-rts ‹RTS.Rts (Hom (H.dom f) (H.cod f))›
      using fg ide-Hom [of H.dom f H.cod f] arr-coincidence
      by (metis H.ide-dom H.seqE RTS.ideD_RTSC(1) mem-Collect-eq)
    interpret CODxDOM: product-rts
                       ‹RTS.Rts (Hom (H.cod f) (H.cod g))›
                       ‹RTS.Rts (Hom (H.dom f) (H.cod f))›
      ..
    show ?MkArr (g ⋆ f) = RC.hcomp (?MkArr g) (?MkArr f)
    proof −
      have RC.hcomp (?MkArr g) (?MkArr f) =
           RC.MkArr (dom f) (cod g)
             (RTS.Map (Comp (dom f) (cod f) (cod g))
               (RTS.Pack
                  (Hom (cod f) (cod g)) (Hom (dom f) (cod f)) (g, f)))
        using fg RC.hcomp-def [of ?MkArr g ?MkArr f] H.seqE by auto
      also have ... = RC.MkArr (dom f) (cod g) (g ⋆ f)
        by (metis 1 CODxDOM.arr-char H.seqE RC.Cod.simps(1)
            RC.Dom.simps(1) RC.Trn.simps(1) RC.arr-char Map-Comp-Pack
            fg fst-conv snd-conv)
      also have ... = ?MkArr (g ⋆ f)
        using fg by simp
      finally show ?thesis by simp
    qed
  qed
qed

interpretation Trn-MkArr: inverse-functors hcomp RC.hcomp
                          ‹λt. if RC.arr t then RC.Trn t else null›
                          ‹λt. if arr t
                               then RC.MkArr (H.dom t) (H.cod t) t
                               else RC.null›
```

proof
let *?Trn* = λt. *if RC.arr t then RC.Trn t else null*
let *?MkArr* = λt. *if arr t*
then RC.MkArr (*H.dom t*) (*H.cod t*) *t*
else RC.null
show *?MkArr* $\circ$ *?Trn* = *RC.H.map*
by (*auto simp add*: *RC.H.map-def Trn-MkArr.inv*)
show (*?Trn* $\circ$ *?MkArr*) = *H.map*
using *arr-coincidence null-coincidence H.is-extensional* **by** *auto*
qed

lemma *inverse-functors-Trn-MkArr*:
shows *inverse-functors hcomp RC.hcomp*
(λt. *if RC.arr t then RC.Trn t else null*)
(λt. *if arr t then RC.MkArr* (*H.dom t*) (*H.cod t*) *t else RC.null*)
..

proposition *induces-rts-category-isomorphism*:
shows *rts-category-isomorphism resid hcomp RC.resid RC.hcomp*
(λt. *if arr t then RC.MkArr* (*H.dom t*) (*H.cod t*) *t else RC.null*)
using *Trn-MkArr.inverse-functors-axioms*
Trn-MkArr.inverse-simulations-axioms
by *unfold-locales auto*

end

5.4.2 Enriched Category to RTS-Category to Enriched Category

context *rts-category-of-small-enriched-category*
begin

As it is easy to get lost in the types and definitions, we begin with a road map of the construction to be performed. We are given a small RTS-enriched category (*Obj*, *Hom*, *Id*, *Comp*) with objects at type $'O$ and as base category the category **RTS** with arrow type $'A$ *rtscat.arr*. From this, we constructed a "global RTS" R by stitching together all of the RTS's underlying the hom-objects. We then reduced the type of R by taking its image under an injective map on arrows, to obtain an isomorphic RTS R' at arrow type $'A$. The smallness assumption was used for this. Next, we will extend R' to a locally small RTS-category R'' (new name is used to avoid name clashes within sublocales) by equipping it with the horizontal composition ($\star'$) derived from the composition of the originally given enriched category. From R'' we then construct an RTS-enriched category (*R''.Obj R''.Hom R''.Id R''.Comp*).

interpretation R'': *locally-small-rts-category* R' *hcomp'*
using *is-locally-small-rts-category* **by** *blast*

interpretation R'': *enriched-category-of-rts-category arr-type* R' *hcomp'*

..

Our objective is now to construct a fully faithful RTS-enriched functor (F_o, F_a), from the originally given RTS-enriched category (*Obj*, *Hom*, *Id*, *Comp*) to the newly constructed RTS-category (*R''.Obj R''.Hom R''.Id R''.Comp*). Note that this makes sense, because, due to the type reduction from *R'* to *R''*, we have arranged for the base category of (*R''.Obj R''.Hom R''.Id R''.Comp*) to be the same category **RTS** as that of the originally given (*Obj*, *Hom*, *Id*, *Comp*). The object map F_o will take $a \in Obj :: 'O\ set$ to *DN* (*MkArr a a* (*RTS.Map* (*Id a*) *one*)) $\in$ *R''.Obj* :: *'A set*. The arrow map F_a will take each pair (*a*, *b*) of elements of *Obj* to an invertible arrow «F_a *a b* : *Hom a b* $\rightarrow$ *R''.Hom* (F_o *a*) (F_o *b*)» of **RTS**. This arrow corresponds to the invertible simulation from HOM_{EC} *a b* to *R''.*HOM_{EC} (F_o *a*) (F_o *b*) that takes *t* $\in$ *Hom a b* to *DN* (*MkArr a b t*) $\in$ *R''.*HOM_{EC} (F_o *a*) (F_o *b*).

abbreviation $F_o :: 'O \Rightarrow 'A$
where $F_o \equiv \lambda a.$ *DN* (*MkArr a a* (*RTS.Map* (*Id a*) *RTS.One.the-arr*))

abbreviation $F_a :: 'O \Rightarrow 'O \Rightarrow 'A$ *rtscat.arr*
where $F_a \equiv \lambda a\ b.$ *if* $a \in Obj \wedge b \in Obj$
then RTS.mkarr (HOM_{EC} *a b*) (*R''.*HOM_{EC} (F_o *a*) (F_o *b*))
($\lambda t.$ *if residuation.arr* (HOM_{EC} *a b*) *t*
then DN (*MkArr a b t*)
else ResiduatedTransitionSystem.partial-magma.null
(*R''.*HOM_{EC} (F_o *a*) (F_o *b*)))
else RTS.null

lemma *ide-*F_o:
assumes $a \in Obj$
shows *DN* (*MkArr a a* (*RTS.Map* (*Id a*) *RTS.One.the-arr*)) $\in$ *Collect H'.ide*
using *H'-ide-char Id-yields-horiz-ide assms obj-implies-sta* **by** *auto*

lemma *bij-*F_o:
shows *bij-betw* F_o *Obj R''.Obj*
proof −
have *bij-betw* (*DN* $\circ$ ($\lambda A.$ *MkArr A A* (*RTS.Map* (*Id A*) *RTS.One.the-arr*)))
Obj (*Collect H'.ide*)
proof −
have *bij-betw DN* (*Collect H.ide*) (*Collect H'.ide*)
using *H'-ide-char H.ideD*(*1*) *H'.ideD*(*1*)
by (*intro bij-betwI*) *auto*
thus *?thesis*
using *bij-betw-Obj-horiz-ide bij-betw-trans* **by** *blast*
qed
moreover
have *DN* $\circ$ ($\lambda A.$ *MkArr A A* (*RTS.Map* (*Id A*) *RTS.One.the-arr*)) = F_o
by *auto*
ultimately show *?thesis* **by** *simp*

qed

lemma F_a*-in-hom* [*intro*, *simp*]:
assumes $a \in Obj$ **and** $b \in Obj$
shows «$F_a\ a\ b : Hom\ a\ b \to R''.Hom\ (F_o\ a)\ (F_o\ b)$»
proof
  show $RTS.arr\ (F_a\ a\ b)$
  proof −
    have $RTS.arr$
          $((RTS.mkarr\ (HOM_{EC}\ a\ b)\ (R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b))$
             $(\lambda t.\ if\ residuation.arr\ (HOM_{EC}\ a\ b)\ t\ then\ DN\ (MkArr\ a\ b\ t)$
                $else\ ResiduatedTransitionSystem.partial\text{-}magma.null$
                   $(R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b)))))$
    proof (*intro* $RTS.arrI_{RTSC}$)
      interpret *HOM*: *extensional-rts* ‹$HOM_{EC}\ a\ b$›
        using *assms* **by** *simp*
      interpret *HOM'*: *extensional-rts* ‹$R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b)$›
        using *assms* $R''.ide\text{-}Hom\ RTS.ideD_{RTSC}\ RTS.arrD$
        by (*metis* (*no-types*, *lifting*) *ide-*F_o)
      show *extensional-rts* $(HOM_{EC}\ a\ b) \wedge$ *small-rts* $(HOM_{EC}\ a\ b)$
        using *assms* **by** *simp*
      show *extensional-rts* $(R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b)) \wedge$
            small-rts $(R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b))$
        using *assms* $R''.ide\text{-}Hom\ RTS.ideD_{RTSC}\ RTS.arrD$
        by (*metis* (*no-types*, *lifting*) *ide-*F_o)

To prove the rest we need information about $R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b)$. Rather than just having it as an abstract RTS, we need to know that it is a sub-RTS of R', which in turn is isomorphic (via DN) to the "global RTS" R, which has arrows of the form $MkArr\ a\ b\ t$.

      have ∗: $R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b) =$
            $sub\text{-}rts.resid\ R'\ (\lambda t.\ H'.in\text{-}hom\ t\ (F_o\ a)\ (F_o\ b))$
        using *assms* $R''.Hom\text{-}def\ [of\ F_o\ a\ F_o\ b]$ *ide-*F_o $RTS.bij\text{-}mkide(3)$
        by *simp*
      interpret *HOM'-alt*: *sub-rts* R' ‹$\lambda t.\ H'.in\text{-}hom\ t\ (F_o\ a)\ (F_o\ b)$›
        using *assms* *ide-*F_o $R''.sub\text{-}rts\text{-}HOM$ **by** *metis*
      have $(\lambda t.\ if\ HOM.arr\ t\ then\ DN\ (MkArr\ a\ b\ t)\ else\ HOM'.null)$
            $\in Collect\ (simulation\ (HOM_{EC}\ a\ b)\ (R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b)))$
      proof
        show $simulation\ (HOM_{EC}\ a\ b)\ (R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b))$
              $(\lambda t.\ if\ HOM.arr\ t\ then\ DN\ (MkArr\ a\ b\ t)\ else\ HOM'.null)$
        proof
          show $\bigwedge t.\ \neg\ HOM.arr\ t \Longrightarrow$
                $(if\ HOM.arr\ t\ then\ DN\ (MkArr\ a\ b\ t)\ else\ HOM'.null) =$
                $HOM'.null$
            by *simp*
          fix *t u*
          assume *tu*: $HOM.con\ t\ u$
          have *0*: $V.con\ (MkArr\ a\ b\ t)\ (MkArr\ a\ b\ u)$

```
    using tu Con-def
    by (simp add: assms(1−2) con-char HOM.con-implies-arr(1−2))
  have 1: R'.con (DN (MkArr a b t)) (DN (MkArr a b u))
    using 0 UP-DN.G.preserves-con by simp
  have 2: HOM'-alt.con (DN (MkArr a b t)) (DN (MkArr a b u))
  proof −
    have H'.in-hom (DN (MkArr a b t)) (F_o a) (F_o b)
    proof −
      have H.arr (UP (DN (MkArr a b t)))
        using 1 R'.con-implies-arr(1) by auto
      thus ?thesis
        using assms
        by (simp add: H'.in-homI H'-cod-char H'-dom-char
            H-cod-char H-dom-char)
    qed
    moreover have H'.in-hom (DN (MkArr a b u)) (F_o a) (F_o b)
      by (metis (no-types, lifting) 1 H'.in-homE H'.in-homI
          R''.con-implies-hpar calculation)
    ultimately show ?thesis
      using 1 HOM'-alt.con-char by blast
  qed
show HOM'.con (if HOM.arr t then DN (MkArr a b t) else HOM'.null)
              (if HOM.arr u then DN (MkArr a b u) else HOM'.null)
    using tu * 2 HOM.con-implies-arr by auto
  show (if HOM.arr (HOM_EC a b t u)
        then DN (MkArr a b (HOM_EC a b t u))
        else HOM'.null) =
       R''.HOM_EC (F_o a) (F_o b)
         (if HOM.arr t then DN (MkArr a b t) else HOM'.null)
         (if HOM.arr u then DN (MkArr a b u) else HOM'.null)
  proof −
    have (if HOM.arr (HOM_EC a b t u)
          then DN (MkArr a b (HOM_EC a b t u))
          else HOM'.null) =
         HOM'-alt.resid
           (if HOM.arr t then DN (MkArr a b t) else HOM'.null)
           (if HOM.arr u then DN (MkArr a b u) else HOM'.null)
    proof −
      have H'.in-hom (DN (MkArr a b t)) (F_o a) (F_o b)
        using assms tu 2 HOM'-alt.con-char by fastforce
      moreover have H'.in-hom (DN (MkArr a b u)) (F_o a) (F_o b)
        using assms tu 2 HOM'-alt.con-char by fastforce
      moreover have R' (DN (MkArr a b t)) (DN (MkArr a b u)) =
                    DN (MkArr a b (HOM_EC a b t u))
        by (metis 0 Cod.simps(1) Dom.simps(1) Trn.simps(1)
            UP-DN.G.preserves-resid con-char resid.simps(3))
      ultimately show ?thesis
        unfolding HOM'-alt.resid-def
      using tu 1 HOM.con-implies-arr UP-DN.inv UP-DN.inv' by auto
```

```
        qed
        also have ... = R''.HOM_EC (F_o a) (F_o b)
                          (if HOM.arr t then DN (MkArr a b t) else HOM'.null)
                        (if HOM.arr u then DN (MkArr a b u) else HOM'.null)
          using ∗ by simp
        finally show ?thesis by simp
      qed
    qed
  qed
  thus RTS.mkarr (HOM_EC a b) (R''.HOM_EC (F_o a) (F_o b))
       (λt. if HOM.arr t then DN (MkArr a b t) else HOM'.null)
       ∈ RTS.mkarr (HOM_EC a b) (R''.HOM_EC (F_o a) (F_o b)) ‘
         Collect (simulation (HOM_EC a b) (R''.HOM_EC (F_o a) (F_o b)))
    by auto
 qed
 thus ?thesis
  using assms by simp
qed
show RTS.dom (F_a a b) = Hom a b
 using assms(1−2) ‹RTS.arr (F_a a b)› by auto
show RTS.cod (F_a a b) = R''.Hom (F_o a) (F_o b)
 using assms(1−2) ‹RTS.arr (F_a a b)› ide-F_o by auto
qed

lemma F_a-simps [simp]:
assumes a ∈ Obj and b ∈ Obj
shows RTS.arr (F_a a b)
and RTS.dom (F_a a b) = Hom a b
and RTS.cod (F_a a b) = R''.Hom (F_o a) (F_o b)
 using assms F_a-in-hom by blast+

lemma Map-F_a-simp [simp]:
assumes a ∈ Obj and b ∈ Obj and residuation.arr (HOM_EC a b) t
shows RTS.Map (F_a a b) t = DN (MkArr a b t)
 using assms RTS.bij-mkarr(3) ide-F_o by force

interpretation Φ: rts-enriched-functor
                  Obj Hom Id Comp R''.Obj R''.Hom R''.Id R''.Comp
                  F_o F_a
proof
 show ⋀a. a ∈ Obj ⟹ F_o a ∈ R''.Obj
  using ide-F_o by blast
 show ⋀a b. ⟦a ∈ Obj; b ∈ Obj⟧ ⟹
              «F_a a b : Hom a b → R''.Hom (F_o a) (F_o b)»
  using F_a-in-hom by blast
 show ⋀a b. a ∉ Obj ∨ b ∉ Obj ⟹ F_a a b = RTS.null
  by auto
 show ⋀a. a ∈ Obj ⟹ F_a a a · Id a = R''.Id (F_o a)
 proof −
```

```
  fix a
  assume a: a ∈ Obj
  show F_a a a · Id a = R''.Id (F_o a)
  proof (intro RTS.arr-eqI)
    show 1: RTS.par (F_a a a · Id a) (R''.Id (F_o a))
      using a Id-in-hom R''.Id-in-hom F_a-in-hom ide-F_o RTS.in-homE
      apply (intro conjI)
        apply auto[1]
      by fastforce+
    show RTS.Map (F_a a a · Id a) = RTS.Map (R''.Id (F_o a))
    proof −
      interpret Map-Id-a: simulation RTS.One.resid ‹HOM_EC a a›
                                     ‹RTS.Map (Id a)›
        using a Id-in-hom [of a] RTS.arrD(3) [of Id a] RTS.unity-agreement
        by auto
      interpret Map-F_a-aa: simulation
                                     ‹HOM_EC a a› ‹RTS.Rts (R''.Hom (F_o a) (F_o a))›
                                     ‹RTS.Map (F_a a a)›
        using a F_a-in-hom [of a a] RTS.arrD(3) by fastforce
      interpret Map-F_a-aa-o-Map-Id-a: simulation
                                          RTS.One.resid
                                          ‹RTS.Rts (R''.Hom (F_o a) (F_o a))›
                                          ‹RTS.Map (F_a a a) ∘ RTS.Map (Id a)›
        using simulation-comp Map-Id-a.simulation-axioms
              Map-F_a-aa.simulation-axioms
        by blast
      interpret Map-R''-Id-F_o-a: simulation
                                     RTS.One.resid
                                     ‹RTS.Rts (R''.Hom (F_o a) (F_o a))›
                                     ‹RTS.Map (R''.Id (F_o a))›
        using a
        by (metis (no-types, lifting) R''.Id-in-hom RTS.Rts-one RTS.arrD(3)
            RTS.in-homE RTS.unity-agreement ide-F_o)
      have RTS.Map (F_a a a · Id a) = RTS.Map (F_a a a) ∘ RTS.Map (Id a)
        using 1 RTS.Map-comp by blast
      also have ... = RTS.Map (R''.Id (F_o a))
        using Map-Id-a.preserves-reflects-arr Map-R''-Id-F_o-a.extensional
              R''.Id-def RTS.One.arr-char RTS.One.is-extensional-rts
              RTS.One.small-rts-axioms RTS.bij-mkarr(3) a ide-F_o
        by auto
      finally show ?thesis by blast
    qed
  qed
qed
show ⋀a b c. ⟦a ∈ Obj; b ∈ Obj; c ∈ Obj⟧ ⟹
              R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b) =
              F_a a c · Comp a b c
proof −
  fix a b c
```

```
      assume a: a ∈ Obj and b: b ∈ Obj and c: c ∈ Obj
      show R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b) =
           F_a a c · Comp a b c
      proof (intro RTS.arr-eqI)
        show 1: RTS.par
                  (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b))
                  (F_a a c · Comp a b c)
        proof (intro conjI)
          show 2: RTS.seq (R''.Comp (F_o a) (F_o b) (F_o c)) (F_a b c ⊗ F_a a b)
            using a b c F_a-in-hom ide-F_o R''.Comp-in-hom R''.Hom-def by blast
          show 3: RTS.seq (F_a a c) (Comp a b c)
            using a b c F_a-in-hom ide-F_o R''.Comp-in-hom R''.Hom-def by blast
        show RTS.dom (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) =
             RTS.dom (F_a a c · Comp a b c)
          proof -
            have RTS.dom (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b))
=
                 RTS.dom (F_a b c ⊗ F_a a b)
              using a b c 2 R''.Comp-in-hom by auto
            also have ... = Hom b c ⊗ Hom a b
              using a b c F_a-in-hom
              by (meson RTS.CMC.dom-tensor)
            also have ... = RTS.dom (F_a a c · Comp a b c)
              using a b c 3 Comp-in-hom [of a b c] by auto
            finally show ?thesis by blast
          qed
        show RTS.cod (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) =
             RTS.cod (F_a a c · Comp a b c)
          proof -
            have RTS.cod (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) =
                 RTS.cod (R''.Comp (F_o a) (F_o b) (F_o c))
              using a b c 2 F_a-in-hom R''.Comp-in-hom by auto
            also have ... = RTS.cod (F_a a c · Comp a b c)
              using a b c 3 ide-F_o R''.Comp-in-hom by auto
            finally show ?thesis by blast
          qed
        qed
        show RTS.Map (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) =
             RTS.Map (F_a a c · Comp a b c)
        proof
          interpret Dom-bc: extensional-rts ‹HOM_EC b c›
            using b c by simp
          interpret Dom-ab: extensional-rts ‹HOM_EC a b›
            using a b by simp
          interpret Dom-bc-X-Dom-ab: product-rts ‹HOM_EC b c› ‹HOM_EC a b›
            ..
          interpret Dom-bcxab: extensional-rts ‹RTS.Rts (Hom b c ⊗ Hom a b)›
            using a b c by simp
          have 3: RTS.ide (RTS.dom (F_a b c)) ∧ RTS.ide (RTS.dom (F_a a b))
```

```
  using a b c F_a-in-hom RTS.ide-dom by blast
have 4: RTS.ide (RTS.cod (F_a b c)) ∧ RTS.ide (RTS.cod (F_a a b))
  using a b c F_a-in-hom RTS.ide-cod by blast
interpret Cod-bc: extensional-rts ‹RTS.Rts (RTS.cod (F_a b c))›
  using 4 RTS.ide-char RTS.arrD RTS.arr-cod-iff-arr
  by (metis (no-types, lifting))
interpret Cod-ab: extensional-rts ‹RTS.Rts (RTS.cod (F_a a b))›
  using 4 RTS.ide-char RTS.arrD RTS.arr-cod-iff-arr
  by (metis (no-types, lifting))
interpret Cod-bc-X-Cod-ab: product-rts
                             ‹RTS.Rts (RTS.cod (F_a b c))›
                             ‹RTS.Rts (RTS.cod (F_a a b))›
  ..
interpret F_abc: simulation ‹HOM_EC b c›
                 ‹RTS.Rts (RTS.cod (F_a b c))›
                 ‹RTS.Map (F_a b c)›
  using b c F_a-in-hom [of b c] RTS.arrD(3) [of F_a b c] by auto
interpret F_aab: simulation ‹HOM_EC a b›
                 ‹RTS.Rts (RTS.cod (F_a a b))›
                 ‹RTS.Map (F_a a b)›
  using a b F_a-in-hom [of a b] RTS.arrD(3) [of F_a a b] by auto
interpret F_abc-X-F_aab: product-simulation
                       ‹HOM_EC b c› ‹HOM_EC a b›
                       ‹RTS.Rts (RTS.cod (F_a b c))›
                       ‹RTS.Rts (RTS.cod (F_a a b))›
                       ‹RTS.Map (F_a b c)› ‹RTS.Map (F_a a b)›
  ..
interpret U: simulation
               ‹RTS.Rts (Hom b c ⊗ Hom a b)›
               Dom-bc-X-Dom-ab.resid
               ‹RTS.Unpack (RTS.dom (F_a b c)) (RTS.dom (F_a a b))›
proof −
  have RTS.arr (F_a b c) ∧ RTS.arr (F_a a b)
    using a b c F_a-in-hom by blast
  thus simulation
         (RTS.Rts (Hom b c ⊗ Hom a b))
         Dom-bc-X-Dom-ab.resid
         (RTS.Unpack (RTS.dom (F_a b c)) (RTS.dom (F_a a b)))
    using a b c 1 F_a-in-hom
         RTS.simulation-Unpack [of Hom b c Hom a b]
    by fastforce
qed
fix x
show RTS.Map
       (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) x =
     RTS.Map (F_a a c · Comp a b c) x
proof (cases Dom-bcxab.arr x)
  show ¬ Dom-bcxab.arr x ⟹
         RTS.Map
```

```
      (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) x =
    RTS.Map (F_a a c · Comp a b c) x
proof −
  interpret LHS: simulation
                ‹RTS.Rts (Hom b c ⊗ Hom a b)›
                ‹R''.HOM_EC (F_o a) (F_o c)›
                ‹RTS.Map
                   (R''.Comp (F_o a) (F_o b) (F_o c) ·
                     (F_a b c ⊗ F_a a b))›
  proof −
   have RTS.seq (R''.Comp (F_o a) (F_o b) (F_o c)) (F_a b c ⊗ F_a a b)
     using a b c 1 ide-F_o by force
   moreover have RTS.Dom
                   (R''.Comp (F_o a) (F_o b) (F_o c) ·
                     (F_a b c ⊗ F_a a b)) =
                 RTS.Rts (Hom b c ⊗ Hom a b)
     using a b c 1 ide-F_o R''.ide-Hom R''.Comp-in-hom
     by (metis (no-types, lifting) Comp-in-hom RTS.dom-comp
         RTS.in-homE)
   moreover have RTS.Cod
                   (R''.Comp (F_o a) (F_o b) (F_o c) ·
                     (F_a b c ⊗ F_a a b)) =
                 R''.HOM_EC (F_o a) (F_o c)
     using a b c 1 ide-F_o R''.ide-Hom R''.Comp-in-hom
     by (metis (no-types, lifting) R''.Comp-simps(3) RTS.cod-comp)
   ultimately
   show simulation
          (RTS.Rts
             (Hom b c ⊗ Hom a b)) (R''.HOM_EC (F_o a) (F_o c))
          (RTS.Map
             (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)))
     using a b c 1 ide-F_o
          RTS.arrD(3)
            [of R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)]
     by auto
  qed
  interpret RHS: simulation
                ‹RTS.Rts (Hom b c ⊗ Hom a b)›
                ‹R''.HOM_EC (F_o a) (F_o c)›
                ‹RTS.Map (F_a a c · Comp a b c)›
  proof −
    have RTS.seq (F_a a c) (Comp a b c)
      using a b c 1 ide-F_o by force
    moreover have RTS.Dom (F_a a c · Comp a b c) =
                  RTS.Rts (Hom b c ⊗ Hom a b)
      using a b c 1 ide-F_o R''.ide-Hom R''.Comp-in-hom
      by (metis (no-types, lifting) Comp-in-hom RTS.dom-comp
          RTS.in-homE)
    moreover have RTS.Cod (F_a a c · Comp a b c) =
```

```
                R''.HOM_EC (F_o a) (F_o c)
      using a b c 1 ide-F_o R''.ide-Hom R''.Comp-in-hom
      by (metis (no-types, lifting) R''.Comp-simps(3) RTS.cod-comp)
    ultimately
     show simulation
           (RTS.Rts
              (Hom b c ⊗ Hom a b)) (R''.HOM_EC (F_o a) (F_o c))
           (RTS.Map (F_a a c · Comp a b c))
      using a b c 1 ide-F_o
           RTS.arrD(3) [of F_a a c · Comp a b c]
      by auto
  qed
  assume x: ¬ Dom-bcxab.arr x
  show RTS.Map
         (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) x =
       RTS.Map (F_a a c · Comp a b c) x
    using x LHS.extensional RHS.extensional by presburger
 qed
 assume x: Dom-bcxab.arr x
have 0: Dom-bc-X-Dom-ab.arr (RTS.Unpack (Hom b c) (Hom a b) x)
  using a b c x U.preserves-reflects-arr [of x]
        F_a-in-hom [of b c] F_a-in-hom [of a b]
        Dom-bc-X-Dom-ab.arr-char
  by (metis (no-types, lifting) RTS.in-homE)

 have RTS.Map
        (R''.Comp (F_o a) (F_o b) (F_o c) · (F_a b c ⊗ F_a a b)) x =
      RTS.Map
        (R''.Comp (F_o a) (F_o b) (F_o c))
        (RTS.Map (F_a b c ⊗ F_a a b) x)
  using a b c 1 ide-F_o RTS.Map-comp by auto
 also have ... =
          fst (RTS.Unpack
                (R''.Hom (F_o b) (F_o c)) (R''.Hom (F_o a) (F_o b))
                  (RTS.Map (F_a b c ⊗ F_a a b) x)) ⋆´
            snd (RTS.Unpack
                  (R''.Hom (F_o b) (F_o c))
                  (R''.Hom (F_o a) (F_o b))
                  (RTS.Map (F_a b c ⊗ F_a a b) x))
  using a b c R''.Comp-def RTS.bij-mkarr(3)
        ‹⋀a. a ∈ Obj ⟹ F_o a ∈ R''.Obj›
  by force
 also have ... =
          fst (RTS.Unpack (RTS.cod (F_a b c)) (RTS.cod (F_a a b))
                (RTS.Map (F_a b c ⊗ F_a a b) x)) ⋆´
            snd (RTS.Unpack (RTS.cod (F_a b c)) (RTS.cod (F_a a b))
                  (RTS.Map (F_a b c ⊗ F_a a b) x))
 proof −
  have R''.Hom (F_o b) (F_o c) = RTS.cod (F_a b c)
```

```
    using a b c ide-F_o F_a-in-hom R''.Hom-def by force
  moreover have R''.Hom (F_o a) (F_o b) = RTS.cod (F_a a b)
    using a b c ide-F_o F_a-in-hom R''.Hom-def by force
  ultimately show ?thesis by argo
qed
also have ... =
          fst
            (RTS.Unpack (RTS.cod (F_a b c)) (RTS.cod (F_a a b))
              ((RTS.Pack (RTS.cod (F_a b c)) (RTS.cod (F_a a b)) ∘
                  product-simulation.map
                    (HOM_EC b c) (HOM_EC a b)
                    (RTS.Map (F_a b c)) (RTS.Map (F_a a b)) ∘
                      RTS.Unpack
                        (RTS.dom (F_a b c)) (RTS.dom (F_a a b))) x)) ⋆´
            snd
              (RTS.Unpack (RTS.cod (F_a b c)) (RTS.cod (F_a a b))
                ((RTS.Pack (RTS.cod (F_a b c)) (RTS.cod (F_a a b)) ∘
                    F_a bc-X-F_a ab.map ∘
                      RTS.Unpack
                        (RTS.dom (F_a b c)) (RTS.dom (F_a a b))) x))
proof −
  have RTS.Map (F_a b c ⊗ F_a a b) =
        RTS.Pack (RTS.cod (F_a b c)) (RTS.cod (F_a a b)) ∘
          product-simulation.map
            (RTS.Dom (F_a b c)) (RTS.Dom (F_a a b))
            (RTS.Map (F_a b c)) (RTS.Map (F_a a b)) ∘
              RTS.Unpack (RTS.dom (F_a b c)) (RTS.dom (F_a a b))
    using a b c 4 RTS.Map-prod [of F_a b c F_a a b] F_a-in-hom
    by (metis (no-types, lifting) RTS.arr-cod-iff-arr RTS.ideD(1)
        RTS.tensor-agreement)
  moreover have RTS.Dom (F_a b c) = HOM_EC b c ∧
                RTS.Dom (F_a a b) = HOM_EC a b
    using a b c F_a-in-hom [of b c] F_a-in-hom [of a b] by auto
  ultimately show ?thesis by simp
qed
also have ... =
          fst (RTS.Unpack (RTS.cod (F_a b c)) (RTS.cod (F_a a b))
                (RTS.Pack (RTS.cod (F_a b c)) (RTS.cod (F_a a b))
                  (F_a bc-X-F_a ab.map
                    (RTS.Unpack (Hom b c) (Hom a b) x)))) ⋆´
          snd (RTS.Unpack (RTS.cod (F_a b c)) (RTS.cod (F_a a b))
                (RTS.Pack (RTS.cod (F_a b c)) (RTS.cod (F_a a b))
                  (F_a bc-X-F_a ab.map
                    (RTS.Unpack (Hom b c) (Hom a b) x))))
  using a b c F_a-in-hom [of b c] F_a-in-hom [of a b] by fastforce
also have ... =
          fst
            (F_a bc-X-F_a ab.map
              (RTS.Unpack (Hom b c) (Hom a b) x)) ⋆´
```

```
        snd
          (F_a bc-X-F_a ab.map
            (RTS.Unpack (Hom b c) (Hom a b) x))
proof −
 have 1: RTS.ide (RTS.cod (F_a b c)) ∧ RTS.ide (RTS.cod (F_a a b))
   using a b c F_a-in-hom RTS.ide-cod by blast
 interpret PU: inverse-simulations
               ‹RTS.Rts (RTS.cod (F_a b c) ⊗ RTS.cod (F_a a b))›
               Cod-bc-X-Cod-ab.resid
               ‹RTS.Pack (RTS.cod (F_a b c)) (RTS.cod (F_a a b))›
               ‹RTS.Unpack (RTS.cod (F_a b c)) (RTS.cod (F_a a b))›
   using a b c 1 F_a-in-hom [of a b] F_a-in-hom [of b c]
         RTS.inverse-simulations-Pack-Unpack
   by fastforce
 show ?thesis
 proof −
   have Cod-bc-X-Cod-ab.arr
          (F_a bc-X-F_a ab.map
            (RTS.Unpack (Hom b c) (Hom a b) x))
     using a b c x U.preserves-reflects-arr
           F_a-in-hom [of b c] F_a-in-hom [of a b]
     by fastforce
   thus ?thesis by simp
 qed
qed
also have ... =
        RTS.Map (F_a b c)
          (fst (RTS.Unpack (Hom b c) (Hom a b) x)) ⋆´
        RTS.Map
          (F_a a b) (snd (RTS.Unpack (Hom b c) (Hom a b) x))
 using 0 F_a bc-X-F_a ab.map-def by auto
also have ... =
        DN (MkArr b c
              (fst (RTS.Unpack (Hom b c) (Hom a b) x))) ⋆´
        DN (MkArr a b
              (snd (RTS.Unpack (Hom b c) (Hom a b) x)))
 using a b c 0 Map-F_a-simp by auto
also have ... =
       DN (MkArr b c (fst (RTS.Unpack (Hom b c) (Hom a b) x)) ⋆
           MkArr a b (snd (RTS.Unpack (Hom b c) (Hom a b) x)))
 using 0 a b c arr-MkArr by force
also have ... =
        DN (MkArr a c
              (RTS.Map (Comp a b c)
                (RTS.Pack (Hom b c) (Hom a b)
                  (RTS.Unpack (Hom b c) (Hom a b) x))))
 using 0 a b c x hcomp-def
 by (simp add: arr-MkArr)
also have ... = DN (MkArr a c (RTS.Map (Comp a b c) x))
```

```
        proof −
          interpret PU: inverse-simulations
                        ‹RTS.Rts (Hom b c ⊗ Hom a b)›
                        Dom-bc-X-Dom-ab.resid
                        ‹RTS.Pack (Hom b c) (Hom a b)›
                        ‹RTS.Unpack (Hom b c) (Hom a b)›
            using a b c RTS.inverse-simulations-Pack-Unpack by simp
          show ?thesis
            using a b c x PU.inv' by simp
        qed
        also have ... = RTS.Map (F_a a c) (RTS.Map (Comp a b c) x)
          using a b c x Map-F_a-simp R''.HOM-null-char RTS.bij-mkarr(3)
                UP-DN.G.extensional arr-char ide-F_o
          by force
        also have ... = (RTS.Map (F_a a c) ∘ RTS.Map (Comp a b c)) x
          by simp
        also have ... = RTS.Map (F_a a c · Comp a b c) x
          using a b c x F_a-in-hom [of a c] Comp-in-hom [of a b c]
                RTS.Map-comp by fastforce
        finally show ?thesis by blast
      qed
    qed
  qed
qed

lemma induces-rts-enriched-functor:
shows rts-enriched-functor
        Obj Hom Id Comp R''.Obj R''.Hom R''.Id R''.Comp F_o F_a
  ..

proposition induces-fully-faithful-rts-enriched-functor:
shows fully-faithful-rts-enriched-functor
        Obj Hom Id Comp R''.Obj R''.Hom R''.Id R''.Comp F_o F_a
proof
  show ⋀a b. ⟦a ∈ Obj; b ∈ Obj⟧ ⟹ RTS.iso (F_a a b)
  proof −
    fix a b
    assume a: a ∈ Obj and b: b ∈ Obj

    have ∗: R''.HOM_EC (F_o a) (F_o b) =
            sub-rts.resid R' (λt. H'.in-hom t (F_o a) (F_o b))
      using a b R''.Hom-def [of F_o a F_o b] ide-F_o RTS.bij-mkide(3) by auto
    show RTS.iso (F_a a b)
    proof −
      have invertible-simulation
            (RTS.Rts (RTS.dom (F_a a b))) (RTS.Rts (RTS.cod (F_a a b)))
            (RTS.Map (F_a a b))
      proof (unfold invertible-simulation-iff, intro conjI)
```

```
interpret F_a-ab: simulation
                ‹RTS.Rts (RTS.dom (F_a a b))›
                ‹RTS.Rts (RTS.cod (F_a a b))›
                ‹RTS.Map (F_a a b)›
  using a b F_a-in-hom RTS.arrD by blast
show simulation
       (RTS.Rts (RTS.dom (F_a a b))) (RTS.Rts (RTS.cod (F_a a b)))
       (RTS.Map (F_a a b))
  using F_a-ab.simulation-axioms by simp
show bij-betw (RTS.Map (F_a a b))
       (Collect (residuation.arr (RTS.Rts (RTS.dom (F_a a b)))))
       (Collect (residuation.arr (RTS.Rts (RTS.cod (F_a a b)))))
proof (intro bij-betwI)
  have Dom: RTS.Rts (RTS.dom (F_a a b)) = HOM_EC a b
    using a b F_a-in-hom [of a b] by auto
 have Cod: RTS.Rts (RTS.cod (F_a a b)) = R''.HOM_EC (F_o a) (F_o b)
    using a b F_a-in-hom [of a b] by auto
  interpret DOM'-alt: sub-rts R' ‹λt. H'.in-hom t (F_o a) (F_o b)›
    using a b ide-F_o R''.sub-rts-HOM by metis
  have Map: RTS.Map (F_a a b) =
            (λt. if residuation.arr (HOM_EC a b) t
                 then DN (MkArr a b t)
                 else ResiduatedTransitionSystem.partial-magma.null
                        (R''.HOM_EC (F_o a) (F_o b)))
    using RTS.bij-mkarr(3) a b ide-F_o by auto
show RTS.Map (F_a a b) ∈ Collect F_a-ab.A.arr → Collect F_a-ab.B.arr
    using F_a-ab.preserves-reflects-arr by blast
  let ?g = λt. if F_a-ab.B.arr t then Trn (UP t) else F_a-ab.A.null
  show g-mapsto: ?g ∈ Collect F_a-ab.B.arr → Collect F_a-ab.A.arr
  proof
    fix t
    assume t: t ∈ Collect F_a-ab.B.arr
    have gt: ?g t = Trn (UP t)
      using t by simp
    have arr (UP t)
      using a b t Cod UP-DN.F.preserves-reflects-arr
            R''.HOM-arr-char ide-F_o
      by fastforce
    moreover have Dom (UP t) = a
    proof -
      have 1: DOM'-alt.arr t
        using t * Cod by auto
      have Dom (UP t) = Dom (H.dom (UP t))
        using t
        by (simp add: H-dom-char UP-DN.F.extensional)
      also have ... = Dom (UP (H'.dom t))
        using t 1 H'-dom-char UP-DN.inv'-simp [of H.dom (UP t)]
              DOM'-alt.arr-char DOM'-alt.inclusion
        by auto
```

```
    also have ... = Dom (UP (F_o a))
      using 1 DOM'-alt.arr-char by auto
    also have ... = a
      using a Id-yields-horiz-ide
      by (simp add: H-ide-char horizontal-unit-def)
    finally show ?thesis by blast
  qed
  moreover have Cod (UP t) = b
  proof -
    have 1: DOM'-alt.arr t
      using t * Cod by auto
    have Cod (UP t) = Cod (H.cod (UP t))
      using t H-cod-char
      by (metis (no-types, lifting) Cod.simps(1) cod.extensional
      H-cod-simp UP-DN.F.extensional UP-DN.F.preserves-reflects-arr)
    also have ... = Cod (UP (H'.cod t))
      using t 1 H'-cod-char UP-DN.inv'-simp
            DOM'-alt.arr-char DOM'-alt.inclusion
      by auto
    also have ... = Cod (UP (F_o b))
      using 1 DOM'-alt.arr-char by auto
    also have ... = b
      using b Id-yields-horiz-ide
      by (simp add: H-ide-char horizontal-unit-def)
    finally show ?thesis by blast
  qed
  ultimately have residuation.arr (HOM_EC a b) (Trn (UP t))
    using arr-char by blast
  thus ?g t ∈ Collect F_a-ab.A.arr
    using gt Dom by simp
qed
show ⋀x. x ∈ Collect F_a-ab.A.arr ⟹ ?g (RTS.Map (F_a a b) x) = x
proof -
  fix x
  assume x: x ∈ Collect F_a-ab.A.arr
  have ?g (RTS.Map (F_a a b) x) = Trn (UP (DN (MkArr a b x)))
    using a b x F_a-ab.preserves-reflects-arr RTS.Map-mkarr
    apply auto[1]
    using Dom Map by force
  also have ... = x
    using Dom a b x arr-MkArr by auto
  finally show ?g (RTS.Map (F_a a b) x) = x by blast
qed
show ⋀y. y ∈ Collect F_a-ab.B.arr ⟹ RTS.Map (F_a a b) (?g y) = y
proof -
  fix y
  assume y: y ∈ Collect F_a-ab.B.arr
  have RTS.Map (F_a a b) (?g y) = DN (MkArr a b (Trn (UP y)))
    using a b y * DOM'-alt.null-char Map
```

```
        UP-DN.G.extensional arr-char
    by auto
  also have ... = y
  proof −
    have arr (UP y)
      using Cod R''.HOM-arr-char a b ide-F_o y by fastforce
    moreover have Dom (UP y) = a
    proof −
      have 1: DOM'-alt.arr y
        using y ∗ Cod by auto
      have Dom (UP y) = Dom (H.dom (UP y))
        using y
        by (simp add: H-dom-char UP-DN.F.extensional)
      also have ... = Dom (UP (H'.dom y))
        using y H'-dom-char UP-DN.inv'-simp
        apply auto[1]
        using 1 DOM'-alt.arr-char DOM'-alt.inclusion by force
      also have ... = Dom (UP (F_o a))
        using 1 DOM'-alt.arr-char by auto
      also have ... = a
        using a Id-yields-horiz-ide
        by (simp add: H-ide-char horizontal-unit-def)
      finally show ?thesis by blast
    qed
    moreover have Cod (UP y) = b
    proof −
      have 1: DOM'-alt.arr y
        using y ∗ Cod by auto
      have Cod (UP y) = Cod (H.cod (UP y))
        using y H-cod-char
        by (metis (no-types, lifting) Cod.simps(1) cod.extensional
     H-cod-simp UP-DN.F.extensional UP-DN.F.preserves-reflects-arr)
      also have ... = Cod (UP (H'.cod y))
        using y H'-cod-char UP-DN.inv'-simp
        apply auto[1]
        using 1 DOM'-alt.arr-char DOM'-alt.inclusion by simp
      also have ... = Cod (UP (F_o b))
        using 1 DOM'-alt.arr-char by auto
      also have ... = b
        using b Id-yields-horiz-ide
        by (simp add: H-ide-char horizontal-unit-def)
      finally show ?thesis by blast
    qed
    ultimately show ?thesis
      using a b y MkArr-Trn [of UP y] by simp
  qed
  finally show RTS.Map (F_a a b) (?g y) = y by blast
 qed
qed
```

```
      show ∀ t u. F_a-ab.B.con (RTS.Map (F_a a b) t) (RTS.Map (F_a a b) u)
                  ⟶ F_a-ab.A.con t u
      proof (intro allI impI)
        fix t u
      assume tu: F_a-ab.B.con (RTS.Map (F_a a b) t) (RTS.Map (F_a a b) u)
        have R'.con (DN (MkArr a b t)) (DN (MkArr a b u))
        proof −
          have F_a-ab.B.con (DN (MkArr a b t)) (DN (MkArr a b u))
          proof −
            have RTS.Map (F_a a b) t = DN (MkArr a b t)
          using a b R''.HOM-null-char RTS.bij-mkarr(3) UP-DN.G.extensional
                  arr-char ide-F_o
              by force
            moreover have RTS.Map (F_a a b) u = DN (MkArr a b u)
          using a b R''.HOM-null-char RTS.bij-mkarr(3) UP-DN.G.extensional
                  arr-char ide-F_o
              by force
            ultimately show ?thesis
              using tu by simp
          qed
          moreover have residuation.con (R''.HOM (F_o a) (F_o b)) =
                        F_a-ab.B.con
           using a b ide-F_o F_a-in-hom [of a b] R''.Hom-def RTS.bij-mkide(3)
            by auto
          ultimately show ?thesis
            using a b ide-F_o R''.sub-rts-HOM
                  sub-rts.con-char
                    [of R' λt. H'.in-hom t (F_o a) (F_o b)
                        DN (MkArr a b t) DN (MkArr a b u)]
            by auto
          qed
          hence V.con (MkArr a b t) (MkArr a b u)
            using UP-DN.G.reflects-con by auto
          thus F_a-ab.A.con t u
            using a b con-char F_a-in-hom [of a b] Con-def by auto
        qed
      qed
      thus ?thesis
        using a b F_a-in-hom RTS.iso-char by blast
    qed
  qed
qed

end
```

5.5 RTS† Determined by its Underlying Category

In this section we show that the category **RTS**† is fully determined by its subcategory **RTS** comprising the arrows that are identities for the residuation. Specifically, we show that there is an invertible RTS-functor from **RTS**† to the RTS-category obtained from the category **RTS** regarded as a category enriched in itself.

context *rtscat*
begin

The following produces a stand-alone instance of the category **RTS**†, independent of the current context. Arrows of **RTS**† have type $'A$ *rtscatx.arr* and they have the form *MkArr A B F*, where *A* and *B* have type $'A$ *resid* and *F* has the type ($'A$, $'A$) *exponential-rts.arr* of an arrow of the exponential RTS [*A*, *B*]).

interpretation *RTSx*: *rtscatx arr-type* **..**

In the current locale context, *comp* is the composition for the ordinary category **RTS**. As a cartesian closed category, this category determines a category enriched in itself.

interpretation *enriched-category comp Prod* α ι
‹*Collect ide*› *exp ECMC.Id ECMC.Comp*
using *extends-to-enriched-category* **by** *blast*

This self-enriched category determines an RTS-category, using the general construction defined in *rts-category-of-enriched-category*. We will refer to this RTS-category as **RC**. Arrows of **RC** have type ($'A$ *rtscatx.arr*, $'A$) *RC.arr* and they have the form *RC.MkArr a b t*, where *a* and *b* are objects of **RTS** and *t* is an arrow of the hom-RTS HOM_{EC} *a b*, which has arrow type $'A$.

interpretation *RC*: *rts-category-of-enriched-category*
arr-type ‹*Collect ide*› *exp ECMC.Id ECMC.Comp*
..

We now define the mapping Φ which we will show to be an RTS-category isomorphism from **RC** to **RTS**. In order to map an arrow *MkArr a b t* of **RC** to an arrow of **RTS**, it is necessary to use the invertible simulation *RTS.Func a b* to lift the arrow $t :: 'A$ of HOM_{EC} *a b* to an arrow *RTS.Func a b t* :: ($'A$, $'A$) *exponential-rts.arr* of the exponential RTS [*RTSx.Rts a*, *RTSx.Rts b*].

definition Φ :: ($'A$ *rtscatx.arr*, $'A$) *RC.arr* $\Rightarrow$ $'A$ *rtscatx.arr*
where Φ *t* $\equiv$ *if RC.arr t*
then RTSx.MkArr
(*RTSx.Dom* (*RC.Dom t*)) (*RTSx.Dom* (*RC.Cod t*))
(*Func* (*RC.Dom t*) (*RC.Cod t*) (*RC.Trn t*))
else RTSx.null

lemma Φ-*simps* [*simp*]:
assumes *RC.arr t*
shows *RTSx.arr* (Φ *t*)
and *RTSx.dom* (Φ *t*) = *RTSx.mkobj* (*RTSx.Dom* (*RC.Dom t*))
and *RTSx.cod* (Φ *t*) = *RTSx.mkobj* (*RTSx.Dom* (*RC.Cod t*))
proof −
  show *1*: *RTSx.arr* (Φ *t*)
    unfolding Φ-*def Rts-def*
    using *assms RTSx.null-char RC.arr-char simulation-Func*
        Rts-def Func-def ideD$_{RTSC}$
    apply (*intro RTSx.arrI*)
      apply *auto*[*3*]
    apply *simp*
    by (*metis Rts-def simulation.preserves-reflects-arr*)
  show *RTSx.dom* (Φ *t*) = *RTSx.mkobj* (*RTSx.Dom* (*RC.Dom t*))
    using *assms 1* Φ-*def RC.arr-char RTSx.dom-char* **by** *simp*
  show *RTSx.cod* (Φ *t*) = *RTSx.mkobj* (*RTSx.Dom* (*RC.Cod t*))
    using *assms 1* Φ-*def RC.arr-char RTSx.cod-char* **by** *simp*
qed

lemma Φ-*in-hom* [*intro*]:
assumes *RC.arr t*
shows *RTSx.H.in-hom* (Φ *t*)
        (*RTSx.mkobj*
          (*RTSx.Dom* (*RC.Dom t*))) (*RTSx.mkobj* (*RTSx.Dom* (*RC.Cod t*)))
  using *assms* **by** *auto*

interpretation Φ: *simulation RC.resid RTSx.resid* Φ
proof
  show ⋀*t*. ¬ *RC.arr t* ⟹ Φ *t* = *RTSx.null*
    using Φ-*def* **by** *auto*
  fix *t u*
  assume *tu*: *RC.V.con t u*
  have *t*: *RC.arr t* **and** *u*: *RC.arr u*
    using *tu RC.V.con-implies-arr* **by** *auto*
  have *0*: *RC.Dom t* = *RC.Dom u* ∧ *RC.Cod t* = *RC.Cod u*
    using *RC.con-implies-Par*(*1*−*2*) *tu* **by** *blast*
  interpret *Func*: *simulation* ‹*RC.HOM*$_{EC}$ (*RC.Dom t*) (*RC.Cod t*)›
                ‹*exponential-rts.resid*
                  (*RTSx.Dom* (*RC.Dom t*)) (*RTSx.Dom* (*RC.Cod t*))›
                ‹*Func* (*RC.Dom t*) (*RC.Cod t*)›
    using *t simulation-Func RC.arr-char*
    by (*simp add*: *Rts-def Func-def*)
  show *1*: *RTSx.V.con* (Φ *t*) (Φ *u*)
    using *tu RTSx.con-char RC.V.con-implies-arr RTSx.arr-char* Φ-*simps*(*1*)
        RC.con-implies-Par Φ-*def*
    apply *auto*[*1*]
     apply *metis*

```
    by (metis Func.preserves-con RC.Con-def RC.con-char)
  show Φ (RC.resid t u) = RTSx.resid (Φ t) (Φ u)
    using t u tu 1 Φ-def
    apply simp
    apply (intro conjI)
          apply (metis (no-types, lifting) Func.preserves-resid
       RC.ConE RC.con-char)
    using RTSx.con-char by auto
qed
```

The following fact is key to showing that Φ is functorial.

```
lemma Func-Trn-obj:
assumes RC.obj a
shows Func (RC.Dom a) (RC.Cod a) (RC.Trn a) =
      exponential-rts.MkIde (I (Rts (RC.Dom a)))
proof −
  have a: ide (RC.Dom a)
    using assms RC.H.ideD(1) RC.H-arr-char by auto

  interpret Dom: extensional-rts ‹RTSx.Dom (RC.Dom a)›
    using assms Rts-def ideD_RTSC(1) RC.H.ideD(1) RC.H-arr-char by simp
  interpret I-Dom: identity-simulation ‹RTSx.Dom (RC.Dom a)› ..
  interpret Exp0: exponential-rts
                ‹RTSx.Dom (RC.Dom a)› ‹RTSx.Dom (RC.Dom a)›
    ..
  interpret DOM: extensional-rts ‹RC.HOM_EC (RC.Dom a) (RC.Dom a)›
    using assms
    by (simp add: RC.H-ide-char RC.arr-char RC.horizontal-unit-def)
  have RC.Trn a = RC.Trn (RC.mkobj (RC.Dom a))
    using assms RC.H.ide-char [of a] RC.arr-char [of a] RC.H-dom-char
    by force
 also have ... = Exp0.Map (RTSx.Trn (ECMC.Id (RC.Dom a))) One.the-arr
  proof −
    have RC.mkobj (RC.Dom a) =
         RC.MkArr (RC.Dom a) (RC.Dom a)
           (RTSx.Map (ECMC.Id (RC.Dom a)) One.the-arr)
      using Map-def by argo
    thus ?thesis by simp
  qed
  also have ... = Exp0.Map
                    (RTSx.Trn
                      (curry CMC.unity (RC.Dom a) (RC.Dom a)
                        (CMC.lunit (RC.Dom a))))
                    One.the-arr
    using ECMC.Id-def by (simp add: curry-def)
  also have ... = (Unfunc (RC.Dom a) (RC.Dom a) ∘
                   Currying.Curry
                     (Cod ι) (RTSx.Dom (RC.Dom a))
                       (RTSx.Dom (RC.Dom a))
```

```
                    (RTSx.Src (CMC.lunit (RC.Dom a)) ∘
                      Pack CMC.unity (RC.Dom a))
                    (RTSx.Trg (CMC.lunit (RC.Dom a)) ∘
                      Pack CMC.unity (RC.Dom a))
                    (RTSx.Map (CMC.lunit (RC.Dom a)) ∘
                      Pack CMC.unity (RC.Dom a)))
                    One.the-arr
  using CMC.ide-unity ECMC.Id-def Pack-def RTSx.Map-curry Rts-def
        Unfunc-def a ide-iff-RTS-obj local.curry-def
  by force
also have ... = (Unfunc (RC.Dom a) (RC.Dom a) ∘
                  Currying.Curry3
                    (Cod ι) (RTSx.Dom (RC.Dom a))
                      (RTSx.Dom (RC.Dom a))
                    (product-rts.P₀
                      (RTSx.Dom RTSx.one) (RTSx.Dom (RC.Dom a))))
                    One.the-arr
proof −
  have RTSx.Map (CMC.lunit (RC.Dom a)) ∘
         Pack CMC.unity (RC.Dom a) =
       product-rts.P₀ (RTSx.Dom RTSx.one) (RTSx.Dom (RC.Dom a))
  proof −
    have RTSx.Map (CMC.lunit (RC.Dom a)) ∘
           RTSx.Pack CMC.unity (RC.Dom a) =
         RTSx.Map (CMC.pr₀ RTSx.one (RC.Dom a)) ∘
           RTSx.Pack CMC.unity (RC.Dom a)
    using assms CMC.lunit-eq RC.H.ide-char RC.H-arr-char unity-agreement
      by (metis (no-types, lifting) mem-Collect-eq one-def)
    also have ... =  product-rts.P₀
                       (RTSx.Dom RTSx.one) (RTSx.Dom (RC.Dom a)) ∘
                       (Unpack RTSx.one (RC.Dom a) ∘
                         Pack CMC.unity (RC.Dom a))
    using assms RTSx.Map-p₀ RC.H.ideD(1) RC.H-arr-char pr-agreement(1)
          ide-iff-RTS-obj
      by (auto simp add: one-def p₀-def Pack-def Unpack-def)
    also have ... = product-rts.P₀
                      (RTSx.Dom RTSx.one) (RTSx.Dom (RC.Dom a)) ∘
                      I (product-rts.resid
                           (RTSx.Dom (RTSx.one)) (RTSx.Dom (RC.Dom a)))
      using assms a one-def RTSx.obj-one ide-iff-RTS-obj ide-one
            RTSx.Unpack-o-Pack
      by (auto simp add: one-def Pack-def Unpack-def)
    also have ... = product-rts.P₀
                      (RTSx.Dom RTSx.one) (RTSx.Dom (RC.Dom a))
      using assms one-def
            comp-simulation-identity
              [of product-rts.resid
                    (RTSx.Dom (RTSx.one)) (RTSx.Dom (RC.Dom a))
                  RTSx.Dom (RC.Dom a)
```

```
                product-rts.P₀ (RTSx.Dom RTSx.one)
                  (RTSx.Dom (RC.Dom a))]
    by (metis (no-types, lifting) Dom.rts-axioms Rts-def extensional-rts-def
        ideD_RTSC ide-one product-rts.P₀-is-simulation product-rts.intro)
   finally show ?thesis
    unfolding Pack-def by blast
 qed
 moreover have 1: RTSx.sta (CMC.lunit (RC.Dom a))
  using assms CMC.arr-lunit RC.H.ideD(1) RC.H-arr-char arr-iff-RTS-sta
   by force
 moreover have RTSx.Src (CMC.lunit (RC.Dom a)) =
        RTSx.Map (CMC.lunit (RC.Dom a))
 using assms 1 RTSx.src-char RTSx.sta-char RTSx.Map-simps(3) RTSx.V.src-ide
   by (metis (no-types, lifting))
 moreover have RTSx.Trg (CMC.lunit (RC.Dom a)) =
        RTSx.Map (CMC.lunit (RC.Dom a))
 using assms 1 RTSx.trg-char RTSx.sta-char RTSx.Map-simps(4) RTSx.V.trg-ide
   by (metis (no-types, lifting))
 ultimately show ?thesis
   by force
qed
also have ... = RTSx.Unfunc (RC.Dom a) (RC.Dom a)
                  (Exp0.MkIde (I (RTSx.Dom (RC.Dom a))))
proof −
 interpret Cod-ι: extensional-rts ‹Cod ι›
   using CMC.ide-unity extensional-rts-def by simp
 interpret C: Currying
              ‹Cod ι› ‹RTSx.Dom (RC.Dom a)› ‹RTSx.Dom (RC.Dom a)›
   ..
 have Currying.Curry3
        (Cod ι) (RTSx.Dom (RC.Dom a)) (RTSx.Dom (RC.Dom a))
        (product-rts.P₀ (RTSx.Dom RTSx.one) (RTSx.Dom (RC.Dom a)))
        One.the-arr =
      Exp0.MkIde (I (RTSx.Dom (RC.Dom a)))
 proof −
   interpret P: product-rts ‹RTSx.Dom one› ‹RTSx.Dom (RC.Dom a)›
     using C.AxB.product-rts-axioms Rts-def unity-agreement by argo
  have 1: Cod-ι.arr = One.arr ∧ Cod-ι.src = One.src ∧ Cod-ι.trg = One.trg
     by (simp add: unity-agreement)
   have Cod-ι.arr One.the-arr
     by (simp add: One.arr-char unity-agreement)
   moreover have (λg. P.P₀ (One.the-arr, g)) = I-Dom.map
     using P.P₀-def One.arr-char P.arr-char Rts-def Rts-one by auto
   ultimately show ?thesis
     using 1 One.src-char One.trg-char C.Curry-def
     by (auto simp add: one-def)
 qed
 thus ?thesis
   unfolding Unfunc-def
```

```
    using One.arr-char by auto
  qed
  finally
  have RC.Trn a =
       RTSx.Unfunc (RC.Dom a) (RC.Dom a) (Exp0.MkIde I-Dom.map)
    by blast
  thus ?thesis
    unfolding Rts-def
    using assms RTSx.Func-Unfunc Unfunc-def Exp0.ide-MkIde
          Exp0.ide-implies-arr I-Dom.simulation-axioms RC.H-ide-char
          RC.horizontal-unit-def a ide-iff-RTS-obj Func-def
    by auto
qed

lemma obj-Φ-obj:
assumes RC.obj a
shows RTSx.obj (Φ a)
proof −
  interpret Dom: extensional-rts ‹RTSx.Dom (RC.Dom a)›
    using assms RC.H.ideD(1) RC.H-arr-char Rts-def ideD_RTSC by force
  interpret I-Dom: identity-simulation ‹RTSx.Dom (RC.Dom a)› ..
  interpret Exp0: exponential-rts
                ‹RTSx.Dom (RC.Dom a)› ‹RTSx.Dom (RC.Dom a)› ..
  show ?thesis
    unfolding Φ-def
    using assms RC.H.ideD(1) Func-Trn-obj RTSx.mkobj-def Rts-def
          RTSx.obj-mkobj [of RTSx.Dom (RC.Dom a)]
    apply auto[1]
    by (metis (no-types, lifting) CollectD RC.H.ideD(1) RC.H-arr-char
        RC.H-ide-char RC.horizontal-unit-def Rts-def ideD_RTSC)
qed

interpretation Φ: functor RC.hcomp RTSx.hcomp Φ
proof
  fix f
  let ?a = RC.Dom f
  let ?b = RC.Cod f
  let ?A = RTSx.Dom ?a
  let ?B = RTSx.Dom ?b
  let ?ab = RTSx.exp ?a ?b
  show ¬ RC.H.arr f ⟹ Φ f = RTSx.H.null
    using Φ-def by force
  show RC.H.arr f ⟹ RTSx.H.arr (Φ f)
    using RC.arr-coincidence RTSx.arr-coincidence Φ.preserves-reflects-arr
    by force
  show RC.H.arr f ⟹ RTSx.dom (Φ f) = Φ (RC.dom f)
  proof −
    assume f: RC.H.arr f
    have 1: RC.Dom (RC.dom f) = ?a
```

```
    using f RC.H.ide-dom RC.H-dom-char by simp
  have 2: RC.Cod (RC.dom f) = ?a
    using f RC.H.ide-dom RC.H-dom-char by simp
  have 3: RTSx.dom (Φ f) = RTSx.mkobj ?A
    using f by simp
  have Φ (RC.dom f) = RTSx.mkobj ?A
  proof -
    have Φ (RC.dom f) =
          RTSx.MkArr
            (RTSx.Dom (RC.Dom (RC.dom f)))
            (RTSx.Dom (RC.Cod (RC.dom f)))
            (RTSx.Func (RC.Dom (RC.dom f)) (RC.Cod (RC.dom f))
              (RC.Trn (RC.dom f)))
      unfolding Φ-def Func-def
      using f by simp
    also have ... = RTSx.MkArr ?A ?A
                      (exponential-rts.MkArr (I ?A) (I ?A) (I ?A))
    proof -
      have RC.obj (RC.dom f)
        using f by simp
      thus ?thesis
        using 1 2 Func-Trn-obj [of RC.dom f]
        unfolding Func-def Rts-def by presburger
    qed
    also have ... = RTSx.mkobj ?A
      unfolding RTSx.mkobj-def
      using f by blast
    finally show ?thesis by blast
  qed
  moreover have RTSx.dom (Φ f) = RTSx.mkobj ?A
    using f by simp
  ultimately show RTSx.dom (Φ f) = Φ (RC.dom f)
    by simp
qed
show RC.H.arr f ⟹ RTSx.cod (Φ f) = Φ (RC.cod f)
proof -
  assume f: RC.H.arr f
  have 1: RC.Dom (RC.cod f) = ?b
    using f RC.H.ide-cod RC.H-cod-char by simp
  have 2: RC.Cod (RC.cod f) = ?b
    using f RC.H.ide-cod RC.H-cod-char by simp
  have 3: RTSx.cod (Φ f) = RTSx.mkobj ?B
    using f by simp
  have Φ (RC.cod f) = RTSx.mkobj ?B
  proof -
    have Φ (RC.cod f) =
          RTSx.MkArr
            (RTSx.Dom (RC.Dom (RC.cod f)))
            (RTSx.Dom (RC.Cod (RC.cod f)))
```

```
          (RTSx.Func (RC.Dom (RC.cod f)) (RC.Cod (RC.cod f))
            (RC.Trn (RC.cod f)))
      unfolding Φ-def Func-def
      using f by simp
    also have ... = RTSx.MkArr ?B ?B
                      (exponential-rts.MkArr (I ?B) (I ?B) (I ?B))
    proof −
      have RC.obj (RC.cod f)
        using f by simp
      thus ?thesis
        using 1 2 Func-Trn-obj [of RC.cod f]
        unfolding Func-def Rts-def by presburger
    qed
    also have ... = RTSx.mkobj ?B
      unfolding RTSx.mkobj-def
      using f by blast
    finally show ?thesis by blast
  qed
  moreover have RTSx.cod (Φ f) = RTSx.mkobj ?B
    using f by simp
  ultimately show RTSx.cod (Φ f) = Φ (RC.cod f)
    by simp
qed
fix g
let ?c = RC.Cod g
let ?C = RTSx.Dom ?c
let ?bc = RTSx.exp ?b ?c
let ?ac = RTSx.exp ?a ?c
show RC.H.seq g f ⟹ Φ (RC.hcomp g f) = RTSx.hcomp (Φ g) (Φ f)
proof −
  assume seq: RC.H.seq g f
  have 0: RC.H.arr f ∧ RC.H.arr g ∧ RC.dom g = RC.cod f
    using seq by blast
  interpret A: extensional-rts ?A
    using seq 0 RC.H-arr-char Rts-def ideD_RTSC by fastforce
  interpret B: extensional-rts ?B
    using seq 0 RC.H-arr-char Rts-def ideD_RTSC by fastforce
  interpret C: extensional-rts ?C
    using seq 0 RC.H-arr-char Rts-def ideD_RTSC by fastforce
  interpret AB: exponential-rts ?A ?B ..
  interpret BC: exponential-rts ?B ?C ..
  interpret AC: exponential-rts ?A ?C ..
  interpret CMP: COMP ?A ?B ?C ..
  interpret ASC: ASSOC BC.resid AB.resid ?A ..
  interpret HOM-ab: extensional-rts ‹RC.HOM_EC ?a ?b›
    by (meson RC.H-seq-char RC.rts-category-of-enriched-category-axioms
        ideD_RTSC(1) ide-Hom rts-category-of-enriched-category.arr-char seq)
  interpret HOM-bc: extensional-rts ‹RC.HOM_EC ?b ?c›
    by (meson RC.H-seq-char RC.rts-category-of-enriched-category-axioms
```

```
    ideD_RTSC(1) ide-Hom rts-category-of-enriched-category.arr-char seq)
 interpret HOM-ac: extensional-rts ‹RC.HOM_EC ?a ?c›
   by (meson RC.H-seq-char RC.rts-category-of-enriched-category-axioms
       ideD_RTSC(1) ide-Hom rts-category-of-enriched-category.arr-char seq)
 let ?Func-ab = RTSx.Func ?a ?b
 let ?Func-bc = RTSx.Func ?b ?c
 let ?Func-ac = RTSx.Func ?a ?c
 interpret Func-ab: simulation ‹RC.HOM_EC ?a ?b› AB.resid ?Func-ab
   using 0 simulation-Func
by (metis (no-types, lifting) RC.H-arr-char Rts-def Func-def mem-Collect-eq)
 interpret Func-ab: simulation-between-extensional-rts
                ‹RC.HOM_EC ?a ?b› AB.resid ?Func-ab
   ..
 interpret Func-bc: simulation ‹RC.HOM_EC ?b ?c› BC.resid ?Func-bc
   using 0 simulation-Func
by (metis (no-types, lifting) RC.H-arr-char Rts-def Func-def mem-Collect-eq)
 interpret Func-bc: simulation-between-extensional-rts
                ‹RC.HOM_EC ?b ?c› BC.resid ?Func-bc
   ..
 interpret Func-ac: simulation ‹RC.HOM_EC ?a ?c› AC.resid ?Func-ac
   using 0 simulation-Func
by (metis (no-types, lifting) RC.H-arr-char Rts-def Func-def mem-Collect-eq)
 interpret Func-ac: simulation-between-extensional-rts
                ‹RC.HOM_EC ?a ?c› AC.resid ?Func-ac
   ..
 interpret Comp: Composition arr-type ?a ?b ?c
   using seq RC.arr-char RC.H-seq-char
   by unfold-locales auto
 interpret bcXab: product-rts ‹Comp.EXP ?b ?c› ‹Comp.EXP ?a ?b› ..
 interpret Func-bc-x-Func-ab:
           product-simulation ‹Comp.EXP ?b ?c› ‹Comp.EXP ?a ?b›
             BC.resid AB.resid ‹RTSx.Func ?b ?c› ‹RTSx.Func ?a ?b›
   ..
 interpret Eval-BC: RTSConstructions.evaluation-map ?B ?C ..
 interpret Eval-AB: RTSConstructions.evaluation-map ?A ?B ..
 interpret I-BC: identity-simulation BC.resid ..
 interpret I-BC-x-Eval-AB: product-simulation
                             BC.resid ASC.BxC.resid BC.resid ?B
                             I-BC.map Eval-AB.map
   ..
 have 0: bcXab.arr (RC.Trn g, RC.Trn f)
   using seq bcXab.arr-char RC.H.seqE RC.H-arr-char RC.H-seq-char
   by auto
 have 1: CMP.BCxAB.arr
           (RTSx.Func ?b ?c (RC.Trn g), RTSx.Func ?a ?b (RC.Trn f))
   using 0 by auto
 have Φ (RC.hcomp g f) =
      RTSx.MkArr
        (RTSx.Dom (RC.Dom (RC.hcomp g f)))
```

```
      (RTSx.Dom (RC.Cod (RC.hcomp g f)))
      (RTSx.Func (RC.Dom (RC.hcomp g f)) (RC.Cod (RC.hcomp g f))
        (RC.Trn (RC.hcomp g f)))
  using seq Φ-def by (auto simp add: Func-def)
also have ... =
      RTSx.MkArr ?A ?C
        ((?Func-ac ∘ AC.Map (RTSx.Trn (ECMC.Comp ?a ?b ?c)))
          (RTSx.Pack ?bc ?ab (RC.Trn g, RC.Trn f)))
  unfolding RC.hcomp-def
  using seq RC.H-seq-char
  by (auto simp add: Func-def Pack-def Map-def exp-def)
also have ... =
      RTSx.MkArr ?A ?C
        (CMP.map
          (Func-bc-x-Func-ab.map
            (RTSx.Unpack ?bc ?ab
              (RTSx.Pack ?bc ?ab (RC.Trn g, RC.Trn f)))))
  using Comp.Func-o-Map-Comp
  by (auto simp add: Func-def Unpack-def Map-def exp-def Rts-def)
also have ... =
      RTSx.MkArr ?A ?C
        (CMP.map
          (Func-bc-x-Func-ab.map
            (RC.Trn g, RC.Trn f)))
  using RC.H-seq-char RC.arr-char ide-Hom seq RTSx.Unpack-Pack
  by (metis (no-types, lifting) 0 Rts-def exp-def ide-iff-RTS-obj)
also have ... =
      RTSx.MkArr ?A ?C
        (CMP.map
          (RTSx.Func ?b ?c (RC.Trn g), RTSx.Func ?a ?b (RC.Trn f)))
  using 0 Func-bc-x-Func-ab.map-simp by fastforce
also have ... =
      RTSx.MkArr ?A ?C
        (CMP.Currying.A-BC.MkArr
          (BC.Dom (RTSx.Func ?b ?c (RC.Trn g)) ∘
            BC.Dom (RTSx.Func ?a ?b (RC.Trn f)))
          (BC.Cod (RTSx.Func ?b ?c (RC.Trn g)) ∘
            BC.Cod (RTSx.Func ?a ?b (RC.Trn f)))
          (BC.Map (RTSx.Func ?b ?c (RC.Trn g)) ∘
            BC.Map (RTSx.Func ?a ?b (RC.Trn f))))
  unfolding CMP.Currying.Curry-def
  using 0 CMP.map-eq by simp
also have ... =
      RTSx.MkArr ?A ?C
        (COMP.map ?A ?B ?C
          (RTSx.Func ?b ?c (RC.Trn g), RTSx.Func ?a ?b (RC.Trn f)))
  unfolding CMP.Currying.Curry-def
  using 0 CMP.map-eq by simp
also have ... = RTSx.hcomp (Φ g) (Φ f)
```

```
      unfolding RTSx.hcomp-def
      using seq RC.H-seq-char Φ-def Φ-simps(1)
      apply (auto simp add: Func-def)[1]
      by (metis (no-types, lifting))
    finally show ?thesis by blast
  qed
qed

interpretation Φ: rts-functor RC.resid RC.hcomp
                  RTSx.resid RTSx.hcomp Φ
  ..

interpretation Φ: fully-faithful-functor RC.hcomp RTSx.hcomp Φ
proof
  fix t u
  assume par: RC.H.par t u
  assume eq: Φ t = Φ u
  show t = u
  proof (intro RC.arr-eqI)
    show t ≠ RC.null and u ≠ RC.null
      using par by auto
    show 1: RC.Dom t = RC.Dom u and 2: RC.Cod t = RC.Cod u
      using par eq Φ-def RC.H-dom-char RC.H-cod-char by auto
    show RC.Trn t = RC.Trn u
    proof -
      have RC.Trn t = RTSx.Unfunc (RC.Dom t) (RC.Cod t)
                        (RTSx.Func (RC.Dom t) (RC.Cod t) (RC.Trn t))
        using par RTSx.Unfunc-Func RC.arr-char [of t]
        apply auto[1]
        by (simp add: Rts-def exp-def ide-iff-RTS-obj)
      also have ... = RTSx.Unfunc (RC.Dom u) (RC.Cod u)
                        (RTSx.Func (RC.Dom t) (RC.Cod t) (RC.Trn t))
        using 1 2 by auto
      also have ... = RTSx.Unfunc (RC.Dom u) (RC.Cod u)
                        (RTSx.Func (RC.Dom u) (RC.Cod u) (RC.Trn u))
        using par eq Φ-def
        by (auto simp add: Func-def)
      also have ... = RC.Trn u
        using par RTSx.Unfunc-Func RC.arr-char [of t] RC.arr-char [of u]
        apply auto[1]
        by (simp add: Rts-def exp-def ide-iff-RTS-obj)
      finally show RC.Trn t = RC.Trn u by blast
    qed
  qed
  next
  fix a b g
  assume a: RC.obj a and b: RC.obj b
  assume g: RTSx.H.in-hom g (Φ a) (Φ b)
  have 1: RTSx.dom g = RC.Dom a
```

```
  by (metis (no-types, lifting) CollectD RC.H.ide-char' RC.H-arr-char
    RC.H-ide-char RC.horizontal-unit-def RTSx.H.cod-dom RTSx.H.in-homE
      RTSx.bij-mkobj(4) Φ-simps(3) a g ide-iff-RTS-obj)
have 2: RTSx.cod g = RC.Dom b
  by (metis (no-types, lifting) CollectD RC.H-ide-char RC.arr-char
   RC.horizontal-unit-def RTSx.Dom.simps(1) RTSx.H.ide-cod RTSx.H.in-homE
      Rts-def Φ-def b bij-mkide(4) g ide-iff-RTS-obj)
interpret Dom-a: extensional-rts ‹RTSx.Dom (RC.Dom a)›
  using a Φ-def RC.H.ide-char [of a] RC.arr-char [of a] obj-Φ-obj
  by force
interpret Dom-b: extensional-rts ‹RTSx.Dom (RC.Dom b)›
  using b Φ-def RC.H.ide-char [of b] RC.arr-char [of b] obj-Φ-obj
  by force
interpret Exp: exponential-rts
              ‹RTSx.Dom (RC.Dom a)› ‹RTSx.Dom (RC.Dom b)›
  ..
interpret Unfunc: simulation
                    Exp.resid ‹RC.HOM_EC (RC.Dom a) (RC.Dom b)›
                    ‹RTSx.Unfunc (RC.Dom a) (RC.Dom b)›
  by (metis 1 2 RTSx.H.arrI RTSx.H.ide-cod RTSx.H.ide-dom
      RTSx.simulation-Unfunc Rts-def exp-def g)
let ?f = RC.MkArr (RC.Dom a) (RC.Dom b)
          (RTSx.Unfunc (RC.Dom a) (RC.Dom b) (RTSx.Trn g))
have RC.H.in-hom ?f a b ∧ Φ ?f = g
proof
  show 3: RC.H.in-hom ?f a b
  proof
    show 4: RC.H.arr ?f
      using a b g 1 2
            RTSx.arr-char RC.arr-char Unfunc.preserves-reflects-arr
            RC.arr-MkArr
      by (metis (no-types, lifting) RC.H.ideD(1) RC.H-arr-char
     RTSx.Dom-cod RTSx.Dom-dom RTSx.H.in-homE RTSx.arr-coincidence_CRC)
    show RC.dom ?f = a
      using 4 RC.H.ideD(1−2) RC.H-dom-char a by auto
    show RC.cod ?f = b
      using 4 RC.H.ideD(1−3) RC.H-dom-char RC.H-cod-char b by auto
  qed
  show Φ ?f = g
  proof −
    have RTSx.Func (RC.Dom a) (RC.Dom b)
           (RTSx.Unfunc (RC.Dom a) (RC.Dom b) (RTSx.Trn g)) =
         RTSx.Trn g
      using 1 2 RTSx.Func-Unfunc
   by (metis (no-types, lifting) RTSx.Dom-cod RTSx.Dom-dom RTSx.H.in-homE
        RTSx.arr-char RTSx.arr-coincidence_CRC a b g obj-Φ-obj)
    thus ?thesis
      unfolding Φ-def
      using g 1 2 3 RTSx.arr-MkArr RTSx.arr-char [of g]
```

```
      apply (auto simp add: Func-def)[1]
    by (metis (no-types, lifting) RTSx.Dom-cod RTSx.Dom-dom RTSx.MkArr-Trn)
   qed
  qed
  thus ∃f. RC.H.in-hom f a b ∧ Φ f = g by blast
qed

interpretation Φ: full-embedding-functor RC.hcomp RTSx.hcomp Φ
proof
  fix f f'
  assume f: RC.H.arr f and f': RC.H.arr f'
  assume eq: Φ f = Φ f'
  have RC.H.par f f'
   using f f' eq bij-mkide(4) RC.H-cod-char RC.H-dom-char
   apply (auto simp add: Φ-def Rts-def)[1]
   by (metis (no-types, lifting) CollectD RC.H-arr-char f f')+
  thus f = f'
   using Φ.is-faithful eq by blast
qed

interpretation Φ: invertible-functor RC.hcomp RTSx.hcomp Φ
proof −
  have Collect RTSx.obj ⊆ Φ ‘ Collect RC.obj
  proof
   fix a
   assume a: a ∈ Collect RTSx.obj
   show a ∈ Φ ‘ Collect RC.obj
   proof
    let ?a' = RC.mkobj a
    show a': ?a' ∈ Collect RC.obj
     using a RC.Id-yields-horiz-ide ide-iff-RTS-obj by blast
    show a = Φ ?a'
     using a a' bij-mkide(4)
     apply (auto simp add: Φ-def)[1]
      apply (metis (no-types, lifting) RC.Cod.simps(1) RC.Dom.simps(1)
     RC.H.ide-char RC.Trn.simps(1) RTSx.Dom.simps(1) RTSx.bij-mkobj(4)
         RTSx.dom-char Φ.as-nat-trans.preserves-dom
         Φ.as-nat-trans.preserves-reflects-arr Φ-def)
     by (metis (no-types, lifting) RC.H.ide-char RC.arr-coincidence)
   qed
  qed
  thus invertible-functor RC.hcomp RTSx.hcomp Φ
   using Φ.is-invertible-if-surjective-on-objects(1) by blast
qed

interpretation Φ: invertible-simulation RC.resid RTSx.resid Φ
proof −
  have ⋀t u. RTSx.V.con (Φ t) (Φ u) ⟹ RC.V.con t u
  proof −
```

```
      fix t u
      assume con: RTSx.V.con (Φ t) (Φ u)
      have 1: RTSx.V.con
                (RTSx.MkArr (RTSx.Dom (RC.Dom t)) (RTSx.Dom (RC.Cod t))
                  (RTSx.Func (RC.Dom t) (RC.Cod t) (RC.Trn t)))
               (RTSx.MkArr (RTSx.Dom (RC.Dom u)) (RTSx.Dom (RC.Cod u))
                  (RTSx.Func (RC.Dom u) (RC.Cod u) (RC.Trn u)))
        using con Φ-def
        unfolding Func-def
        by (metis RTSx.V.not-con-null(1) RTSx.V.not-con-null(2))
      hence residuation.con
              (exponential-rts.resid
                (RTSx.Dom (RC.Dom t)) (RTSx.Dom (RC.Cod t)))
              (RTSx.Func (RC.Dom t) (RC.Cod t) (RC.Trn t))
              (RTSx.Func (RC.Dom u) (RC.Cod u) (RC.Trn u))
        using RTSx.con-char by force
      hence residuation.con (RC.HOM_EC (RC.Dom t) (RC.Cod t))
              (RTSx.Unfunc (RC.Dom t) (RC.Cod t)
                (RTSx.Func (RC.Dom t) (RC.Cod t) (RC.Trn t)))
              (RTSx.Unfunc (RC.Dom u) (RC.Cod u)
                (RTSx.Func (RC.Dom u) (RC.Cod u) (RC.Trn u)))
        using con 1 simulation.preserves-con simulation-Unfunc bij-mkide(4)
        by (metis (no-types, lifting) CollectD RC.H-arr-char RTSx.Cod.simps(1)
            RTSx.Dom.simps(1) RTSx.con-char RTSx.con-implies-par Rts-def
            Unfunc-def Φ.as-nat-trans.preserves-reflects-arr)
      hence residuation.con (RC.HOM_EC (RC.Dom t) (RC.Cod t))
              (RC.Trn t) (RC.Trn u)
        using RC.arr-char RTSx.con-char Unfunc-Func
       by (metis (no-types, lifting) CollectD RC.H-arr-char RTSx.con-implies-par
            Unfunc-def Φ.as-nat-trans.preserves-reflects-arr con Func-def)
      thus RC.V.con t u
        using 1 con bij-mkide(4) RC.con-char RTSx.con-char RC.arr-char
          RTSx.con-implies-par RTSx.null-char Φ.extensional Rts-def bij-mkide(4)
        apply auto[1]
        apply (intro RC.ConI conjI, auto)
        by metis+
    qed
    thus invertible-simulation RC.resid RTSx.resid Φ
      using Φ.is-invertible-simulation-if Φ.invertible-functor-axioms by blast
  qed

  theorem rts-category-isomorphism RC.resid RC.hcomp
            RTSx.resid RTSx.hcomp Φ
    ..

end

end
```

Bibliography

[1] G. M. Kelly. Basic concepts of enriched category theory. *Reprints in Theory and Applications of Categories*, 10, 2005. http://www.tac.mta.ca/tac/reprints/articles/10/tr10.pdf.

[2] E. W. Stark. Category theory with adjunctions and limits. *Archive of Formal Proofs*, June 2016. https://isa-afp.org/entries/Category3.html, Formal proof development.

[3] E. W. Stark. Monoidal categories. *Archive of Formal Proofs*, May 2017. https://isa-afp.org/entries/MonoidalCategory.html, Formal proof development.

[4] E. W. Stark. Residuated transition systems. *Archive of Formal Proofs*, February 2022. https://isa-afp.org/entries/ResiduatedTransitionSystem.html, Formal proof development.