

Residuated Transition Systems II: Categorical Properties

Eugene W. Stark

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

February 6, 2026

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 \mathbf{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 \mathbf{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 \mathbf{RTS} , and that the RTS-category \mathbf{RTS}^\dagger is determined up to isomorphism by its cartesian closed subcategory \mathbf{RTS} .

Contents

Contents	2
Introduction	3
1 Preliminaries	8
1.1 Simulations	8
1.2 Transformations	14
1.3 Binary Simulations	18
1.4 Horizontal Composite of Transformations	19
2 RTS's as Categories	21
2.1 Categories with Bounded Pushouts	21
2.1.1 Bounded Spans	21
2.1.2 Pushouts	22
2.2 Categories of Transitions	24
2.3 Extensional RTS's with Composites as Categories	25
2.4 Characterization	27
3 RTS Constructions	31
3.1 Notation	31
3.2 Some Constraints on a Type	32
3.2.1 Nondegenerate	32
3.2.2 Lifting	32
3.2.3 Pairing	32
3.2.4 Exponentiation	34
3.2.5 Universe	35
3.3 Small RTS's	36
3.4 Injective Images of RTS's	37
3.5 Empty RTS	40
3.6 One-Transition RTS	43
3.7 Fibered Product RTS	45
3.8 Product RTS	50
3.8.1 Associators	55
3.9 Exponential RTS	56

3.9.1	Exponential of Small RTS's	65
3.9.2	Exponential into RTS with Composites	66
3.9.3	Exponential by One	67
3.9.4	Evaluation Map	68
3.9.5	Currying	68
3.9.6	Currying and Uncurrying as Inverse Simulations	71
3.9.7	Coextension of a Simulation	74
3.9.8	Compositors	76
3.9.9	Functoriality of Exponential	77
4	RTS's in Categories	81
4.1	RTS-Categories	81
4.1.1	Definition and Basic Properties	81
4.1.2	Hom-RTS's	85
4.1.3	Additional Notions	87
4.2	Concrete RTS-Categories	88
4.3	The RTS-Category of RTS's and Transformations	98
4.3.1	Terminal Object	104
4.3.2	Products	107
4.3.3	Exponentials	114
4.3.4	Cartesian Closure	124
4.3.5	Repleteness	125
4.4	The Category of RTS's and Simulations	127
4.4.1	Terminal Object	131
4.4.2	Products	132
4.4.3	Exponentials	135
4.4.4	Cartesian Closure	136
4.4.5	Associators	137
4.4.6	Compositors	139
4.5	Top-Level Interpretation	141
5	RTS-Enriched Categories	143
5.1	RTS-Enriched Categories	143
5.2	RTS-Enriched Categories induce RTS-Categories	146
5.2.1	The Small Case	154
5.2.2	Functoriality	158
5.3	RTS-Categories induce RTS-Enriched Categories	159
5.3.1	Functoriality	161
5.4	Equivalence of RTS-Enriched Categories and RTS-Categories	162
5.4.1	RTS-Category to Enriched Category to RTS-Category	162
5.4.2	Enriched Category to RTS-Category to Enriched Category	164
5.5	RTS [†] Determined by its Underlying Category	166

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 \mathbf{RTS} and prove facts about it as well, and finally establish the equivalence of RTS-categories and categories enriched in \mathbf{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
```

1.1 Simulations

```
abbreviation I
where I  $\equiv$  identity-simulation.map
```

```
lemma comp-identity-simulation:
assumes simulation A B F
shows I B  $\circ$  F = F
  <proof>
```

```
lemma comp-simulation-identity:
assumes simulation A B F
shows F  $\circ$  I A = F
  <proof>
```

```
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>
```

```
locale constant-simulation =
  A: rts A +
  B: rts B
for A :: 'a resid   (infix <\_A> 70)
and B :: 'b resid   (infix <\_B> 70)
```

```

and  $b :: 'b +$ 
assumes ide-b:  $B.ide\ b$ 
begin

  abbreviation map
  where  $map\ t \equiv if\ A.arr\ t\ then\ b\ else\ B.null$ 

  sublocale simulation  $A\ B\ map$ 
   $\langle proof \rangle$ 

  lemma is-simulation:
  shows simulation  $A\ B\ map$ 
   $\langle proof \rangle$ 

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  $\langle \backslash_A \rangle 70$ )
and  $B :: 'b\ resid$  (infix  $\langle \backslash_B \rangle 70$ )
and  $F :: 'b \Rightarrow 'a$ 
and  $G :: 'a \Rightarrow 'b +$ 
assumes inv:  $G\ o\ F = I\ B$ 
and inv':  $F\ o\ G = I\ A$ 
begin

  lemma inv-simp [simp]:
  assumes  $B.arr\ y$ 
  shows  $G\ (F\ y) = y$ 
   $\langle proof \rangle$ 

  lemma inv'-simp [simp]:
  assumes  $A.arr\ x$ 
  shows  $F\ (G\ x) = x$ 
   $\langle proof \rangle$ 

  lemma induce-bij-betw-arr-sets:
  shows bij-betw  $F\ (Collect\ B.arr)\ (Collect\ A.arr)$ 
   $\langle proof \rangle$ 

  lemma preserve-sources-exactly:
  assumes  $B.arr\ t$ 
  shows  $A.sources\ (F\ t) = F\ ' B.sources\ t$ 
   $\langle proof \rangle$ 

```

lemma *preserve-targets-exactly*:
assumes $B.\text{arr } t$
shows $A.\text{targets } (F t) = F \cdot B.\text{targets } t$
 $\langle \text{proof} \rangle$

lemma *preserve-weakly-extensional-rts*:
assumes *weakly-extensional-rts* A
shows *weakly-extensional-rts* B
 $\langle \text{proof} \rangle$

lemma *preserve-extensional-rts*:
assumes *extensional-rts* A
shows *extensional-rts* B
 $\langle \text{proof} \rangle$

lemma *preserve-rts-with-composites*:
assumes *rts-with-composites* A
shows *rts-with-composites* B
 $\langle \text{proof} \rangle$

lemma *preserve-rts-with-joins*:
assumes *rts-with-joins* A
shows *rts-with-joins* B
 $\langle \text{proof} \rangle$

end

lemma *inverse-simulations-sym*:
assumes *inverse-simulations* $A B F G$
shows *inverse-simulations* $B A G F$
 $\langle \text{proof} \rangle$

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

lemma *invertible-simulation-def'*:
shows *invertible-simulation* $A B F \longleftrightarrow (\exists G. \text{inverse-simulations } A B G F)$
 $\langle \text{proof} \rangle$

lemma *invertible-simulation-iff*:
shows *invertible-simulation* $A B F \longleftrightarrow$
simulation $A B F \wedge$
bij-betw $F (\text{Collect } (\text{residuation}.\text{arr } A)) (\text{Collect } (\text{residuation}.\text{arr } B)) \wedge$
 $(\forall t u. \text{residuation}.\text{con } B (F t) (F u) \longrightarrow \text{residuation}.\text{con } A t u)$
 $\langle \text{proof} \rangle$

context *invertible-simulation*

begin

lemma *is-bijection-betw-arr-sets:*
shows *bij-betw* F (*Collect* $A.arr$) (*Collect* $B.arr$)
 $\langle proof \rangle$

lemma *reflects-con:*
assumes *residuation.con* B (F t) (F u)
shows *residuation.con* A t u
 $\langle proof \rangle$

end

context *inverse-simulations*

begin

sublocale F : *invertible-simulation* B A F
 $\langle proof \rangle$

sublocale G : *invertible-simulation* A B G
 $\langle proof \rangle$

end

lemma *inverse-simulation-unique:*
assumes *inverse-simulations* A B G F
and *inverse-simulations* A B G' F
shows $G = G'$
 $\langle proof \rangle$

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

begin

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

interpretation *inverse-simulations* A B *map* F
 $\langle proof \rangle$

sublocale *simulation* B A *map* $\langle proof \rangle$

lemma *is-simulation:*
shows *simulation* B A *map*
 $\langle proof \rangle$

sublocale *inverse-simulations* A B *map* F $\langle proof \rangle$

lemma *map-simp*:
assumes $B.arr\ x$
shows $map\ x = inv-into\ (Collect\ A.arr)\ F\ x$
 $\langle proof \rangle$

lemma *map-eq*:
shows $map = (\lambda x. if\ B.arr\ x\ then\ inv-into\ (Collect\ A.arr)\ F\ x\ else\ A.null)$
 $\langle proof \rangle$

end

context *inverse-simulations*
begin

lemma *inverse-eq*:
shows $inverse-simulation.map\ A\ B\ G = F$
 $\langle proof \rangle$

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 $inverse-simulation.map\ A\ A\ (I\ A) = I\ A$
 $\langle proof \rangle$

lemma *inverse-simulations-compose*:
assumes $inverse-simulations\ A\ B\ F'\ F$ **and** $inverse-simulations\ B\ C\ G'\ G$
shows $inverse-simulations\ A\ C\ (F' \circ G')\ (G \circ F)$
 $\langle proof \rangle$

lemma *invertible-simulation-comp* [*intro*]:
assumes $invertible-simulation\ A\ B\ F$ **and** $invertible-simulation\ B\ C\ G$
shows $invertible-simulation\ A\ C\ (G \circ F)$
and $inverse-simulations\ A\ C$
 $(inverse-simulation.map\ A\ B\ F \circ inverse-simulation.map\ B\ C\ G)\ (G \circ F)$
and $inverse-simulation.map\ A\ C\ (G \circ F) =$
 $inverse-simulation.map\ A\ B\ F \circ inverse-simulation.map\ B\ C\ G$
 $\langle proof \rangle$

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 $inverse-simulations\ (product-rts.resid\ A\ C)\ (product-rts.resid\ B\ D)$
 $(product-simulation.map\ B\ D$
 $(inverse-simulation.map\ A\ B\ F)\ (inverse-simulation.map\ C\ D\ G))$

$(\text{product-simulation.map } A \ C \ F \ G)$
and $\text{inverse-simulation.map } (\text{product-rts.resid } A \ C) \ (\text{product-rts.resid } B \ D)$
 $(\text{product-simulation.map } A \ C \ F \ G) =$
 $\text{product-simulation.map } B \ D$
 $(\text{inverse-simulation.map } A \ B \ F) \ (\text{inverse-simulation.map } C \ D \ G)$
 $\langle \text{proof} \rangle$

lemma *invertible-simulation-cancel-left*:
assumes *invertible-simulation* $A \ B \ H$
shows $\llbracket \text{simulation } C \ A \ F; \text{simulation } C \ A \ G; H \circ F = H \circ G \rrbracket \implies F = G$
 $\langle \text{proof} \rangle$

lemma *invertible-simulation-cancel-right*:
assumes *invertible-simulation* $A \ B \ H$
shows $\llbracket \text{simulation } B \ C \ F; \text{simulation } B \ C \ G; F \circ H = G \circ H \rrbracket \implies F = G$
 $\langle \text{proof} \rangle$

definition *isomorphic-rts*
where *isomorphic-rts* $A \ B \equiv \exists F \ G. \text{inverse-simulations } A \ B \ G \ F$

lemma *isomorphic-rts-reflexive*:
assumes *rts* A
shows *isomorphic-rts* $A \ A$
 $\langle \text{proof} \rangle$

lemma *isomorphic-rts-symmetric*:
assumes *isomorphic-rts* $A \ B$
shows *isomorphic-rts* $B \ A$
 $\langle \text{proof} \rangle$

lemma *isomorphic-rts-transitive* [*trans*]:
assumes *isomorphic-rts* $A \ B$ **and** *isomorphic-rts* $B \ C$
shows *isomorphic-rts* $A \ C$
 $\langle \text{proof} \rangle$

lemma *isomorphism-cong-props*:
assumes *isomorphic-rts* $A \ B$
shows *weakly-extensional-cong-iso*: *weakly-extensional-rts* $A \implies \text{weakly-extensional-rts } B$
and *extensional-cong-iso*: *extensional-rts* $A \implies \text{extensional-rts } B$
and *rts-with-composites-cong-iso*: *rts-with-composites* $A \implies \text{rts-with-composites } B$
and *rts-with-joins-cong-iso*: *rts-with-joins* $A \implies \text{rts-with-joins } B$
 $\langle \text{proof} \rangle$

lemma (**in** *simulation*) *simulation-inv-intoI*:
assumes *inj-on* $F \ (\text{Collect } A.\text{arr})$ **and** $F \ ' (\text{Collect } A.\text{arr}) = \text{Collect } B.\text{arr}$
and $\bigwedge t. \neg B.\text{arr } t \implies \text{inv-into } (\text{Collect } A.\text{arr}) \ F \ t = A.\text{null}$
and $\bigwedge t \ u. t \sim_B u \implies$

$inv\text{-into } (Collect\ A.arr)\ F\ t \frown_A inv\text{-into } (Collect\ A.arr)\ F\ u$

shows *simulation* $B\ A\ (inv\text{-into } (Collect\ A.arr)\ F)$
 $\langle proof \rangle$

1.2 Transformations

lemma (*in transformation*) *preserves-arr*:

assumes $A.arr\ t$

shows $B.arr\ (\tau\ t)$

$\langle proof \rangle$

lemma (*in transformation*) *preserves-con*:

assumes $t \frown_A u$

shows $\tau\ t \frown_B \tau\ u$ **and** $\tau\ t \frown_B F\ u$

$\langle proof \rangle$

lemma (*in transformation*) *naturality1'*:

assumes $A.arr\ t$

shows $B.composite\text{-of } (F\ t)\ (\tau\ (A.trg\ t))\ (\tau\ t)$

$\langle proof \rangle$

lemma (*in transformation*) *naturality2'*:

assumes $A.arr\ t$

shows $B.composite\text{-of } (\tau\ (A.src\ t))\ (G\ t)\ (\tau\ t)$

$\langle proof \rangle$

locale *transformation-to-extensional-rts* =
transformation +

B : *extensional-rts* B

begin

notation $B.comp$ (**infixr** $\langle \cdot_B \rangle$ 55)

notation $B.join$ (**infixr** $\langle \sqcup_B \rangle$ 52)

lemma *naturality1'E*:

shows $F\ t \cdot_B \tau\ (A.trg\ t) = \tau\ t$

and $A.arr\ t \implies B.composable\ (F\ t)\ (\tau\ (A.trg\ t))$

$\langle proof \rangle$

lemma *naturality2'E*:

shows $\tau\ (A.src\ t) \cdot_B G\ t = \tau\ t$

and $A.arr\ t \implies B.composable\ (\tau\ (A.src\ t))\ (G\ t)$

$\langle proof \rangle$

lemma *naturality3'E*:

shows $\tau\ (A.src\ t) \sqcup_B F\ t = \tau\ t$

and $A.arr\ t \implies B.joinable\ (\tau\ (A.src\ t))\ (F\ t)$

$\langle proof \rangle$

lemma *naturality_E*:
shows $\tau (A.\text{src } t) \cdot_B G t = F t \cdot_B \tau (A.\text{trg } t)$
 $\langle \text{proof} \rangle$

lemma *general-naturality*:
assumes $A.\text{con } x y$
shows $\tau x \setminus_B F y = \tau (x \setminus_A y)$
and $F x \setminus_B \tau y = G (x \setminus_A y)$
 $\langle \text{proof} \rangle$

lemma *preserves-prfx*:
assumes $t \lesssim_A u$
shows $\tau t \lesssim_B \tau u$
 $\langle \text{proof} \rangle$

end

locale *transformation-by-components* =
 A : *rts* A +
 B : *extensional-rts* B +
 F : *simulation* $A B F$ +
 G : *simulation* $A B G$
for A :: 'a *resid* (**infix** $\langle \setminus_A \rangle$ 55)
and B :: 'b *resid* (**infix** $\langle \setminus_B \rangle$ 55)
and F :: 'a \Rightarrow 'b
and G :: 'a \Rightarrow 'b
and τ :: 'a \Rightarrow 'b +
assumes *preserves-src*: $A.\text{ide } a \Longrightarrow B.\text{src } (\tau a) = F a$
and *preserves-trg*: $A.\text{ide } a \Longrightarrow B.\text{trg } (\tau a) = G a$
and *respects-cong-ide*: $\llbracket A.\text{ide } a; A.\text{cong } a a' \rrbracket \Longrightarrow \tau a = \tau a'$
and *naturality1*: $a \in A.\text{sources } t \Longrightarrow \tau a \setminus_B F t = \tau (a \setminus_A t)$
and *naturality2*: $a \in A.\text{sources } t \Longrightarrow F t \setminus_B \tau a = G t$
and *joinable*: $a \in A.\text{sources } t \Longrightarrow B.\text{joinable } (\tau a) (F t)$
begin

notation $B.\text{comp}$ (**infixr** $\langle \cdot_B \rangle$ 55)
notation $B.\text{join}$ (**infixr** $\langle \sqcup_B \rangle$ 52)

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

lemma *map-eq*:
shows $\text{map } t = (\text{if } A.\text{arr } t \text{ then } \tau (A.\text{src } t) \sqcup_B F t \text{ else } B.\text{null})$
 $\langle \text{proof} \rangle$

lemma *map-simp-ide* [*simp*]:

assumes $A.ide\ a$
shows $map\ a = \tau\ a$
 $\langle proof \rangle$

sublocale *transformation* $A\ B\ F\ G\ map$
 $\langle proof \rangle$

lemma *is-transformation*:
shows *transformation* $A\ B\ F\ G\ map$
 $\langle proof \rangle$

end

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

lemma *src-eq-trg*:
shows $F = G$
 $\langle proof \rangle$

sublocale *simulation* $\langle proof \rangle$

end

lemma *comp-identity-transformation*:
assumes *transformation* $A\ B\ F\ G\ T$
shows $I\ B \circ T = T$
 $\langle proof \rangle$

lemma *comp-transformation-identity*:
assumes *transformation* $A\ B\ F\ G\ T$
shows $T \circ I\ A = T$
 $\langle proof \rangle$

locale *constant-transformation* =
 $A: rts\ A\ +$
 $B: weakly-extensional-rts\ B$
for $A :: 'a\ resid$ (**infix** $\langle \backslash_A \rangle\ 70$)
and $B :: 'b\ resid$ (**infix** $\langle \backslash_B \rangle\ 70$)
and $t :: 'b\ +$
assumes *arr-t*: $B.arr\ t$
begin

abbreviation *map*
where $map\ x \equiv if\ A.arr\ x\ then\ t\ else\ B.null$

abbreviation F

where $F \equiv \text{constant-simulation.map } A \ B \ (B.\text{src } t)$

abbreviation G

where $G \equiv \text{constant-simulation.map } A \ B \ (B.\text{trg } t)$

interpretation $F: \text{simulation } A \ B \ F$
 $\langle \text{proof} \rangle$

interpretation $G: \text{simulation } A \ B \ G$
 $\langle \text{proof} \rangle$

sublocale $\text{transformation } A \ B \ F \ G \ \text{map}$
 $\langle \text{proof} \rangle$

lemma is-transformation :
shows $\text{transformation } A \ B \ F \ G \ \text{map}$
 $\langle \text{proof} \rangle$

end

locale $\text{simulation-as-transformation} =$
 $\text{simulation} +$
 $B: \text{weakly-extensional-rts } B$

begin

sublocale $\text{transformation } A \ B \ F \ F \ F$
 $\langle \text{proof} \rangle$

sublocale $\text{identity-transformation } A \ B \ F \ F \ F$
 $\langle \text{proof} \rangle$

end

lemma $\text{transformation-eqI}$:
assumes $\text{transformation } A \ B \ F \ G \ \sigma$ **and** $\text{transformation } A \ B \ F \ H \ \tau$
and $\text{extensional-rts } B$
and $\bigwedge a. \text{residuation.ide } A \ a \implies \sigma \ a = \tau \ a$
shows $\sigma = \tau$
 $\langle \text{proof} \rangle$

lemma $\text{invertible-simulation-cancel-left'}$:
assumes $\text{invertible-simulation } A \ B \ H$
shows $\llbracket \text{transformation } C \ A \ F \ G \ S; \text{transformation } C \ A \ F \ G \ T;$
 $H \circ S = H \circ T \rrbracket$
 $\implies S = T$
 $\langle \text{proof} \rangle$

lemma $\text{invertible-simulation-cancel-right'}$:
assumes $\text{invertible-simulation } A \ B \ H$

shows $\llbracket \text{transformation } B \ C \ F \ G \ S; \text{transformation } B \ C \ F \ G \ T; \\ S \circ H = T \circ H \rrbracket \\ \implies S = T$
 $\langle \text{proof} \rangle$

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* $\langle \text{proof} \rangle$
sublocale *simulation-to-weakly-extensional-rts A.resid B F* $\langle \text{proof} \rangle$

lemma *fixing-arr-gives-transformation-1*:
assumes *A1.arr t1*
shows *transformation A0 B*
 $(\lambda t0. F (A1.src \ t1, \ t0)) (\lambda t0. F (A1.trg \ t1, \ t0))$
 $(\lambda t0. F (t1, \ t0))$
 $\langle \text{proof} \rangle$

lemma *fixing-arr-gives-transformation-0*:
assumes *A0.arr t2*
shows *transformation A1 B*
 $(\lambda t1. F (t1, \ A0.src \ t2)) (\lambda t1. F (t1, \ A0.trg \ t2))$
 $(\lambda t1. F (t1, \ t2))$
 $\langle \text{proof} \rangle$

end

locale *transformation-of-binary-simulations* =
A1: rts A1 +
A0: 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 $\langle \backslash_{A1} \rangle$ 55)
and *A0* :: '*a0* resid (infix $\langle \backslash_{A0} \rangle$ 55)
and *B* :: '*b* resid (infix $\langle \backslash_B \rangle$ 55)
and *F* :: '*a1* * '*a0* \Rightarrow '*b*
and *G* :: '*a1* * '*a0* \Rightarrow '*b*
and τ :: '*a1* * '*a0* \Rightarrow '*b*
begin

notation $A0.con$ (infix $\langle \frown_{A0} \rangle$ 50)
notation $A0.prfx$ (infix $\langle \lesssim_{A0} \rangle$ 50)
notation $A0.cong$ (infix $\langle \sim_{A0} \rangle$ 50)

notation $A1.con$ (infix $\langle \frown_{A1} \rangle$ 50)
notation $A1.prfx$ (infix $\langle \lesssim_{A1} \rangle$ 50)
notation $A1.cong$ (infix $\langle \sim_{A1} \rangle$ 50)

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

notation $A1xA0.resid$ (infix $\langle \backslash_{A1xA0} \rangle$ 55)

sublocale $A1xA0$: *product-rts* $A1 A0$ $\langle proof \rangle$

lemma *fixing-ide-gives-transformation-1*:
assumes $A1.ide$ $a1$
shows *transformation* $A0 B$ $(\lambda f0. F (a1, f0))$ $(\lambda f0. G (a1, f0))$
 $(\lambda f0. \tau (a1, f0))$
 $\langle proof \rangle$

lemma *fixing-ide-gives-transformation-0*:
assumes $A0.ide$ $a0$
shows *transformation* $A1 B$ $(\lambda f1. F (f1, a0))$ $(\lambda f1. G (f1, a0))$
 $(\lambda f1. \tau (f1, a0))$
 $\langle proof \rangle$

end

1.4 Horizontal Composite of Transformations

lemma *transformation-whisker-left*:
assumes *transformation* $A B F G \tau$ **and** *simulation* $B C H$
and *weakly-extensional-rts* C
shows *transformation* $A C$ $(H \circ F)$ $(H \circ G)$ $(H \circ \tau)$
 $\langle proof \rangle$

lemma *transformation-whisker-right*:
assumes *transformation* $B C F G \tau$ **and** *simulation* $A B H$
and *rts* A
shows *transformation* $A C$ $(F \circ H)$ $(G \circ H)$ $(\tau \circ H)$
 $\langle proof \rangle$

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* $B C F G \sigma$ **and** *transformation* $A B H K \tau$
and *extensional-rts* B **and** *extensional-rts* C

shows *transformation* $A \ C \ (F \circ H) \ (G \circ K) \ (\sigma \circ \tau)$
<proof>

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  $\equiv \exists f g. \text{commutative-square } f g h k$ 
```

```
lemma bounded-spanI [intro]:
assumes commutative-square f g h k
shows bounded-span h k
   $\langle \text{proof} \rangle$ 
```

```
lemma bounded-spanE [elim]:
assumes bounded-span h k
obtains f g where commutative-square f g h k
   $\langle \text{proof} \rangle$ 
```

lemma *bounded-span-sym*:
shows *bounded-span h k* \implies *bounded-span k h*
 ⟨*proof*⟩

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
 $(\forall f' g'. \text{commutative-square } f' g' h k \longrightarrow (\exists ! l. l \cdot f = f' \wedge l \cdot g = g'))$

lemma *pushout-squareI* [*intro*]:
assumes *cospan f g* **and** *span h k* **and** *dom f = cod h* **and** *f · h = g · k*
and $\bigwedge f' g'. \text{commutative-square } f' g' h k \implies \exists ! l. l \cdot f = f' \wedge l \cdot g = g'$
shows *pushout-square f g h k*
 ⟨*proof*⟩

lemma *composition-of-pushouts*:
assumes *pushout-square u' t' t u* **and** *pushout-square v' t'' t' v*
shows *pushout-square (v' · u') t'' t (v · u)*
 ⟨*proof*⟩

end

locale *category-with-bounded-pushouts* =
 category C
for *C* :: 'a *comp* (**infixr** · 55) +
assumes *has-bounded-pushouts*: *bounded-span h k* $\implies \exists f g. \text{pushout-square } f g h$
k

locale *elementary-category-with-bounded-pushouts* =
 category C
for *C* :: 'a *comp* (**infixr** · 55)
and *inj0* :: 'a \Rightarrow 'a \Rightarrow 'a (*i*₀[-, -])
and *inj1* :: 'a \Rightarrow 'a \Rightarrow 'a (*i*₁[-, -]) +
assumes *inj0-ext*: $\neg \text{bounded-span } h k \implies \text{i}_0[h, k] = \text{null}$

and *inj1-ext*: $\neg \text{bounded-span } h \ k \implies i_1[h, k] = \text{null}$
and *pushout-commutes* [*intro*]:
 $\text{bounded-span } h \ k \implies \text{commutative-square } i_1[h, k] \ i_0[h, k] \ h \ k$
and *pushout-universal*:
 $\text{commutative-square } f \ g \ h \ k \implies \exists !l. l \cdot i_1[h, k] = f \wedge l \cdot i_0[h, k] = g$
begin

lemma *dom-inj* [*simp*]:
assumes *bounded-span* $h \ k$
shows $\text{dom } i_0[h, k] = \text{cod } k$ **and** $\text{dom } i_1[h, k] = \text{cod } h$
 $\langle \text{proof} \rangle$

lemma *cod-inj*:
assumes *bounded-span* $h \ k$
shows $\text{cod } i_1[h, k] = \text{cod } i_0[h, k]$
 $\langle \text{proof} \rangle$

lemma *has-bounded-pushouts*:
assumes *bounded-span* $h \ k$
shows *pushout-square* $i_1[h, k] \ i_0[h, k] \ h \ k$
 $\langle \text{proof} \rangle$

sublocale *category-with-bounded-pushouts* C
 $\langle \text{proof} \rangle$

lemma *is-category-with-bounded-pushouts*:
shows *category-with-bounded-pushouts* C
 $\langle \text{proof} \rangle$

end

context *category-with-bounded-pushouts*
begin

definition *inj0* ($i_0[-, -]$)
where *inj0* $h \ k \equiv$ *if* *bounded-span* $h \ k$
 $\text{then } \text{fst } (\text{SOME } \text{inj. } \text{pushout-square } (\text{snd } \text{inj}) \ (\text{fst } \text{inj}) \ h \ k)$
 $\text{else } \text{null}$

definition *inj1* ($i_1[-, -]$)
where *inj1* $h \ k \equiv$ *if* *bounded-span* $h \ k$
 $\text{then } \text{snd } (\text{SOME } \text{inj. } \text{pushout-square } (\text{snd } \text{inj}) \ (\text{fst } \text{inj}) \ h \ k)$
 $\text{else } \text{null}$

lemma *extends-to-elementary-category-with-bounded-pushouts*:
shows *elementary-category-with-bounded-pushouts* C *inj0* *inj1*
 $\langle \text{proof} \rangle$

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 =
  category-with-bounded-pushouts +
assumes arr-implies-epi:  $arr\ t \implies epi\ t$ 
and iso-implies-ide:  $iso\ t \implies ide\ t$ 
begin
```

```
lemma commutative-square-sym:
shows commutative-square  $f\ g\ h\ k \implies commutative-square\ g\ f\ k\ h$ 
   $\langle proof \rangle$ 
```

In this setting, pushouts are uniquely determined.

```
sublocale elementary-category-with-bounded-pushouts  $C\ inj0\ inj1$ 
   $\langle proof \rangle$ 
```

```
lemma pushouts-unique:
assumes pushout-square  $f\ g\ h\ k$ 
shows  $f = i_1[h, k]$  and  $g = i_0[h, k]$ 
   $\langle proof \rangle$ 
```

```
lemma inj-sym:
shows  $i_0[k, h] = i_1[h, k]$ 
   $\langle proof \rangle$ 
```

```
lemma inj-arr-self:
assumes arr  $t$ 
shows  $i_0[t, t] = cod\ t$  and  $i_1[t, t] = cod\ t$ 
   $\langle proof \rangle$ 
```

```
lemma inj-arr-dom:
assumes arr  $t$ 
shows  $i_0[t, dom\ t] = t$  and  $i_1[t, dom\ t] = cod\ t$ 
   $\langle proof \rangle$ 
```

```
lemma eq-iff-ide-inj:
assumes span  $t\ u$ 
shows  $t = u \iff ide\ i_0[t, u] \wedge ide\ i_0[u, t]$ 
```

<proof>

lemma *inj-comp*:

assumes *bounded-span t (v · u)*

shows $i_0[t, v \cdot u] = i_0[i_0[t, u], v]$

and $i_0[v \cdot u, t] = i_0[v, i_0[t, u]] \cdot i_0[u, t]$

<proof>

lemma *inj-prefix*:

assumes *arr (u · t)*

shows $i_0[u \cdot t, t] = u$ **and** $i_0[t, u \cdot t] = \text{cod } u$

<proof>

end

lemma *category-of-transitions-preserved-by-iso*:

assumes *isomorphic-categories A B*

and *category-of-transitions A*

shows *category-of-transitions B*

<proof>

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.

abbreviation *comp*

where $\text{comp} \equiv \lambda u t. t \cdot u$

interpretation *Category.partial-magma comp*

<proof>

interpretation *Category.partial-composition comp* *<proof>*

lemma *null-char*:

shows $\text{null} = A.\text{null}$

<proof>

lemma *ide-char*:

shows $\text{ide } a \longleftrightarrow A.\text{ide } a$

<proof>

lemma *src-in-domains*:
assumes $A.arr\ t$
shows $A.src\ t \in domains\ t$
<proof>

lemma *trg-in-codomains*:
assumes $A.arr\ t$
shows $A.trg\ t \in codomains\ t$
<proof>

lemma *arr-char*:
shows $arr = A.arr$
<proof>

lemma *seq-char*:
shows $seq\ u\ t \longleftrightarrow A.seq\ t\ u$
<proof>

sublocale *category comp*
<proof>

proposition *is-category*:
shows *category comp*
<proof>

lemma *cod-char*:
shows $cod = A.trg$
<proof>

lemma *dom-char*:
shows $dom = A.src$
<proof>

lemma *arr-implies-epi*:
assumes $arr\ t$
shows $epi\ t$
<proof>

lemma *iso-implies-ide*:
assumes $iso\ t$
shows $ide\ t$
<proof>

end

locale *composite-completion-as-category* =
 $R: rts +$
extensional-rts-with-composites-as-category <composite-completion.resid resid>

2.4 Characterization

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

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

lemma *has-bounded-pushouts:*
assumes *bounded-span h k*
shows *pushout-square (k \ h) (h \ k) h k*
<proof>

sublocale *category-of-transitions comp*
<proof>

proposition *is-category-of-transitions:*
shows *category-of-transitions comp*
<proof>

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

abbreviation *resid*
where *resid* $\equiv \lambda h k. i_0[h, k]$

interpretation *ResiduatedTransitionSystem.partial-magma resid*
<proof>

lemma *null-char:*
shows *null = C.null*
<proof>

interpretation *residuation resid*
<proof>

lemma *con-char:*
shows *con t u* \longleftrightarrow *C.bounded-span t u*
<proof>

lemma *arr-char:*
shows *arr t* \longleftrightarrow *C.arr t*
<proof>

lemma *ide-char:*

shows $ide\ a \longleftrightarrow C.ide\ a$
<proof>

interpretation $rts\ \langle \lambda h\ k. i_0[h, k] \rangle$
<proof>

interpretation $extensional\text{-}rts\ resid$
<proof>

lemma $src\text{-}char$:
assumes $arr\ t$
shows $src\ t = C.dom\ t$
<proof>

lemma $trg\text{-}char$:
assumes $arr\ t$
shows $trg\ t = C.cod\ t$
<proof>

lemma $seq\text{-}char$:
shows $seq\ t\ u \longleftrightarrow C.seq\ u\ t$
<proof>

lemma $comp\text{-}char$:
shows $comp\ t\ u = u \cdot t$
<proof>

sublocale $extensional\text{-}rts\text{-}with\text{-}composites\ resid$
<proof>

proposition $is\text{-}extensional\text{-}rts\text{-}with\text{-}composites$:
shows $extensional\text{-}rts\text{-}with\text{-}composites\ resid$
<proof>

end

context $extensional\text{-}rts\text{-}with\text{-}composites\text{-}as\text{-}category$
begin

interpretation R : $underlying\text{-}rts\ comp$ *<proof>*

lemma $resid\text{-}char$:
shows $R.resid = resid$
<proof>

end

locale $pushout\text{-}preserving\text{-}functor =$
 $functor +$

assumes *preserves-pushouts*: $A.\text{pushout-square } f \ g \ h \ k$
 $\implies B.\text{pushout-square } (F \ f) \ (F \ g) \ (F \ h) \ (F \ k)$

lemma *pushout-preserving-functor-identity*:
assumes *category* A
shows *pushout-preserving-functor* $A \ A$ (*identity-functor.map* A)
 $\langle \text{proof} \rangle$

lemma *pushout-preserving-functor-comp*:
assumes *pushout-preserving-functor* $A \ B \ F$
and *pushout-preserving-functor* $B \ C \ G$
shows *pushout-preserving-functor* $A \ C$ ($G \circ F$)
 $\langle \text{proof} \rangle$

lemma *pushout-preserving-functor-const*:
assumes *category* A **and** *category* B
and *partial-composition.ide* $B \ b$
shows *pushout-preserving-functor* $A \ B$ (*constant-functor.map* $A \ B \ b$)
 $\langle \text{proof} \rangle$

lemma *invertible-functor-is-pushout-preserving*:
assumes *invertible-functor* $A \ B \ F$
shows *pushout-preserving-functor* $A \ B \ F$
 $\langle \text{proof} \rangle$

locale *simulation-as-functor* =
simulation +
 A : *extensional-rts-with-composites* A +
 B : *extensional-rts-with-composites* B
begin

sublocale $A.\text{as-category}$: *extensional-rts-with-composites-as-category* A $\langle \text{proof} \rangle$
sublocale $B.\text{as-category}$: *extensional-rts-with-composites-as-category* B $\langle \text{proof} \rangle$

sublocale *functor* $A.\text{as-category.comp } B.\text{as-category.comp } F$
 $\langle \text{proof} \rangle$

lemma *is-functor*:
shows *functor* $A.\text{as-category.comp } B.\text{as-category.comp } F$
 $\langle \text{proof} \rangle$

sublocale *pushout-preserving-functor* $A.\text{as-category.comp } B.\text{as-category.comp } F$
 $\langle \text{proof} \rangle$

lemma *is-pushout-preserving-functor*:
shows *pushout-preserving-functor* $A.\text{as-category.comp } B.\text{as-category.comp } F$
 $\langle \text{proof} \rangle$

end

lemma *simulation-is-pushout-preserving-functor:*

assumes *extensional-rts-with-composites A*

and *extensional-rts-with-composites B*

and *simulation A B F*

shows *pushout-preserving-functor*

(*extensional-rts-with-composites-as-category.comp A*)

(*extensional-rts-with-composites-as-category.comp B*)

F

<proof>

lemma *pushout-preserving-functor-is-simulation:*

assumes *category-of-transitions A*

and *category-of-transitions B*

and *pushout-preserving-functor A B F*

shows *simulation*

(*category-with-bounded-pushouts.inj0 A*)

(*category-with-bounded-pushouts.inj0 B*)

F

<proof>

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** $\langle \approx \rangle$ 50)
no-notation *Equipollence.lepoll* (**infixl** $\langle \lesssim \rangle$ 50)
no-notation *Lattices.sup-class.sup* (**infixl** $\langle \sqcup \rangle$ 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* :: $[p\text{trn}, p\text{atterns}] \Rightarrow p\text{trn}$ ($\langle \langle (-, -) \rangle \rangle$)

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*: $\exists l :: 'a \text{ option} \Rightarrow 'a. \text{inj } l$
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*
 $\langle \text{proof} \rangle$

A type that admits lifting is obviously nondegenerate.

sublocale *nondegenerate*
 $\langle \text{proof} \rangle$

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  $\equiv$   $x \in \text{range } \text{some-pair}$ 

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

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

  lemma inj-some-pair:
  shows inj some-pair
   $\langle \text{proof} \rangle$ 

  lemma first-conv:
  shows first (some-pair (x, y)) = x
   $\langle \text{proof} \rangle$ 

  lemma second-conv:
  shows second (some-pair (x, y)) = y
   $\langle \text{proof} \rangle$ 

  lemma pair-conv:
  assumes is-pair x
  shows some-pair (first x, second x) = x
   $\langle \text{proof} \rangle$ 

end

```

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

```

locale nondegenerate-and-pairing =
  nondegenerate + pairing
begin

  sublocale lifting type
   $\langle \text{proof} \rangle$ 

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* $:: ('a \Rightarrow 'b) \Rightarrow 'b \Rightarrow \text{bool}$
where *popular-value* $F y \equiv \neg \text{small } \{x. F x = y\}$

definition *some-popular-value* $:: ('a \Rightarrow 'b) \Rightarrow 'b$
where *some-popular-value* $F \equiv \text{SOME } y. \text{popular-value } F y$

abbreviation *at-most-one-popular-value*
where *at-most-one-popular-value* $F \equiv \exists_{\leq 1} y. \text{popular-value } F y$

definition *small-function*
where *small-function* $F \equiv \text{small } (\text{range } F) \wedge \text{at-most-one-popular-value } F$

lemma *small-preimage-unpopular*:
fixes $F :: 'a \Rightarrow 'b$
assumes *small-function* F
shows $\text{small } \{x. F x \neq \text{some-popular-value } F\}$
 $\langle \text{proof} \rangle$

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* $:: 'a \text{ itself}$
assumes *admits-exponentiation*:
 $\exists e :: ('a \Rightarrow 'a) \Rightarrow 'a. \text{inj-on } e \text{ (Collect small-function)}$
begin

definition *some-inj* $:: ('a \Rightarrow 'a) \Rightarrow 'a$
where *some-inj* $\equiv \text{SOME } e :: ('a \Rightarrow 'a) \Rightarrow 'a. \text{inj-on } e \text{ (Collect small-function)}$

lemma *inj-some-inj*:
shows $\text{inj-on } \text{some-inj} \text{ (Collect small-function)}$
 $\langle \text{proof} \rangle$

definition $app :: 'a \Rightarrow 'a \Rightarrow 'a$
where $app\ f \equiv inv\ into$
 $\{F. small\ (range\ F) \wedge$
 $at\ most\ one\ popular\ value\ F\} some\ inj\ f$

lemma $app\ some\ inj$:
assumes $small\ function\ F$
shows $app\ (some\ inj\ F) = F$
 $\langle proof \rangle$

lemma $some\ inj\ lam\ app$:
assumes $f \in some\ inj$ ‘*Collect small-function*
shows $some\ inj\ (\lambda x. app\ f\ x) = f$
 $\langle proof \rangle$

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 $exponentiation \langle TYPE(V) \rangle$
 $\langle proof \rangle$

lemma $V\ admits\ exponentiation$:
shows $exponentiation\ TYPE(V)$
 $\langle proof \rangle$

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 $nondegenerate \langle TYPE(V) \rangle$
 $\langle proof \rangle$

lemma $V\ is\ nondegenerate$:
shows $nondegenerate\ TYPE(V)$

⟨proof⟩

interpretation *pairing* ⟨TYPE(V)⟩
⟨proof⟩

lemma *V-admits-pairing*:
shows *pairing* TYPE(V)
⟨proof⟩

interpretation *exponentiation* ⟨TYPE(V)⟩
⟨proof⟩

interpretation *universe* ⟨TYPE(V)⟩
⟨proof⟩

lemma *V-is-universe*:
shows *universe* TYPE(V)
⟨proof⟩

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⟩

lemma *small-function-transformation*:
assumes *small-rts* *A* **and** *small-rts* *B* **and** *transformation* *A* *B* *F* *G* *T*
shows *small-function* *T*
⟨proof⟩

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⟩

lemma *small-function-resid*:
fixes *A* :: 'a *resid*

```

assumes small-rts A
shows small-function A
and  $\bigwedge t. \textit{small-function}$  (A t)
 $\langle \textit{proof} \rangle$ 

context exponentiation
begin

  lemma small-function-some-inj-resid:
  fixes A :: 'a resid
  assumes small-rts A
  shows small-function ( $\lambda t. \textit{some-inj}$  (A t))
   $\langle \textit{proof} \rangle$ 

  fun some-inj-resid :: 'a resid  $\Rightarrow$  'a
  where some-inj-resid A = (some-inj ( $\lambda t. \textit{some-inj}$  (A t)))

  lemma inj-on-some-inj-resid:
  shows inj-on some-inj-resid {A :: 'a resid. small-rts A}
   $\langle \textit{proof} \rangle$ 

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  $\Rightarrow$  'b
  and A :: 'a resid (infix  $\langle \backslash_A \rangle$  70) +
  assumes inj-map: inj-on map (Collect A.arr  $\cup$  {A.null})
begin

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

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

  abbreviation Arr
  where Arr t  $\equiv$  t  $\in$  map ' Collect A.arr

  abbreviation map'

```

where $\text{map}' t \equiv \text{inv-into } (\text{Collect } A.\text{arr}) \text{ map } t$

definition $\text{resid} :: 'b \text{ resid}$ (**infix** $\langle \backslash \rangle$ 70)

where $t \backslash u = (\text{if } \text{Arr } t \wedge \text{Arr } u \text{ then } \text{map } (\text{map}' t \backslash_A \text{map}' u) \text{ else } \text{Null})$

lemma $\text{inj-map}'$:

shows $\text{inj-on map } (\text{Collect } A.\text{arr})$

$\langle \text{proof} \rangle$

lemma map-null :

shows $\text{map } A.\text{null} \notin \text{map } ' \text{Collect } A.\text{arr}$

$\langle \text{proof} \rangle$

lemma $\text{map}'\text{-map}$ [*simp*]:

assumes $A.\text{arr } t$

shows $\text{map}' (\text{map } t) = t$

$\langle \text{proof} \rangle$

lemma $\text{map-map}'$ [*simp*]:

assumes $\text{Arr } t$

shows $\text{map } (\text{map}' t) = t$

$\langle \text{proof} \rangle$

sublocale $\text{ResiduatedTransitionSystem.partial-magma resid}$

$\langle \text{proof} \rangle$

lemma null-char :

shows $\text{null} = \text{Null}$

$\langle \text{proof} \rangle$

sublocale residuation resid

$\langle \text{proof} \rangle$

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

lemma con-char :

shows $t \frown u \iff \text{Arr } t \wedge \text{Arr } u \wedge \text{map}' t \frown_A \text{map}' u$

$\langle \text{proof} \rangle$

lemma arr-char :

shows $\text{arr } t \iff \text{Arr } t$

$\langle \text{proof} \rangle$

lemma ide-char_{II} :

shows $\text{ide } t \iff \text{Arr } t \wedge A.\text{ide } (\text{map}' t)$

$\langle \text{proof} \rangle$

lemma trg-char :

shows $\text{trg } t = (\text{if } \text{Arr } t \text{ then } \text{map } (A.\text{trg } (\text{map}' t)) \text{ else } \text{null})$

<proof>

sublocale *rts resid*

<proof>

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

notation *cong* (infix $\langle \sim \rangle$ 50)

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

abbreviation map_{ext}

where $\text{map}_{\text{ext}} t \equiv \text{if } A.\text{arr } t \text{ then } \text{map } t \text{ else null}$

abbreviation map'_{ext}

where $\text{map}'_{\text{ext}} t \equiv \text{if } \text{Arr } t \text{ then } \text{map}' t \text{ else } A.\text{null}$

sublocale *Map: simulation A resid map_{ext}*

<proof>

sublocale *Map': simulation resid A map'_{ext}*

<proof>

sublocale *inverse-simulations resid A map_{ext} map'_{ext}*

<proof>

lemma *invertible-simulation-map:*

shows *invertible-simulation A resid map_{ext}*

<proof>

lemma *invertible-simulation-map':*

shows *invertible-simulation resid A map'_{ext}*

<proof>

lemma *inj-on-map:*

shows *inj-on map_{ext} (Collect A.arr)*

<proof>

lemma *range-map':*

shows $\text{map}'_{\text{ext}} \text{ ` } (\text{Collect arr}) = \text{Collect } A.\text{arr}$

<proof>

lemma *cong-char_{II}:*

shows $t \sim u \iff \text{Arr } t \wedge \text{Arr } u \wedge \text{map}' t \sim_A \text{map}' u$

<proof>

lemma *preserves-weakly-extensional-rts:*

assumes *weakly-extensional-rts A*

shows *weakly-extensional-rts resid*

<proof>

```

lemma preserves-extensional-rts:
assumes extensional-rts A
shows extensional-rts resid
  ⟨proof⟩

lemma preserves-reflects-small-rts:
shows small-rts A  $\longleftrightarrow$  small-rts resid
  ⟨proof⟩

```

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⟩

```

```

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 inj-image-rts.map_ext ( $G \circ F$ )  $X =$ 
  inj-image-rts.map_ext  $G$  (inj-image-rts.resid  $F X$ )  $\circ$ 
  (inj-image-rts.map_ext  $F X$ )
and inj-image-rts.map'_ext ( $G \circ F$ )  $X =$ 
  inj-image-rts.map'_ext  $F X$   $\circ$ 
  inj-image-rts.map'_ext  $G$  (inj-image-rts.resid  $F X$ )
  ⟨proof⟩

```

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
    ⟨proof⟩

  lemma null-char:
  shows null = undefined

```

<proof>

sublocale *residuation resid*

<proof>

lemma *arr-char:*

shows *arr t \longleftrightarrow False*

<proof>

lemma *ide-char_{ERTS}:*

shows *ide t \longleftrightarrow False*

<proof>

lemma *con-char:*

shows *con t u \longleftrightarrow False*

<proof>

lemma *trg-char:*

shows *trg t = null*

<proof>

sublocale *rts resid*

<proof>

lemma *cong-char_{ERTS}:*

shows *cong t u \longleftrightarrow False*

<proof>

sublocale *small-rts resid*

<proof>

lemma *is-small-rts:*

shows *small-rts resid*

<proof>

sublocale *extensional-rts resid*

<proof>

lemma *is-extensional-rts:*

shows *extensional-rts resid*

<proof>

lemma *src-char:*

shows *src t = null*

<proof>

lemma *prfx-char:*

shows *prfx t u \longleftrightarrow False*

<proof>

lemma *composite-of-char*:
shows *composite-of t u v* \longleftrightarrow *False*
<proof>

lemma *composable-char*:
shows *composable t u* \longleftrightarrow *False*
<proof>

lemma *seq-char*:
shows *seq t u* \longleftrightarrow *False*
<proof>

sublocale *rts-with-composites resid*
<proof>

lemma *is-rts-with-composites*:
shows *rts-with-composites resid*
<proof>

sublocale *extensional-rts-with-composites resid* *<proof>*

lemma *is-extensional-rts-with-composites*:
shows *extensional-rts-with-composites resid*
<proof>

lemma *comp-char*:
shows *comp t u = null*
<proof>

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

definition *initiator* $:: 'e \text{ resid} \Rightarrow 'a \Rightarrow 'e$
where *initiator A* $\equiv (\lambda t. \text{ResiduatedTransitionSystem.partial-magma.null } A)$

lemma *initiator-is-simulation*:
assumes *rts A*
shows *simulation resid A (initiator A)*
<proof>

lemma *universality*:
assumes *rts A*
shows $\exists! F. \text{simulation resid } A \ F$
<proof>

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  $\equiv$  SOME t. t  $\neq$  undefined
```

```
definition resid :: 't resid (infix  $\langle \backslash_1 \rangle$  70)
where resid t u = (if t = the-arr  $\wedge$  u = the-arr then the-arr else undefined)
```

```
sublocale ResiduatedTransitionSystem.partial-magma resid
   $\langle$ proof $\rangle$ 
```

```
lemma null-char:
shows null = undefined
   $\langle$ proof $\rangle$ 
```

```
sublocale residuation resid
   $\langle$ proof $\rangle$ 
```

```
notation con (infix  $\langle \frown_1 \rangle$  50)
```

```
lemma arr-char:
shows arr t  $\longleftrightarrow$  t = the-arr
   $\langle$ proof $\rangle$ 
```

```
lemma ide-char1RTS:
shows ide t  $\longleftrightarrow$  t = the-arr
   $\langle$ proof $\rangle$ 
```

```
lemma con-char:
shows con t u  $\longleftrightarrow$  arr t  $\wedge$  arr u
   $\langle$ proof $\rangle$ 
```

```
lemma trg-char:
shows trg t = (if arr t then t else null)
   $\langle$ proof $\rangle$ 
```

```
sublocale rts resid
   $\langle$ proof $\rangle$ 
```

```
notation prfx (infix  $\langle \lesssim_1 \rangle$  50)
```

notation *cong* (infix $\langle \sim_1 \rangle$ 50)

lemma *cong-char_{1RTS}*:

shows $t \lesssim_1 u \longleftrightarrow \text{arr } t \wedge \text{arr } u$
<proof>

sublocale *extensional-rts resid*

<proof>

lemma *is-extensional-rts*:

shows *extensional-rts resid*
<proof>

lemma *src-char*:

shows $\text{src } t = (\text{if } t = \text{the-arr then } t \text{ else null})$
<proof>

lemma *prfx-char*:

shows $t \lesssim_1 u \longleftrightarrow \text{arr } t \wedge \text{arr } u$
<proof>

lemma *composite-of-char*:

shows *composite-of t u v* $\longleftrightarrow \text{arr } t \wedge \text{arr } u \wedge \text{arr } v$
<proof>

lemma *composable-char*:

shows *composable t u* $\longleftrightarrow \text{arr } t \wedge \text{arr } u$
<proof>

lemma *seq-char*:

shows *seq t u* $\longleftrightarrow \text{arr } t \wedge \text{arr } u$
<proof>

sublocale *rts-with-composites resid*

<proof>

lemma *is-rts-with-composites*:

shows *rts-with-composites resid*
<proof>

sublocale *extensional-rts-with-composites resid* *<proof>*

lemma *is-extensional-rts-with-composites*:

shows *extensional-rts-with-composites resid*
<proof>

sublocale *small-rts resid*

<proof>

lemma *is-small-rts*:
shows *small-rts resid*
 ⟨*proof*⟩

lemma *comp-char*:
shows $\text{comp } t \ u = (\text{if } \text{arr } t \wedge \text{arr } u \text{ then } \text{the-arr } \text{else } \text{null})$
 ⟨*proof*⟩

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

definition *terminator* $:: 'a \text{ resid} \Rightarrow 'a \Rightarrow 't$
where *terminator* $A \equiv (\lambda t. \text{if } \text{residuation.arr } A \ t \text{ then } \text{the-arr } \text{else } \text{null})$

lemma *terminator-is-simulation*:
assumes *rts* A
shows *simulation* $A \text{ resid } (\text{terminator } A)$
 ⟨*proof*⟩

lemma *universality*:
assumes *rts* A
shows $\exists! F. \text{simulation } A \text{ resid } F$
 ⟨*proof*⟩

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 $F = G \iff$
 $(\forall Q \ R \ T. \text{transformation } \text{resid } A \ Q \ R \ T \longrightarrow F \circ T = G \circ T)$
 ⟨*proof*⟩

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 $U = V \iff$
 $(\forall Q \ R \ T. \text{transformation } \text{resid } A \ Q \ R \ T \longrightarrow U \circ T = V \circ T)$
 ⟨*proof*⟩

end

3.7 Fibered Product RTS

locale *fibered-product-rts* =
 $A: \text{rts } A +$
 $B: \text{rts } B +$
 $C: \text{weakly-extensional-rts } C +$

F: simulation $A \ C \ F +$
G: simulation $B \ C \ G$
for $A :: 'a \ resid \ (\mathbf{infix} \ \langle \backslash_A \rangle \ 70)$
and $B :: 'b \ resid \ (\mathbf{infix} \ \langle \backslash_B \rangle \ 70)$
and $C :: 'c \ resid \ (\mathbf{infix} \ \langle \backslash_C \rangle \ 70)$
and $F :: 'a \Rightarrow 'c$
and $G :: 'b \Rightarrow 'c$
begin

notation $A.con \ (\mathbf{infix} \ \langle \frown_A \rangle \ 50)$
notation $B.con \ (\mathbf{infix} \ \langle \frown_B \rangle \ 50)$
notation $C.con \ (\mathbf{infix} \ \langle \frown_C \rangle \ 50)$
notation $A.prfx \ (\mathbf{infix} \ \langle \lesssim_A \rangle \ 50)$
notation $B.prfx \ (\mathbf{infix} \ \langle \lesssim_B \rangle \ 50)$
notation $C.prfx \ (\mathbf{infix} \ \langle \lesssim_C \rangle \ 50)$
notation $A.cong \ (\mathbf{infix} \ \langle \sim_A \rangle \ 50)$
notation $B.cong \ (\mathbf{infix} \ \langle \sim_B \rangle \ 50)$
notation $C.cong \ (\mathbf{infix} \ \langle \sim_C \rangle \ 50)$

abbreviation *Arr*
where $Arr \equiv \lambda tu. A.arr \ (fst \ tu) \wedge B.arr \ (snd \ tu) \wedge F \ (fst \ tu) = G \ (snd \ tu)$

abbreviation *Ide*
where $Ide \equiv \lambda tu. A.ide \ (fst \ tu) \wedge B.ide \ (snd \ tu) \wedge F \ (fst \ tu) = G \ (snd \ tu)$

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 $resid :: ('a * 'b) \ resid \ (\mathbf{infix} \ \langle \backslash \rangle \ 70)$
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*
 $\langle proof \rangle$

lemma *null-char*:
shows $null = (A.null, \ B.null)$
 $\langle proof \rangle$

sublocale *residuation resid*
 $\langle proof \rangle$

notation $con \ (\mathbf{infix} \ \langle \frown \rangle \ 50)$

lemma *arr-char*:
shows $arr \ t \longleftrightarrow Arr \ t$
 $\langle proof \rangle$

lemma *con-char*:

shows $t \frown u \longleftrightarrow \text{Con } t \ u$
<proof>

lemma *ide-char_{FP}*:

shows $\text{ide } t \longleftrightarrow \text{Ide } t$
<proof>

lemma *trg-char*:

shows $\text{trg } t = (\text{if arr } t \text{ then } (A.\text{trg } (fst \ t), B.\text{trg } (snd \ t)) \text{ else null})$
<proof>

sublocale *rts resid*

<proof>

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

notation *cong* (**infix** \sim 50)

lemma *prfx-char*:

shows $t \lesssim u \longleftrightarrow F (fst \ t) = G (snd \ t) \wedge F (fst \ u) = G (snd \ u) \wedge$
 $fst \ t \lesssim_A fst \ u \wedge snd \ t \lesssim_B snd \ u$
<proof>

lemma *cong-char_{FP}*:

shows $t \sim u \longleftrightarrow F (fst \ t) = G (snd \ t) \wedge F (fst \ u) = G (snd \ u) \wedge$
 $fst \ t \sim_A fst \ u \wedge snd \ t \sim_B snd \ u$
<proof>

lemma *sources-char*:

shows $\text{sources } t =$
 $\{a. F (fst \ t) = G (snd \ t) \wedge F (fst \ a) = G (snd \ a) \wedge$
 $fst \ a \in A.\text{sources } (fst \ t) \wedge snd \ a \in B.\text{sources } (snd \ t)\}$
<proof>

lemma *targets-char_{FP}*:

shows $\text{targets } t =$
 $\{a. F (fst \ t) = G (snd \ t) \wedge F (fst \ a) = G (snd \ a) \wedge$
 $fst \ a \in A.\text{targets } (fst \ t) \wedge snd \ a \in B.\text{targets } (snd \ t)\}$
<proof>

definition $P_0 :: 'a \times 'b \Rightarrow 'b$

where $P_0 \ t \equiv \text{if arr } t \text{ then snd } t \text{ else B.null}$

definition $P_1 :: 'a \times 'b \Rightarrow 'a$

where $P_1 \ t \equiv \text{if arr } t \text{ then fst } t \text{ else A.null}$

sublocale P_0 : *simulation resid B P₀*

<proof>

lemma *P₀-is-simulation*:
shows *simulation resid B P₀*
 ⟨*proof*⟩

sublocale *P₁: simulation resid A P₁*
 ⟨*proof*⟩

lemma *P₁-is-simulation*:
shows *simulation resid A P₁*
 ⟨*proof*⟩

lemma *commutativity*:
shows *F o P₁ = G o P₀*
 ⟨*proof*⟩

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 *P₁ o tuple X H K = H and P₀ o tuple X H K = K*
and *∃!HK. simulation X resid HK ∧ P₁ o HK = H ∧ P₀ o HK = K*
 ⟨*proof*⟩

lemma *preserves-weakly-extensional-rts*:
assumes *weakly-extensional-rts A and weakly-extensional-rts B*
shows *weakly-extensional-rts resid*
 ⟨*proof*⟩

lemma *preserves-extensional-rts*:
assumes *extensional-rts A and extensional-rts B*
shows *extensional-rts resid*
 ⟨*proof*⟩

lemma *preserves-small-rts*:
assumes *small-rts A and small-rts B*
shows *small-rts resid*
 ⟨*proof*⟩

end

locale *fibred-product-of-weakly-extensional-rts =*
A: weakly-extensional-rts A +
B: weakly-extensional-rts B +

```

    fibred-product-rts
begin

    sublocale weakly-extensional-rts resid
      ⟨proof⟩

    lemma is-weakly-extensional-rts:
    shows weakly-extensional-rts resid
      ⟨proof⟩

    lemma src-char:
    shows src t = (if arr t then (A.src (fst t), B.src (snd t)) else null)
      ⟨proof⟩

end

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

    sublocale fibred-product-of-weakly-extensional-rts A B ⟨proof⟩
    sublocale extensional-rts resid
      ⟨proof⟩

    lemma is-extensional-rts:
    shows extensional-rts resid
      ⟨proof⟩

end

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

    sublocale small-rts resid
      ⟨proof⟩

    lemma is-small-rts:
    shows small-rts resid
      ⟨proof⟩

end

```

3.8 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.

notation *product-rts.resid* (infixr $\langle \otimes \rangle$ 51)

definition *pointwise-tuple* :: ('x \Rightarrow 'a) \Rightarrow ('x \Rightarrow 'b) \Rightarrow 'x \Rightarrow 'a \times 'b ($\langle \langle \langle -, - \rangle \rangle \rangle$)
where *pointwise-tuple* H K \equiv ($\lambda t. (H\ t, K\ t)$)

context *product-rts*
begin

definition P_0 :: 'a \times 'b \Rightarrow 'b
where $P_0\ t \equiv$ if arr t then snd t else B.null

definition P_1 :: 'a \times 'b \Rightarrow 'a
where $P_1\ t \equiv$ if arr t then fst t else A.null

sublocale P_0 : simulation resid B P_0
 \langle proof \rangle

lemma P_0 -is-simulation:
shows simulation resid B P_0
 \langle proof \rangle

sublocale P_1 : simulation resid A P_1
 \langle proof \rangle

lemma P_1 -is-simulation:
shows simulation resid A P_1
 \langle proof \rangle

abbreviation *tuple* :: ('x \Rightarrow 'a) \Rightarrow ('x \Rightarrow 'b) \Rightarrow 'x \Rightarrow 'a \times 'b
where *tuple* \equiv *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 $\exists ! HK. \text{simulation } X \text{ resid } HK \wedge P_1 \circ HK = H \wedge P_0 \circ HK = K$
 \langle proof \rangle

lemma *proj-joint-monic*:
assumes simulation X resid F **and** simulation X resid G
and $P_0 \circ F = P_0 \circ G$ **and** $P_1 \circ F = P_1 \circ G$

shows $F = G$
<proof>

lemma *tuple-proj*:
assumes *simulation X resid F*
shows $\langle\langle P_1 \circ F, P_0 \circ F \rangle\rangle = F$
<proof>

lemma *proj-tuple*:
assumes *simulation X A F and simulation X B G*
shows $P_1 \circ \langle\langle F, G \rangle\rangle = F$ **and** $P_0 \circ \langle\langle F, G \rangle\rangle = G$
<proof>

lemma *preserves-weakly-extensional-rts*:
assumes *weakly-extensional-rts A and weakly-extensional-rts B*
shows *weakly-extensional-rts resid*
<proof>

lemma *preserves-extensional-rts*:
assumes *extensional-rts A and extensional-rts B*
shows *extensional-rts resid*
<proof>

lemma *preserves-small-rts*:
assumes *small-rts A and small-rts B*
shows *small-rts resid*
<proof>

lemma *preserves-rts-with-composites*:
assumes *rts-with-composites A and rts-with-composites B*
shows *rts-with-composites resid*
<proof>

lemma *preserves-extensional-rts-with-composites*:
assumes *extensional-rts-with-composites A and extensional-rts-with-composites B*
shows *extensional-rts-with-composites resid*
<proof>

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* *<proof>*

sublocale *extensional-rts resid*
⟨*proof*⟩

lemma *is-extensional-rts*:
shows *extensional-rts resid*
⟨*proof*⟩

lemma *proj-tuple2*:
assumes *transformation X A F G S and transformation X B H K T*
shows $P_1 \circ \langle\langle S, T \rangle\rangle = S$ **and** $P_0 \circ \langle\langle S, T \rangle\rangle = T$
⟨*proof*⟩

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 $P_1 \circ \langle\langle S, T \rangle\rangle = S$ **and** $P_0 \circ \langle\langle S, T \rangle\rangle = T$
and $\exists!ST.$ *transformation X resid F G ST* $\wedge P_1 \circ ST = S \wedge P_0 \circ ST = T$
⟨*proof*⟩

lemma *proj-joint-monic2*:
assumes *transformation X resid F G S and transformation X resid F G T*
and $P_0 \circ S = P_0 \circ T$ **and** $P_1 \circ S = P_1 \circ T$
shows $S = T$
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *join-simp*:
assumes *joinable t u*
shows $join\ t\ u = (A.join\ (fst\ t)\ (fst\ u), B.join\ (snd\ t)\ (snd\ u))$
⟨*proof*⟩

end

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

begin

sublocale *small-rts resid*
⟨*proof*⟩

lemma *is-small-rts*:
shows *small-rts resid*
 ⟨*proof*⟩

end

lemma *simulation-tuple* [*intro*]:
assumes *simulation X A H and simulation X B K*
and $Y = A \otimes B$
shows *simulation X Y ⟨⟨H, K⟩⟩*
 ⟨*proof*⟩

lemma *simulation-product* [*intro*]:
assumes *simulation A B H and simulation C D K*
and $X = A \otimes C$ **and** $Y = B \otimes D$
shows *simulation X Y (product-simulation.map A C H K)*
 ⟨*proof*⟩

lemma *comp-pointwise-tuple*:
shows $\langle\langle H, K \rangle\rangle \circ L = \langle\langle H \circ L, K \circ L \rangle\rangle$
 ⟨*proof*⟩

lemma *comp-product-simulation-tuple2*:
assumes *simulation A A' F and simulation B B' G*
and *transformation X A H₀ H₁ H and transformation X B K₀ K₁ K*
shows $\text{product-simulation.map } A \ B \ F \ G \circ \langle\langle H, K \rangle\rangle = \langle\langle F \circ H, G \circ K \rangle\rangle$
 ⟨*proof*⟩

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 $\text{product-simulation.map } A \ B \ F \ G \circ \langle\langle H, K \rangle\rangle = \langle\langle F \circ H, G \circ K \rangle\rangle$
 ⟨*proof*⟩

locale *product-transformation* =
A1: rts A1 +
A0: 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 <proof>
  sublocale F0: simulation-to-weakly-extensional-rts A0 B0 F0 <proof>
  sublocale G1: simulation-to-weakly-extensional-rts A1 B1 G1 <proof>
  sublocale G0: simulation-to-weakly-extensional-rts A0 B0 G0 <proof>

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

  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>

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

  sublocale transformation
    A1xA0.resid B1xB0.resid F1xF0.map G1xG0.map map
  <proof>

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

  lemma map-simp:
  shows map t = B1xB0.join (map0 (A1xA0.src t)) (F1xF0.map t)
  <proof>

  lemma map-simp-ide:
  assumes A1xA0.ide t
  shows map t = map0 (A1xA0.src t)
  <proof>

```

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 $product\text{-}transformation.map\ A1\ A0\ B1\ B0\ F1\ F0\ T1\ T0 \circ \langle\langle H1, H0 \rangle\rangle =$
 $\langle\langle T1 \circ H1, T0 \circ H0 \rangle\rangle$

<proof>

lemma *simulation-interchange*:

assumes *simulation* $A\ A'\ F$ **and** *simulation* $B\ B'\ G$

and *simulation* $A'\ A''\ F'$ **and** *simulation* $B'\ B''\ G'$

shows $product\text{-}simulation.map\ A\ B\ (F' \circ F)\ (G' \circ G) =$

$product\text{-}simulation.map\ A'\ B'\ F'\ G' \circ product\text{-}simulation.map\ A\ B\ F\ G$

<proof>

3.8.1 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* $A\ B$ *<proof>*

sublocale BxC : *product-rts* $B\ C$ *<proof>*

sublocale $AxB-xC$: *product-rts* $AxB.resid\ C$ *<proof>*

sublocale $Ax-BxC$: *product-rts* $A\ BxC.resid$ *<proof>*

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* $AxB\text{-}xC.resid\ Ax\text{-}BxC.resid\ map$

<proof>

lemma *is-simulation*:

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

<proof>

The following explicit formula is more convenient for calculations.

lemma *map-eq*:

shows $map = (\lambda x. \text{if } AxB\text{-}xC.arr\ x$
 $\text{then } (fst\ (fst\ x),\ (snd\ (fst\ x),\ snd\ x))$
 $\text{else } Ax\text{-}BxC.null)$

⟨*proof*⟩

definition $map' :: 'a \times 'b \times 'c \Rightarrow ('a \times 'b) \times 'c$

where $map' \equiv AxB\text{-}xC.tuple$
 $(AxB.tuple\ Ax\text{-}BxC.P_1\ (BxC.P_1 \circ Ax\text{-}BxC.P_0))$
 $(BxC.P_0 \circ Ax\text{-}BxC.P_0)$

sublocale *inv*: *simulation* $Ax\text{-}BxC.resid\ AxB\text{-}xC.resid\ map'$

⟨*proof*⟩

lemma *inv-is-simulation*:

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

⟨*proof*⟩

lemma *map'-eq*:

shows $map' =$
 $(\lambda x. \text{if } Ax\text{-}BxC.arr\ x$
 $\text{then } ((fst\ x,\ fst\ (snd\ x)),\ snd\ (snd\ x))$
 $\text{else } AxB\text{-}xC.null)$

⟨*proof*⟩

lemma *inverse-simulations-map'-map*:

shows *inverse-simulations* $AxB\text{-}xC.resid\ Ax\text{-}BxC.resid\ map'\ map$

⟨*proof*⟩

end

3.9 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: *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 <\A> 70)
and B :: 'b resid    (infix <\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 <proof>
    sublocale τ: transformation-to-extensional-rts A B F H τ <proof>

```

```

    sublocale sym: consistent-transformations A B F H G τ σ
    <proof>

```

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 \B (σ (A.src t) \B τ (A.src t))
                else B.null)

```

```

abbreviation resid :: 'a ⇒ 'b
where resid ≡ (λt. if A.arr t
                then (σ (A.src t) \B τ (A.src t)) ⊔B 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>

```

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*
 $\langle proof \rangle$

lemma *simulation-apex:*
shows *simulation* $A B$ *apex*
 $\langle proof \rangle$

lemma *resid-ide:*
assumes $A.ide\ a$
shows $resid\ a = \sigma\ a\ \backslash_B\ \tau\ a$
 $\langle proof \rangle$

interpretation *resid: transformation* $\langle \backslash_A \rangle\ \langle \backslash_B \rangle\ H$ *apex resid*
 $\langle proof \rangle$

lemma *transformation-resid:*
shows *transformation* $\langle \backslash_A \rangle\ \langle \backslash_B \rangle\ H$ *apex resid*
 $\langle proof \rangle$

lemma *whisker-left:*
assumes *simulation* $B C K$ **and** *extensional-rts* C
shows *consistent-transformations* $A C (K \circ F) (K \circ G) (K \circ H) (K \circ \sigma) (K$
 $\circ \tau)$
 $\langle proof \rangle$

lemma *whisker-right:*
assumes *simulation* $C A K$
shows *consistent-transformations* $C B (F \circ K) (G \circ K) (H \circ K) (\sigma \circ K) (\tau$
 $\circ K)$
 $\langle proof \rangle$

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

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

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

notation $B.con$ (**infix** $\langle \frown_B \rangle$ 50)

notation $B.join$ (**infixr** $\langle \sqcup_B \rangle$ 52)

notation $B.prfx$ (**infix** $\langle \lesssim_B \rangle$ 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
 ⟨proof⟩

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⟩

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 $\tau = \text{Apex } \tau$ σ
 $\langle \text{proof} \rangle$

lemma *Apex-is-simulation* [*intro*]:
assumes *Con* σ τ
shows *simulation* *A* *B* (*Apex* σ τ)
 $\langle \text{proof} \rangle$

abbreviation *Resid* :: ('a, 'b) *arr* \Rightarrow ('a, 'b) *arr* \Rightarrow 'a \Rightarrow 'b
where *Resid* σ $\tau \equiv (\lambda t. \text{if } A.\text{arr } t$
 $\text{then } (\text{Map } \sigma (A.\text{src } t) \setminus_B \text{Map } \tau (A.\text{src } t)) \sqcup_B \text{Cod } \tau$ t
 $\text{else } B.\text{null})$

definition *resid* :: ('a, 'b) *arr* *resid* (**infix** $\langle \setminus \rangle$ 70)
where $\sigma \setminus \tau =$
 $(\text{if } \text{Con } \sigma$ τ
 $\text{then } \text{MkArr } (\text{Cod } \tau)$
 $(\text{consistent-transformations.apex } A$ *B* (*Cod* τ) (*Map* σ) (*Map* τ))
 $(\text{consistent-transformations.resid } A$ *B* (*Cod* τ) (*Map* σ) (*Map* τ))
 $\text{else } \text{Null})$

lemma *Dom-resid'*:
assumes *Con* σ τ
shows *Dom* ($\sigma \setminus \tau$) = *Cod* τ
 $\langle \text{proof} \rangle$

lemma *Cod-resid'*:
assumes *Con* σ τ
shows *Cod* ($\sigma \setminus \tau$) = *Apex* σ τ
 $\langle \text{proof} \rangle$

lemma *Map-resid'*:
assumes *Con* σ τ
shows *Map* ($\sigma \setminus \tau$) = $(\lambda t. \text{if } A.\text{arr } t$
 $\text{then } \text{Map } \sigma (A.\text{src } t) \setminus_B \text{Map } \tau (A.\text{src } t) \sqcup_B \text{Cod } \tau$ t
 $\text{else } B.\text{null})$
 $\langle \text{proof} \rangle$

lemma *Map-resid-ide'*:
assumes *Con* σ τ **and** *A.ide* *a*
shows *Map* ($\sigma \setminus \tau$) *a* = *Map* σ *a* \setminus_B *Map* τ *a*
 $\langle \text{proof} \rangle$

lemma *transformation-Map-resid*:
assumes *Con* σ τ
shows *transformation* (\setminus_A) (\setminus_B) (*Cod* τ) (*Apex* σ τ) (*Map* ($\sigma \setminus \tau$))
 $\langle \text{proof} \rangle$

sublocale *ResiduatedTransitionSystem.partial-magma resid*
⟨proof⟩

lemma *null-char*:
shows $null = Null$
⟨proof⟩

sublocale *residuation resid*
⟨proof⟩

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

lemma *con-char*:
shows $\sigma \frown \tau \longleftrightarrow Con \sigma \tau$
⟨proof⟩

lemma *arr-char*:
shows $arr \tau \longleftrightarrow Arr \tau$
⟨proof⟩

lemma *Dom-resid [simp]*:
assumes $\sigma \frown \tau$
shows $Dom (\sigma \setminus \tau) = Cod \tau$
⟨proof⟩

lemma *Cod-resid [simp]*:
assumes $\sigma \frown \tau$
shows $Cod (\sigma \setminus \tau) = Apex \sigma \tau$
⟨proof⟩

lemma *Map-resid [simp]*:
assumes $\sigma \frown \tau$
shows $Map (\sigma \setminus \tau) = (\lambda t. \text{if } A.arr \ t$
 $\text{then } Map \ \sigma \ (A.src \ t) \setminus_B \ Map \ \tau \ (A.src \ t) \sqcup_B \ Cod \ \tau \ t$
 $\text{else } B.null)$
⟨proof⟩

lemma *Map-resid-ide [simp]*:
assumes $con \ \sigma \ \tau$ **and** $A.ide \ a$
shows $Map (\sigma \setminus \tau) \ a = Map \ \sigma \ a \setminus_B \ Map \ \tau \ a$
⟨proof⟩

lemma *resid-Map*:
assumes $con \ \varrho \ \sigma$ **and** $t \frown_A \ u$
shows $Map \ \varrho \ t \setminus_B \ Map \ \sigma \ u = Map \ (\varrho \setminus \sigma) \ (t \setminus_A \ u)$
⟨proof⟩

lemma *resid-def'*:

shows $\sigma \setminus \tau =$
 (if $\sigma \frown \tau$
 then $MkArr (Cod \tau) (Apex \sigma \tau)$
 ($\lambda t. \text{if } A.arr \ t$
 then $Map \ \sigma \ (A.src \ t) \setminus_B \ Map \ \tau \ (A.src \ t) \sqcup_B \ Cod \ \tau \ t$
 else $B.null$)
 else $null$)
 $\langle proof \rangle$

lemma *trg-simp*:
assumes $arr \ \tau$
shows $trg \ \tau = MkArr (Cod \ \tau) (Cod \ \tau) (Cod \ \tau)$
 $\langle proof \rangle$

lemma *trg-char*:
shows $trg = (\lambda \tau. \text{if } arr \ \tau \text{ then } MkIde (Cod \ \tau) \text{ else } null)$
 $\langle proof \rangle$

lemma *Map-trg [simp]*:
assumes $arr \ \tau$
shows $Map (trg \ \tau) = Cod \ \tau$
 $\langle proof \rangle$

lemma *resid-Map-self*:
assumes $arr \ \sigma$ **and** $t \frown_A \ u$
shows $Map \ \sigma \ t \setminus_B \ Map \ \sigma \ u = Cod \ \sigma \ (t \setminus_A \ u)$
 $\langle proof \rangle$

lemma *ide-char_{ERTS} [iff]*:
shows $ide \ \tau \longleftrightarrow \tau \neq null \wedge \text{simulation } A \ B \ (Map \ \tau) \wedge$
 $Dom \ \tau = Map \ \tau \wedge Cod \ \tau = Map \ \tau$
 $\langle proof \rangle$

sublocale *rts resid*
 $\langle proof \rangle$

lemma *is-rts*:
shows $rts \ resid$
 $\langle proof \rangle$

sublocale *extensional-rts resid*
 $\langle proof \rangle$

lemma *is-extensional-rts*:
shows $extensional-rts \ resid$
 $\langle proof \rangle$

lemma *conI_{ERTS} [intro]*:
assumes $coinitial \ \sigma \ \tau$

and $\bigwedge a. A. \text{ide } a \implies \text{Map } \sigma \ a \ \frown_B \ \text{Map } \tau \ a$
shows $\sigma \ \frown \ \tau$
 $\langle \text{proof} \rangle$

lemma *conE_{ERTS}* [*elim*]:
assumes $\sigma \ \frown \ \tau$
and $\llbracket \text{coinitial } \sigma \ \tau; \bigwedge t \ u. A. \text{con } t \ u \implies \text{Map } \sigma \ t \ \frown_B \ \text{Map } \tau \ u \rrbracket \implies T$
shows T
 $\langle \text{proof} \rangle$

lemma *arrI* [*intro*]:
assumes $f \neq \text{null}$ **and** *transformation* $A \ B \ (\text{Dom } f) \ (\text{Cod } f) \ (\text{Map } f)$
shows *arr* f
 $\langle \text{proof} \rangle$

lemma *arrE* [*elim*]:
assumes *arr* f
and $\llbracket f \neq \text{null}; \text{transformation } A \ B \ (\text{Dom } f) \ (\text{Cod } f) \ (\text{Map } f) \rrbracket \implies T$
shows T
 $\langle \text{proof} \rangle$

lemma *arr-MkArr* [*iff*]:
shows *arr* $(\text{MkArr } F \ G \ \tau) \longleftrightarrow \text{transformation } A \ B \ F \ G \ \tau$
 $\langle \text{proof} \rangle$

lemma *src-simp*:
assumes *arr* τ
shows *src* $\tau = \text{MkIde } (\text{Dom } \tau)$
 $\langle \text{proof} \rangle$

lemma *src-char*:
shows *src* $= (\lambda \tau. \text{if } \text{arr } \tau \text{ then } \text{MkIde } (\text{Dom } \tau) \text{ else } \text{null})$
 $\langle \text{proof} \rangle$

lemma *Map-src* [*simp*]:
assumes *arr* τ
shows *Map* $(\text{src } \tau) = \text{Dom } \tau$
 $\langle \text{proof} \rangle$

lemma *ide-MkIde* [*iff*]:
shows *ide* $(\text{MkIde } F) \longleftrightarrow \text{simulation } A \ B \ F$
 $\langle \text{proof} \rangle$

lemma *MkArr-Map*:
assumes $\tau \neq \text{Null}$
shows $\tau = \text{MkArr } (\text{Dom } \tau) \ (\text{Cod } \tau) \ (\text{Map } \tau)$
 $\langle \text{proof} \rangle$

lemma *MkIde-Dom*:
assumes *arr* τ
shows $MkIde (Dom \tau) = src \tau$
 $\langle proof \rangle$

lemma *MkIde-Cod*:
assumes *arr* τ
shows $MkIde (Cod \tau) = trg \tau$
 $\langle proof \rangle$

lemma *MkIde-Map*:
assumes *ide* a
shows $MkIde (Map a) = a$
 $\langle proof \rangle$

lemma *arr-eqI*:
assumes *arr* σ **and** *arr* τ **and** $Dom \sigma = Dom \tau$ **and** $Cod \sigma = Cod \tau$
and $\bigwedge a. A.ide a \implies Map \sigma a = Map \tau a$
shows $\sigma = \tau$
 $\langle proof \rangle$

lemma *seq-char*:
shows $seq \sigma \tau \longleftrightarrow Arr \sigma \wedge Arr \tau \wedge Cod \sigma = Dom \tau$
 $\langle proof \rangle$

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

lemma *prfx-char*:
shows $\sigma \lesssim \tau \longleftrightarrow arr \sigma \wedge arr \tau \wedge Dom \sigma = Dom \tau \wedge$
 $(\forall a. A.ide a \longrightarrow Map \sigma a \lesssim_B Map \tau a)$
 $\langle proof \rangle$

lemma *Map-preserves-prfx*:
assumes $\sigma \lesssim \tau$ **and** *A.arr* t
shows $Map \sigma t \lesssim_B Map \tau t$
 $\langle proof \rangle$

Joins in an Exponential RTS

notation *join* (**infixr** $\langle \sqcup \rangle$ 52)

lemma *join-char*:
shows *joinable* $\sigma \tau \longleftrightarrow$
 $arr \sigma \wedge arr \tau \wedge Dom \sigma = Dom \tau \wedge$
 $(\forall t. A.arr t \longrightarrow$
 $B.joinable (Map \sigma (A.src t) \sqcup_B Map \tau (A.src t)) (Dom \sigma t))$
and $\sigma \sqcup \tau =$
 $(if \textit{joinable} \sigma \tau$
 $then MkArr (Dom \tau) (Apex \sigma \tau)$

$(\lambda t. (\text{Map } \sigma (A.\text{src } t) \sqcup_B \text{Map } \tau (A.\text{src } t)) \sqcup_B \text{Dom } \tau t)$
else null
 <proof>

lemma *Dom-join*:
assumes *joinable* σ τ
shows $\text{Dom } (\sigma \sqcup \tau) = \text{Dom } \sigma$
 <proof>

lemma *Cod-join*:
assumes *joinable* σ τ
shows $\text{Cod } (\sigma \sqcup \tau) = \text{Apex } \sigma \tau$
 <proof>

lemma *Map-join*:
assumes *joinable* σ τ
shows $\text{Map } (\sigma \sqcup \tau) =$
 $(\lambda t. (\text{Map } \sigma (A.\text{src } t) \sqcup_B \text{Map } \tau (A.\text{src } t)) \sqcup_B \text{Dom } \tau t)$
 <proof>

end

3.9.1 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 $\text{small } \{F. F \text{ ' Collect } A.\text{arr} \subseteq \text{Collect } B.\text{arr} \wedge$
 $F \text{ ' (UNIV - Collect } A.\text{arr}) \subseteq \{B.\text{null}\}\}$
 <proof>

lemma *small-Collect-simulation*:
shows $\text{small } (\text{Collect } (\text{simulation } A B))$
 <proof>

lemma *small-Collect-transformation*:
assumes *simulation A B F* **and** *simulation A B G*
shows $\text{small } (\text{Collect } (\text{transformation } A B F G))$
 <proof>

sublocale *small-rts resid*
 <proof>

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

<proof>

end

3.9.2 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* *<proof>*

interpretation *B*: *extensional-rts-with-composites B* *<proof>*

notation *B.comp* (**infixr** $\langle \cdot_B \rangle$ 55)

abbreviation *COMP* :: $(\text{'a}, \text{'b}) \text{ arr} \Rightarrow (\text{'a}, \text{'b}) \text{ arr} \Rightarrow (\text{'a}, \text{'b}) \text{ arr}$

where *COMP t u* $\equiv \text{MkArr } (\text{Dom } t) (\text{Cod } u)$

$(\lambda x. \text{Map } t (A.\text{src } x) \cdot_B \text{Map } u (A.\text{src } x) \sqcup_B \text{Dom } t x)$

lemma *composite-of-iff*:

shows *composite-of t u v* $\longleftrightarrow \text{seq } t u \wedge v = \text{COMP } t u$

<proof>

corollary *is-rts-with-composites*:

shows *rts-with-composites resid*

<proof>

sublocale *rts-with-composites resid*

<proof>

sublocale *extensional-rts-with-composites resid* *<proof>*

lemma *naturality*:

assumes *arr* τ **and** *A.arr u*

shows $\text{Dom } \tau u \cdot_B \text{Map } \tau (A.\text{trg } u) = \text{Map } \tau (A.\text{src } u) \cdot_B \text{Cod } \tau u$

<proof>

lemma *Dom-comp* [*simp*]:

assumes *seq* $\sigma \tau$

shows $\text{Dom } (\text{comp } \sigma \tau) = \text{Dom } \sigma$

<proof>

lemma *Cod-comp* [*simp*]:

assumes *seq* $\sigma \tau$

shows $\text{Cod } (\text{comp } \sigma \tau) = \text{Cod } \tau$

<proof>

lemma *Map-comp* [*simp*]:

assumes *seq* $\sigma \tau$
shows $\text{Map } (\text{comp } \sigma \tau) =$
 $(\lambda x. \text{Map } \sigma (A.\text{src } x) \cdot_B \text{Map } \tau (A.\text{src } x) \sqcup_B \text{Dom } \sigma x)$
 $\langle \text{proof} \rangle$

lemma *Map-comp-ide*:
assumes *seq* $\sigma \tau$ **and** *A.ide* x
shows $\text{Map } (\text{comp } \sigma \tau) x = \text{Map } \sigma x \cdot_B \text{Map } \tau x$
 $\langle \text{proof} \rangle$

end

3.9.3 Exponential by One

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

locale *exponential-by-One* =
One: *one-arr-rts* +
A: *extensional-rts* A
for $A :: 'a \text{ resid}$ (**infix** $\langle \backslash_A \rangle$ 70)
begin

sublocale *exponential-rts* *One.resid* A $\langle \text{proof} \rangle$
notation *resid* (**infix** $\langle \backslash \rangle$ 70)
notation *con* (**infix** $\langle \frown \rangle$ 50)

abbreviation $Up :: 'a \Rightarrow ('b, 'a) \text{ arr}$
where $Up \ t \equiv$
 if $A.\text{arr } t$
 then MkArr
 $(\text{constant-simulation.map } \text{One.resid } A (A.\text{src } t))$
 $(\text{constant-simulation.map } \text{One.resid } A (A.\text{trg } t))$
 $(\text{constant-transformation.map } \text{One.resid } A \ t)$
 else null

abbreviation $Dn :: ('b, 'a) \text{ arr} \Rightarrow 'a$
where $Dn \ t \equiv \text{if } \text{arr } t \text{ then } \text{Map } t \ \text{One.the-arr} \ \text{else } A.\text{null}$

sublocale Up : *simulation* A *resid* Up
 $\langle \text{proof} \rangle$

sublocale Dn : *simulation* *resid* A Dn
 $\langle \text{proof} \rangle$

lemma *inverse-simulations-Dn-Up*:
shows *inverse-simulations* A *resid* Dn Up
 $\langle \text{proof} \rangle$

end

3.9.4 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 <proof>
  sublocale ABxA: product-rts AB.resid A <proof>

  notation AB.resid      (infix <\[A,B]> 55)
  notation ABxA.resid   (infix <\[A,B]xA> 55)
  notation AB.con       (infix <\∩[A,B]> 50)
  notation ABxA.con     (infix <\∩[A,B]xA> 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)
    <proof>

  lemma is-simulation:
  shows simulation ABxA.resid B map
    <proof>

  sublocale simulation ABxA.resid B map
    <proof>
  sublocale binary-simulation AB.resid A B map <proof>

  lemma src-map:
  assumes AB.arr Fg and A.arr f
  shows B.src (map (Fg, f)) = AB.Dom Fg (A.src f)
    <proof>

  lemma trg-map:
  assumes AB.arr Fg and A.arr f
  shows B.trg (map (Fg, f)) = AB.Cod Fg (A.trg f)
    <proof>

end

```

3.9.5 Currying

```

locale Currying =
  A: weakly-extensional-rts A +
  B: weakly-extensional-rts B +

```


lemma *Dom-Curry*:
assumes $A.arr\ f$
shows $BC.Dom\ (Curry\ F\ G\ \tau\ f) = (\lambda g. F\ (A.src\ f,\ g))$
 $\langle proof \rangle$

lemma *Cod-Curry*:
assumes $A.arr\ f$
shows $BC.Cod\ (Curry\ F\ G\ \tau\ f) = (\lambda g. G\ (A.trg\ f,\ g))$
 $\langle proof \rangle$

lemma *Map-Curry*:
assumes $A.arr\ f$
shows $BC.Map\ (Curry\ F\ G\ \tau\ f) = (\lambda g. \tau\ (f,\ g))$
 $\langle proof \rangle$

lemma *Map-simulation-expansion*:
assumes *simulation* $A\ BC.resid\ G$ **and** $AxB.arr\ f$
shows $BC.Map\ (G\ (fst\ f))\ (snd\ f) =$
 $BC.Map\ (G\ (fst\ f))\ (B.src\ (snd\ f)) \sqcup_C$
 $BC.Map\ (G\ (A.src\ (fst\ f)))\ (snd\ f)$
 $\langle proof \rangle$

lemma *Map-simulation-monotone*:
assumes *simulation* $A\ BC.resid\ G$ **and** $f \lesssim_{AxB} g$
shows $BC.Map\ (G\ (fst\ f))\ (snd\ f) \lesssim_C BC.Map\ (G\ (fst\ g))\ (snd\ g)$
 $\langle proof \rangle$

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

lemma *Uncurry-preserves-simulations* [*intro*]:
assumes *simulation* $A\ BC.resid\ F$
shows *simulation* $AxB.resid\ C\ (Uncurry\ F)$
 $\langle proof \rangle$

lemma *Curry-preserves-transformations*:
assumes *transformation* $AxB.resid\ C\ F\ G\ \tau$
shows *transformation* $A\ BC.resid\ (Curry3\ F)\ (Curry3\ G)\ (Curry\ F\ G\ \tau)$
 $\langle proof \rangle$

lemma *Uncurry-preserves-transformations*:
assumes *transformation* $A\ BC.resid\ F\ G\ \tau$
shows *transformation* $AxB.resid\ C\ (Uncurry\ F)\ (Uncurry\ G)\ (Uncurry\ \tau)$
 $\langle proof \rangle$

lemma *Uncurry-Curry*:
assumes *transformation* $AxB.resid\ C\ F\ G\ \tau$

shows $Uncurry (Curry F G \tau) = \tau$
 $\langle proof \rangle$

lemma *Curry-Uncurry*:
assumes transformation $A BC.resid F G \tau$
shows $Curry (Uncurry F) (Uncurry G) (Uncurry \tau) = \tau$
 $\langle proof \rangle$

lemma *src-Curry*:
assumes transformation $AxB.resid C F G \tau$ **and** $A.arr f$
shows $BC.src (Curry F G \tau f) = Curry3 F (A.src f)$
 $\langle proof \rangle$

lemma *trg-Curry*:
assumes transformation $AxB.resid C F G \tau$ **and** $A.arr f$
shows $BC.trg (Curry F G \tau f) = Curry3 G (A.trg f)$
 $\langle proof \rangle$

lemma *src-Uncurry*:
assumes transformation $A BC.resid F G \tau$ **and** $AxB.arr f$
shows $C.src (Uncurry \tau f) = Uncurry F (AxB.src f)$
 $\langle proof \rangle$

lemma *trg-Uncurry*:
assumes transformation $A BC.resid F G \tau$ **and** $AxB.arr f$
shows $C.trg (Uncurry \tau f) = Uncurry G (AxB.trg f)$
 $\langle proof \rangle$

end

3.9.6 Currying and Uncurrying as Inverse Simulations

context *Currying*
begin

sublocale $AxB-C$: *exponential-rts* $AxB.resid C$ $\langle proof \rangle$
sublocale $A-BC$: *exponential-rts* $A BC.resid$ $\langle proof \rangle$

notation $AxB-C.resid$ (**infix** $\langle _ \rangle_{[AxB,C]}$ 55)
notation $AxB-C.con$ (**infix** $\langle _ \rangle_{\neg[AxB,C]}$ 55)
notation $A-BC.resid$ (**infix** $\langle _ \rangle_{[A,[B,C]]}$ 55)
notation $A-BC.con$ (**infix** $\langle _ \rangle_{\neg[A,[B,C]]}$ 50)

definition $CURRY :: ('a \times 'b, 'c) AxB-C.arr \Rightarrow ('a, ('b, 'c) BC.arr) A-BC.arr$
where $CURRY \equiv$
 $(\lambda X. \text{ if } AxB-C.arr X$
 $\text{ then } A-BC.MkArr$
 $(Curry (AxB-C.Dom X) (AxB-C.Dom X) (AxB-C.Dom X))$
 $(Curry (AxB-C.Cod X) (AxB-C.Cod X) (AxB-C.Cod X))$

(*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*)

⟨*proof*⟩

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*)

⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *CURRY-preserves-con*:

assumes *con*: *AxB-C.con* *t* *u*

shows *A-BC.con* (*CURRY* *t*) (*CURRY* *u*)

⟨*proof*⟩

lemma *CURRY-is-extensional*:

assumes \neg *AxB-C.arr* *t*

shows *CURRY* *t* = *A-BC.null*

⟨*proof*⟩

lemma *CURRY-preserves-arr*:

assumes *AxB-C.arr* *t*

shows *A-BC.arr* (*CURRY* *t*)

⟨*proof*⟩

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*⟩

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*⟩

lemma *Map-Map-CURRY-resid*:

assumes *con*: $AxB-C.con\ t\ u$ **and** $f1: A.arr\ f1$
shows $BC.Map\ (A-BC.Map\ (CURRY\ (t\ \backslash_{[AxB,C]}\ u))\ f1) =$
 $BC.Map\ (A-BC.Map\ (A-BC.resid\ (CURRY\ t)\ (CURRY\ u))\ f1)$
 $\langle proof \rangle$

lemma *Cod-Curry-Apex*:
assumes *con*: $AxB-C.con\ t\ u$ **and** $f1: A.arr\ f1$
shows $BC.Cod$
 $(Curry\ (AxB-C.Apex\ t\ u)\ (AxB-C.Apex\ t\ u)\ (AxB-C.Apex\ t\ u)\ f1) =$
 $BC.Cod\ (A-BC.Map\ (A-BC.resid\ (CURRY\ t)\ (CURRY\ u))\ f1)$
 $\langle proof \rangle$

lemma *Curry-preserves-Apex*:
assumes $AxB-C.con\ t\ u$
shows $Curry^3\ (AxB-C.Apex\ t\ u) = A-BC.Apex\ (CURRY\ t)\ (CURRY\ u)$
 $(is\ ?LHS = ?RHS)$
 $\langle proof \rangle$

sublocale *CURRY*: *simulation* $AxB-C.resid\ A-BC.resid\ CURRY$
 $\langle proof \rangle$

lemma *CURRY-is-simulation*:
shows *simulation* $AxB-C.resid\ A-BC.resid\ CURRY$
 $\langle proof \rangle$

definition *UNCURRY*
 $:: ('a, ('b, 'c)\ BC.arr)\ A-BC.arr \Rightarrow ('a \times 'b, 'c)\ AxB-C.arr$
where $UNCURRY\ f \equiv if\ A-BC.arr\ f$
 $then\ AxB-C.MkArr\ (Uncurry\ (A-BC.Dom\ f))$
 $(Uncurry\ (A-BC.Cod\ f))$
 $(Uncurry\ (A-BC.Map\ f))$
 $else\ AxB-C.null$

lemma *Dom-UNCURRY* [*simp*]:
assumes $A-BC.arr\ f$
shows $AxB-C.Dom\ (UNCURRY\ f) = Uncurry\ (A-BC.Dom\ f)$
 $\langle proof \rangle$

lemma *Cod-UNCURRY* [*simp*]:
assumes $A-BC.arr\ f$
shows $AxB-C.Cod\ (UNCURRY\ f) = Uncurry\ (A-BC.Cod\ f)$
 $\langle proof \rangle$

lemma *Map-UNCURRY* [*simp*]:
assumes $A-BC.arr\ f$
shows $AxB-C.Map\ (UNCURRY\ f) = Uncurry\ (A-BC.Map\ f)$
 $\langle proof \rangle$

lemma *UNCURRY-CURRY* [*simp*]:

assumes $AxB-C.arr\ t$
shows $UNCURRY\ (CURRY\ t) = t$
 $\langle proof \rangle$

lemma $CURRY-UNCURRY$ [*simp*]:
assumes $A-BC.arr\ t$
shows $CURRY\ (UNCURRY\ t) = t$
 $\langle proof \rangle$

lemma $UNCURRY-is-simulation$:
shows $simulation\ A-BC.resid\ AxB-C.resid\ UNCURRY$
 $\langle proof \rangle$

sublocale $UNCURRY: simulation\ A-BC.resid\ AxB-C.resid\ UNCURRY$
 $\langle proof \rangle$

interpretation $inverse-simulations$
 $A-BC.resid\ AxB-C.resid\ CURRY\ UNCURRY$
 $\langle proof \rangle$

sublocale $CURRY: invertible-simulation\ AxB-C.resid\ A-BC.resid\ CURRY$
 $\langle proof \rangle$

lemma $invertible-simulation-CURRY$:
shows $invertible-simulation\ AxB-C.resid\ A-BC.resid\ CURRY$
 $\langle proof \rangle$

sublocale $UNCURRY: invertible-simulation$
 $A-BC.resid\ AxB-C.resid\ UNCURRY$
 $\langle proof \rangle$

lemma $invertible-simulation-UNCURRY$:
shows $invertible-simulation\ A-BC.resid\ AxB-C.resid\ UNCURRY$
 $\langle proof \rangle$

sublocale $inverse-simulations\ A-BC.resid\ AxB-C.resid\ CURRY\ UNCURRY$
 $\langle proof \rangle$

lemma $inverse-simulations-CURRY-UNCURRY$:
shows $inverse-simulations\ A-BC.resid\ AxB-C.resid\ CURRY\ UNCURRY$
 $\langle proof \rangle$

end

3.9.7 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*
 $:: 'c \text{ resid} \Rightarrow ('c \times 'a \Rightarrow 'b) \Rightarrow 'c \Rightarrow ('a, 'b) \text{ AB.arr}$
where $\text{coext } X \ G \equiv \text{Currying.Curry3 } X \ A \ B \ G$

lemma *Uncurry-simulation-expansion:*
assumes *weakly-extensional-rtts* X
and *simulation* $X \ \text{AB.resid } F$
shows $\text{Currying.Uncurry } X \ A \ B \ F =$
 $\text{map} \circ (\text{product-simulation.map } X \ A \ F \ (I \ A))$
 $\langle \text{proof} \rangle$

lemma *universality:*
assumes *weakly-extensional-rtts* X
and *simulation* $(X \otimes A) \ B \ G$
shows *simulation* $X \ \text{AB.resid} \ (\text{Currying.Curry3 } X \ A \ B \ G)$
and $\text{Currying.Uncurry } X \ A \ B \ (\text{coext } X \ G) = G$
and $\exists ! F. \text{simulation } X \ \text{AB.resid } F \wedge \text{Currying.Uncurry } X \ A \ B \ F = G$
 $\langle \text{proof} \rangle$

lemma *comp-coext-simulation:*
assumes *weakly-extensional-rtts* X **and** *weakly-extensional-rtts* X'
and *simulation* $X \ X' \ G$
and *simulation* $(X' \otimes A) \ B \ H$
shows $\text{coext } X' \ H \circ G = \text{coext } X \ (H \circ \text{product-simulation.map } X \ A \ G \ (I \ A))$
 $\langle \text{proof} \rangle$

end

locale *evaluation-map-between-extensional-rtts* =
 $\text{evaluation-map} +$
 $A: \text{extensional-rtts } A$
begin

lemma *Uncurry-transformation-expansion:*
assumes *weakly-extensional-rtts* X
and *transformation* $X \ \text{AB.resid } F \ G \ T$
shows $\text{Currying.Uncurry } X \ A \ B \ T =$
 $\text{map} \circ \text{product-transformation.map } X \ A \ \text{AB.resid } A \ F \ (I \ A) \ T \ (I \ A)$
 $\langle \text{proof} \rangle$

lemma *universality2:*
assumes *weakly-extensional-rtts* X
and *transformation* $(X \otimes A) \ B \ F \ G \ T$
shows *transformation* $X \ \text{AB.resid}$
 $(\text{Currying.Curry3 } X \ A \ B \ F) \ (\text{Currying.Curry3 } X \ A \ B \ G)$

```

      (Currying.Curry X A B F G T)
and Currying.Uncurry X A B (Currying.Curry X A B F G T) = T
and  $\exists! T'. \text{transformation } X \text{ AB.resid}$ 
      (Currying.Curry3 X A B F) (Currying.Curry3 X A B G)
      T'  $\wedge$ 
      Currying.Uncurry X A B T' = T
    <proof>

end

```

3.9.8 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 <proof>
  sublocale AB: exponential-rts A B <proof>
  sublocale BC: exponential-rts B C <proof>

  sublocale BCxAB: product-rts BC.resid AB.resid <proof>
  sublocale BCxAB: product-of-extensional-rts BC.resid AB.resid <proof>

  interpretation AB: identity-simulation AB.resid <proof>
  interpretation BC: identity-simulation BC.resid <proof>

  interpretation ABxA: product-rts AB.resid A <proof>
  interpretation BCxB: product-rts BC.resid B <proof>
  interpretation BCxAB: identity-simulation BCxAB.resid <proof>
  interpretation BCxAB-x-A: product-rts BCxAB.resid A <proof>
  interpretation BC-x-ABxA: product-rts BC.resid ABxA.resid <proof>

  interpretation E-AB: RTSConstructions.evaluation-map A B <proof>
  interpretation E-BC: RTSConstructions.evaluation-map B C <proof>
  interpretation BCxE-AB: product-simulation
    BC.resid ABxA.resid BC.resid B
    BC.map E-AB.map <proof>

  interpretation ASSOC-BC-AB-A: ASSOC BC.resid AB.resid A <proof>
  sublocale Currying: Currying BCxAB.resid A C <proof>

```

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

definition $map :: ('b, 'c) BC.arr \times ('a, 'b) AB.arr \Rightarrow ('a, 'c) AC.arr$
where $map = (\lambda F. Currying.Curry3 F)$
 $(E-BC.map \circ BCxE-AB.map \circ ASSOC-BC-AB-A.map)$

sublocale *simulation* $BCxAB.resid AC.resid map$
 $\langle proof \rangle$

lemma *is-simulation:*
shows *simulation* $BCxAB.resid AC.resid map$
 $\langle proof \rangle$

sublocale *binary-simulation* $BC.resid AB.resid AC.resid map \langle proof \rangle$

lemma *is-binary-simulation:*
shows *binary-simulation* $BC.resid AB.resid AC.resid map$
 $\langle proof \rangle$

sublocale *E-BC-o-BCxE-AB: composite-simulation*
 $BC-x-ABxA.resid BCxB.resid C$
 $BCxE-AB.map E-BC.map$
 $\langle proof \rangle$

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 $map = (\lambda gf. \text{if } BCxAB.arr \text{ gf}$
 $\text{then } AC.MkArr$
 $(BC.Dom (fst gf) \circ AB.Dom (snd gf))$
 $(BC.Cod (fst gf) \circ AB.Cod (snd gf))$
 $(BC.Map (fst gf) \circ AB.Map (snd gf))$
 $\text{else } AC.Null)$
 $\langle proof \rangle$

end

3.9.9 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*

$:: 'c \text{ resid} \Rightarrow 'a \text{ resid} \Rightarrow 'b \text{ resid} \Rightarrow ('a \Rightarrow 'b)$
 $\Rightarrow ('c, 'a) \text{ exponential-rts.arr} \Rightarrow ('c, 'b) \text{ exponential-rts.arr} \quad (\langle \text{Exp}^{\rightarrow} \rangle)$
where $\text{Exp}^{\rightarrow} X B C \equiv$
 $\lambda G t2. \text{COMP.map } X B C (\text{exponential-rts.MkIde } G, t2)$

abbreviation *cnt-Exp*

$:: 'a \text{ resid} \Rightarrow 'b \text{ resid} \Rightarrow 'c \text{ resid} \Rightarrow ('a \Rightarrow 'b)$
 $\Rightarrow ('b, 'c) \text{ exponential-rts.arr} \Rightarrow ('a, 'c) \text{ exponential-rts.arr} \quad (\langle \text{Exp}^{\leftarrow} \rangle)$
where $\text{Exp}^{\leftarrow} A B X \equiv$
 $\lambda F t1. \text{COMP.map } A B X (t1, \text{exponential-rts.MkIde } F)$

lemma *cov-Exp-eq*:

assumes *extensional-rts X and extensional-rts B and extensional-rts C*

shows $\text{Exp}^{\rightarrow} X B C G =$

$(\lambda t2. \text{if simulation } B C G \wedge \text{residuation.arr} (\text{exponential-rts.resid } X B) t2$
 $\text{then exponential-rts.MkArr}$
 $\quad (G \circ \text{exponential-rts.Dom } t2)$
 $\quad (G \circ \text{exponential-rts.Cod } t2)$
 $\quad (G \circ \text{exponential-rts.Map } t2)$
 $\text{else exponential-rts.Null})$

<proof>

lemma *cnt-Exp-eq*:

assumes *extensional-rts X and extensional-rts A and extensional-rts B*

shows $\text{Exp}^{\leftarrow} A B X F =$

$(\lambda t1. \text{if residuation.arr} (\text{exponential-rts.resid } B X) t1 \wedge \text{simulation } A B F$
 $\text{then exponential-rts.MkArr}$
 $\quad (\text{exponential-rts.Dom } t1 \circ F)$
 $\quad (\text{exponential-rts.Cod } t1 \circ F)$
 $\quad (\text{exponential-rts.Map } t1 \circ F)$
 $\text{else exponential-rts.Null})$

<proof>

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)*

$(\text{Exp}^{\rightarrow} X B C G)$

<proof>

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)*

$(\text{Exp}^{\leftarrow} A B X F)$

<proof>

lemma *cov-Exp-ide*:

assumes *extensional-rts X and extensional-rts B*

shows $Exp^{\rightarrow} X B B (I B) = I$ (*exponential-rts.resid X B*)
(*proof*)

lemma *cnt-Exp-ide*:

assumes *extensional-rts X* **and** *extensional-rts B*
shows $Exp^{\leftarrow} B B X (I B) = I$ (*exponential-rts.resid B X*)
(*proof*)

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 $Exp^{\rightarrow} X B D (G \circ F) = Exp^{\rightarrow} X C D G \circ Exp^{\rightarrow} X B C F$
(*proof*)

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 $Exp^{\leftarrow} B D X (G \circ F) = Exp^{\leftarrow} B C X F \circ Exp^{\leftarrow} C D X G$
(*proof*)

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*)
 ($Exp^{\rightarrow} X C B G$) ($Exp^{\rightarrow} X B C F$)
(*proof*)

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*)
 ($Exp^{\rightarrow} X B C F$)
(*proof*)

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*)
 ($Exp^{\leftarrow} B C X F$) ($Exp^{\leftarrow} C B X G$)
(*proof*)

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*)

$(Exp \leftarrow B C X F)$
 $\langle proof \rangle$

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 RTSConstrutions 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: fibred-product-rts resid resid resid H.dom H.cod +
  H: simulation VV.resid resid
     $\langle \lambda t. \text{if } VV.arr\ t \text{ then } hcomp\ (fst\ t)\ (snd\ t) \text{ else } V.null \rangle$ 
for resid :: 'a resid (infix  $\langle \rangle$  70)
and hcomp :: 'a comp (infixr  $\langle \star \rangle$  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 ( $\langle \langle - : - \rightarrow - \rangle \rangle$ )

  abbreviation null
  where null  $\equiv V.null$ 

  abbreviation arr
  where arr  $\equiv V.arr$ 
```

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*: *fibred-product-of-extensional-rts resid resid resid dom cod*
 $\langle proof \rangle$

interpretation *H*: *simulation-between-extensional-rts VV.resid resid*
 $\langle \lambda t. \text{if } VV.arr\ t \text{ then } fst\ t \star snd\ t \text{ else } null \rangle$
 $\langle proof \rangle$

lemma *obj-is-isolated*:
assumes *obj a* **and** *obj a'* **and** $a \star a' \neq null$
shows $a = a'$
 $\langle proof \rangle$

lemma *obj-implies-sta*:
assumes *obj a*
shows *sta a*
 $\langle proof \rangle$

lemma *trg-dom* [*simp*]:
shows $trg\ (dom\ t) = dom\ t$
 $\langle proof \rangle$

lemma *src-cod* [*simp*]:
shows $src\ (cod\ t) = cod\ t$
 $\langle proof \rangle$

lemma *trg-cod* [*simp*]:
shows $trg\ (cod\ t) = cod\ t$
 $\langle proof \rangle$

lemma *dom-src* [*simp*]:
shows $\text{dom } (\text{src } t) = \text{dom } t$
 $\langle \text{proof} \rangle$

lemma *dom-trg* [*simp*]:
shows $\text{dom } (\text{trg } t) = \text{dom } t$
 $\langle \text{proof} \rangle$

lemma *cod-src* [*simp*]:
shows $\text{cod } (\text{src } t) = \text{cod } t$
 $\langle \text{proof} \rangle$

lemma *cod-trg* [*simp*]:
shows $\text{cod } (\text{trg } t) = \text{cod } t$
 $\langle \text{proof} \rangle$

lemma *arr-hcomp* [*intro*]:
assumes $H.\text{seq } t \ u$
shows $\text{arr } (t \star u)$
 $\langle \text{proof} \rangle$

lemma *sta-hcomp* [*intro*]:
assumes $H.\text{seq } t \ u$ **and** $\text{sta } t$ **and** $\text{sta } u$
shows $\text{sta } (t \star u)$
 $\langle \text{proof} \rangle$

lemma *src-hcomp* [*simp*]:
assumes $H.\text{seq } t \ u$
shows $\text{src } (t \star u) = \text{src } t \star \text{src } u$
 $\langle \text{proof} \rangle$

lemma *trg-hcomp* [*simp*]:
assumes $H.\text{seq } t \ u$
shows $\text{trg } (t \star u) = \text{trg } t \star \text{trg } u$
 $\langle \text{proof} \rangle$

lemma *con-implies-hpar*:
assumes $t \frown u$
shows $H.\text{par } t \ u$
 $\langle \text{proof} \rangle$

lemma *hpar-arr-resid*:
assumes $t \frown u$
shows $H.\text{par } t \ (t \setminus u)$
 $\langle \text{proof} \rangle$

lemma *dom-resid* [*simp*]:
assumes $t \frown u$

shows $dom (t \setminus u) = dom t$
 ⟨proof⟩

lemma *cod-resid* [simp]:
assumes $t \frown u$
shows $cod (t \setminus u) = cod t$
 ⟨proof⟩

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⟩

lemma *dom-vcomp* [simp]:
assumes $V.composable\ t\ u$
shows $dom (t \cdot u) = dom t$
 ⟨proof⟩

lemma *cod-vcomp* [simp]:
assumes $V.composable\ t\ u$
shows $cod (t \cdot u) = cod t$
 ⟨proof⟩

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_{RTSC}*:
assumes $V.composable\ t\ r$ **and** $V.composable\ u\ s$ **and** $H.seq\ t\ u$
shows $V.composable\ (t \star u)\ (r \star s)$
and $(t \star u) \cdot (r \star s) = (t \cdot r) \star (u \cdot s)$
 ⟨proof⟩

lemma *hcomp-monotone*:
assumes $r \lesssim t$ **and** $s \lesssim u$ **and** $H.seq\ r\ s$
shows $r \star s \lesssim t \star u$
 ⟨proof⟩

lemma *dom-join* [simp]:
assumes $V.joinable\ t\ u$
shows $H.dom (t \sqcup u) = H.dom t$
 ⟨proof⟩

lemma *cod-join* [simp]:
assumes $V.joinable\ t\ u$
shows $H.cod (t \sqcup u) = H.cod t$
 ⟨proof⟩

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>

The source and target maps given by the vertical structure are functorial with respect to the horizontal structure.

sublocale *src: functor hcomp hcomp src*

<proof>

sublocale *trg: functor hcomp hcomp trg*

<proof>

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>

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 $HOM\ a\ b \equiv sub\text{-}rts.resid\ resid\ (\lambda t. \langle t : a \rightarrow b \rangle)$

lemma *sub-rts-HOM*:

shows $sub\text{-}rts\ resid\ (\lambda t. \langle t : a \rightarrow b \rangle)$

<proof>

lemma *extensional-rts-HOM*:

assumes $obj\ a$ **and** $obj\ b$

shows $HOM\ a\ b \in Collect\ extensional\text{-}rts$

<proof>

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 $cov\text{-}HOM\ (\langle HOM^{\rightarrow} \rangle)$

where $HOM^{\rightarrow}\ a\ t \equiv$

$(\lambda x. if\ residuation.arr\ (HOM\ a\ (dom\ t))\ x\ then\ t \star x\ else\ null)$

lemma *simulation-cov-HOM-sta*:

assumes *obj a* **and** *sta f*

shows *simulation* ($HOM\ a\ (dom\ f)$) ($HOM\ a\ (cod\ f)$) ($HOM^{\rightarrow}\ a\ f$)
 $\langle proof \rangle$

lemma *transformation-cov-HOM-arr*:

assumes *obj a* **and** *arr t*

shows *transformation* ($HOM\ a\ (dom\ t)$) ($HOM\ a\ (cod\ t)$)
 $(HOM^{\rightarrow}\ a\ (src\ t))\ (HOM^{\rightarrow}\ a\ (trg\ t))\ (HOM^{\rightarrow}\ a\ t)$
 $\langle proof \rangle$

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 $HOM^{\rightarrow}\ a\ (t\ \star\ u) = HOM^{\rightarrow}\ a\ t\ \circ\ HOM^{\rightarrow}\ a\ u$
 $\langle proof \rangle$

The mapping $HOM^{\rightarrow}\ a$ preserves consistency and residuation.

lemma *cov-HOM-resid*:

assumes *obj a* **and** $V.con\ t\ u$

shows *consistent-transformations*
 $(HOM\ a\ (dom\ t))\ (HOM\ a\ (cod\ t))$
 $(HOM^{\rightarrow}\ a\ (src\ t))\ (HOM^{\rightarrow}\ a\ (trg\ t))\ (HOM^{\rightarrow}\ a\ (trg\ u))$
 $(HOM^{\rightarrow}\ a\ t)\ (HOM^{\rightarrow}\ a\ u)$

and $cov-HOM\ a\ (t\ \setminus\ u) =$

consistent-transformations.resid
 $(HOM\ a\ (dom\ t))\ (HOM\ a\ (cod\ t))\ (HOM^{\rightarrow}\ a\ (trg\ u))$
 $(HOM^{\rightarrow}\ a\ t)\ (HOM^{\rightarrow}\ a\ u)$

$\langle proof \rangle$

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* ($\langle HOM^{\leftarrow} \rangle$)

where $HOM^{\leftarrow}\ c\ t \equiv$

$(\lambda x. \text{if } residuation.arr\ (HOM\ (cod\ t)\ c)\ x \text{ then } x\ \star\ t \text{ else null})$

lemma *simulation-cnt-HOM-sta*:

assumes *sta f* **and** $\langle f : a \rightarrow b \rangle$ **and** *obj c*

shows *simulation* ($HOM\ b\ c$) ($HOM\ a\ c$) ($HOM^{\leftarrow}\ c\ f$)
 $\langle proof \rangle$

lemma *HOM-preserves-isomorphic-left*:

assumes $H.isomorphic\ a\ b$ **and** *obj c*

shows *isomorphic-rts* ($HOM\ a\ c$) ($HOM\ b\ c$)
 $\langle proof \rangle$

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:  $\llbracket \text{obj } a; \text{obj } b \rrbracket \implies \text{small } (H.\text{hom } a \ b)$   
begin
```

```
lemma HOM-is-small-extensional-rts:  
  assumes obj a and obj b  
  shows  $HOM \ a \ b \in \text{Collect extensional-rts} \cap \text{Collect small-rts}$   
  <proof>
```

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 residA compA +  
  B: rts-category residB compB +  
  functor compA compB F +  
  simulation residA residB F  
for residA :: 'a resid (infix <\A> 70)  
and compA :: 'a comp (infixr <★A> 53)  
and residB :: 'b resid (infix <\B> 70)  
and compB :: 'b comp (infixr <★B> 53)  
and F :: 'a  $\Rightarrow$  'b  
begin
```

```
notation A.V.con (infix <\A> 50)  
notation B.V.con (infix <\B> 50)
```

```
lemma is-invertible-simulation-if:  
  assumes invertible-functor compA compB F  
  and  $\bigwedge t \ u. F \ t \ \frown_B \ F \ u \implies t \ \frown_A \ u$   
  shows invertible-simulation residA residB F  
  <proof>
```

```
lemma is-invertible-if:  
  assumes invertible-simulation residA residB F  
  shows invertible-functor compA compB F  
  <proof>
```

```

end

locale rts-category-isomorphism =
  rts-functor +
  invertible-simulation residA residB F
begin

  sublocale invertible-functor compA compB F
    ⟨proof⟩

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* × *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 ⇒ 'O ⇒ 'A resid
and Id :: 'O ⇒ 'A
and Comp :: 'O ⇒ 'O ⇒ 'O ⇒ 'A ⇒ 'A ⇒ 'A
assumes rts-Hom: [ A ∈ Obj; B ∈ Obj ] ⇒ extensional-rts (Hom A B)
and binary-simulation-Comp:
  [ A ∈ Obj; B ∈ Obj; C ∈ Obj ]
  ⇒ binary-simulation
    (Hom B C) (Hom A B) (Hom A C) (λ(t, u). Comp A B C t u)
and ide-Id: A ∈ Obj ⇒ residuation.ide (Hom A A) (Id A)
and Comp-Hom-Id: [ A ∈ Obj; B ∈ Obj; residuation.arr (Hom A B) t ]
  ⇒ Comp A A B t (Id A) = t
and Comp-Id-Hom: [ A ∈ Obj; B ∈ Obj; residuation.arr (Hom A B) u ]
  ⇒ Comp A B B (Id B) u = u

```

```

and Comp-assoc: [  $A \in \text{Obj}; B \in \text{Obj}; C \in \text{Obj}; D \in \text{Obj};$ 
  residuation.arr (Hom  $C D$ )  $t$ ; residuation.arr (Hom  $B C$ )  $u$ ;
  residuation.arr (Hom  $A B$ )  $v$  ]  $\implies$ 
  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  $\Rightarrow$  'O
  where Dom (MkArr  $a$  -) =  $a$ 
    | Dom - = undefined

  fun Cod :: ('O, 'A) arr  $\Rightarrow$  'O
  where Cod (MkArr -  $b$  -) =  $b$ 
    | Cod - = undefined

  fun Trn :: ('O, 'A) arr  $\Rightarrow$  'A
  where Trn (MkArr - -  $t$ ) =  $t$ 
    | Trn - = undefined

  abbreviation Arr :: ('O, 'A) arr  $\Rightarrow$  bool
  where Arr  $\equiv$   $\lambda t. t \neq \text{Null} \wedge \text{Dom } t \in \text{Obj} \wedge \text{Cod } t \in \text{Obj} \wedge$ 
    residuation.arr (Hom (Dom  $t$ ) (Cod  $t$ )) (Trn  $t$ )

  abbreviation Ide :: ('O, 'A) arr  $\Rightarrow$  bool
  where Ide  $\equiv$   $\lambda t. t \neq \text{Null} \wedge \text{Dom } t \in \text{Obj} \wedge \text{Cod } t \in \text{Obj} \wedge$ 
    residuation.ide (Hom (Dom  $t$ ) (Cod  $t$ )) (Trn  $t$ )

  abbreviation Con :: ('O, 'A) arr  $\Rightarrow$  ('O, 'A) arr  $\Rightarrow$  bool
  where Con  $t u \equiv$   $\text{Arr } t \wedge \text{Arr } u \wedge \text{Dom } t = \text{Dom } u \wedge \text{Cod } t = \text{Cod } u \wedge$ 
    residuation.con (Hom (Dom  $t$ ) (Cod  $t$ )) (Trn  $t$ ) (Trn  $u$ )

  fun resid (infix  $\langle \rangle$  70)
  where resid Null  $u = \text{Null}$ 
    | resid  $t \text{ Null} = \text{Null}$ 
    | 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
     $\langle$ proof $\rangle$ 

  lemma null-char:
  shows V.null = Null
     $\langle$ proof $\rangle$ 

```

sublocale V : *residuation resid*
 $\langle proof \rangle$

notation $V.con$ (**infix** $\langle \frown \rangle$ 50)

lemma *con-char*:
shows $t \frown u \longleftrightarrow Con\ t\ u$
 $\langle proof \rangle$

lemma *conI* [*intro*]:
assumes $Con\ t\ u$
shows $t \frown u$
 $\langle proof \rangle$

lemma *conE* [*elim*]:
assumes $t \frown u$
and $\llbracket V.arr\ t; Con\ t\ u \rrbracket \Longrightarrow T$
shows T
 $\langle proof \rangle$

lemma *arr-char*:
shows $V.arr\ t \longleftrightarrow Arr\ t$
 $\langle proof \rangle$

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 $V.arr\ t$
 $\langle proof \rangle$

lemma *arrE* [*elim*]:
assumes $V.arr\ t$
and $\llbracket t \neq V.null; Dom\ t \in Obj; Cod\ t \in Obj;$
 $residuation.arr\ (Hom\ (Dom\ t)\ (Cod\ t))\ (Trn\ t) \rrbracket$
 $\Longrightarrow T$
shows T
 $\langle proof \rangle$

lemma *sta-char*:
shows $V.ide\ t \longleftrightarrow Ide\ t$
 $\langle proof \rangle$

lemma *staI* [*intro*]:
assumes $t \neq V.null$ **and** $Dom\ t \in Obj$ **and** $Cod\ t \in Obj$
and $residuation.ide\ (Hom\ (Dom\ t)\ (Cod\ t))\ (Trn\ t)$
shows $V.ide\ t$
 $\langle proof \rangle$

lemma *staE* [*elim*]:
assumes $V.ide\ t$
and $[[t \neq V.null; Dom\ t \in Obj; Cod\ t \in Obj;$
 $residuation.ide\ (Hom\ (Dom\ t)\ (Cod\ t))\ (Trn\ t)]]$
 $\implies T$
shows T
 $\langle proof \rangle$

lemma *trg-char*:
shows $V.trg\ t =$
 $(if\ V.arr\ t$
 $then\ MkArr\ (Dom\ t)\ (Cod\ t)$
 $(residuation.trg\ (Hom\ (Dom\ t)\ (Cod\ t))\ (Trn\ t))$
 $else\ Null)$
 $\langle proof \rangle$

lemma *con-implies-Par*:
assumes $t \frown u$
shows $Dom\ t = Dom\ u$ **and** $Cod\ t = Cod\ u$
 $\langle proof \rangle$

lemma *Dom-resid* [*simp*]:
assumes $t \frown u$
shows $Dom\ (t \setminus u) = Dom\ t$
 $\langle proof \rangle$

lemma *Cod-resid* [*simp*]:
assumes $t \frown u$
shows $Cod\ (t \setminus u) = Cod\ t$
 $\langle proof \rangle$

lemma *Trn-resid* [*simp*]:
assumes $t \frown u$
shows $Trn\ (t \setminus u) = Hom\ (Dom\ t)\ (Cod\ t)\ (Trn\ t)\ (Trn\ u)$
 $\langle proof \rangle$

sublocale $V: rts\ resid$
 $\langle proof \rangle$

lemma *is-rts*:
shows $rts\ resid$
 $\langle proof \rangle$

sublocale $V: extensional-rts\ resid$
 $\langle proof \rangle$

lemma *is-extensional-rts*:
shows $extensional-rts\ resid$
 $\langle proof \rangle$

lemma *arr-MkArr* [*intro*]:
assumes $a \in \text{Obj}$ **and** $b \in \text{Obj}$
and *residuation.arr* (*Hom* a b) t
shows $V.\text{arr}$ (MkArr a b t)
 $\langle \text{proof} \rangle$

lemma *arr-eqI*:
assumes $t \neq \text{Null}$ **and** $u \neq \text{Null}$
and $\text{Dom } t = \text{Dom } u$ **and** $\text{Cod } t = \text{Cod } u$ **and** $\text{Trn } t = \text{Trn } u$
shows $t = u$
 $\langle \text{proof} \rangle$

lemma *MkArr-Trn*:
assumes $V.\text{arr } t$
shows MkArr ($\text{Dom } t$) ($\text{Cod } t$) ($\text{Trn } t$) = t
 $\langle \text{proof} \rangle$

lemma *src-char*:
shows $V.\text{src } t = (\text{if } V.\text{arr } t$
 $\quad \text{then } \text{MkArr}$ ($\text{Dom } t$) ($\text{Cod } t$)
 $\quad \quad (\text{rts.src}$ (Hom ($\text{Dom } t$) ($\text{Cod } t$)) ($\text{Trn } t$))
 $\quad \text{else } \text{Null})$
 $\langle \text{proof} \rangle$

definition *hcomp* (**infix** $\langle \star \rangle$ 53)
where $t \star u \equiv \text{if } V.\text{arr } t \wedge V.\text{arr } u \wedge \text{Dom } t = \text{Cod } u$
 $\quad \text{then } \text{MkArr}$ ($\text{Dom } u$) ($\text{Cod } t$)
 $\quad \quad (\text{Comp}$ ($\text{Dom } u$) ($\text{Cod } u$) ($\text{Cod } t$) ($\text{Trn } t$) ($\text{Trn } u$))
 $\quad \text{else } V.\text{null}$

lemma *arr-hcomp_{CRC}*:
assumes $V.\text{arr } t$ **and** $V.\text{arr } u$ **and** $\text{Dom } t = \text{Cod } u$
shows $V.\text{arr}$ ($t \star u$)
 $\langle \text{proof} \rangle$

lemma *Dom-hcomp* [*simp*]:
assumes $V.\text{arr } t$ **and** $V.\text{arr } u$ **and** $\text{Dom } t = \text{Cod } u$
shows Dom ($t \star u$) = $\text{Dom } u$
 $\langle \text{proof} \rangle$

lemma *Cod-hcomp* [*simp*]:
assumes $V.\text{arr } t$ **and** $V.\text{arr } u$ **and** $\text{Dom } t = \text{Cod } u$
shows Cod ($t \star u$) = $\text{Cod } t$
 $\langle \text{proof} \rangle$

lemma *Trn-hcomp* [*simp*]:
assumes $V.\text{arr } t$ **and** $V.\text{arr } u$ **and** $\text{Dom } t = \text{Cod } u$
shows Trn ($t \star u$) = Comp ($\text{Dom } u$) ($\text{Cod } u$) ($\text{Cod } t$) ($\text{Trn } t$) ($\text{Trn } u$)

⟨proof⟩

lemma *hcomp-Null* [simp]:

shows $t \star \text{Null} = \text{Null}$ **and** $\text{Null} \star u = \text{Null}$

⟨proof⟩

sublocale *H*: *Category.partial-magma hcomp*

⟨proof⟩

lemma *null-coincidence_{CR}* [simp]:

shows $H.\text{null} = V.\text{null}$

⟨proof⟩

sublocale *H*: *partial-composition hcomp* ⟨proof⟩

lemma *H-composable-char*:

shows $t \star u \neq V.\text{null} \iff V.\text{arr } t \wedge V.\text{arr } u \wedge \text{Dom } t = \text{Cod } u$

⟨proof⟩

lemma *objI* [intro]:

assumes $t \neq V.\text{null}$

and $\text{Dom } t \in \text{Obj}$ **and** $\text{Cod } t = \text{Dom } t$ **and** $\text{Trn } t = \text{Id } (\text{Dom } t)$

shows $H.\text{ide } t$

⟨proof⟩

lemma *objE* [elim]:

assumes $H.\text{ide } a$

and $\llbracket a \neq V.\text{null}; \text{Dom } a \in \text{Obj}; \text{Cod } a = \text{Dom } a; \text{Trn } a = \text{Id } (\text{Dom } a) \rrbracket \implies T$

shows T

⟨proof⟩

definition *mkobj*

where $\text{mkobj } A \equiv \text{MkArr } A \ A \ (\text{Id } A)$

lemma *mkobj-simps* [simp]:

shows $\text{Dom } (\text{mkobj } A) = A$ **and** $\text{Cod } (\text{mkobj } A) = A$

and $\text{Trn } (\text{mkobj } A) = \text{Id } A$

⟨proof⟩

lemma *obj-mkobj*:

assumes $A \in \text{Obj}$

shows $H.\text{ide } (\text{mkobj } A)$

⟨proof⟩

lemma *obj-char*:

shows $H.\text{ide } a \iff V.\text{arr } a \wedge \text{mkobj } (\text{Dom } a) = a$

⟨proof⟩

lemma *obj-is-sta*:

assumes $H.ide\ a$

shows $V.ide\ a$

$\langle proof \rangle$

lemma *obj-simps*:

assumes $H.ide\ a$

shows $Cod\ a = Dom\ a$ **and** $Trn\ a = Id\ (Dom\ a)$

$\langle proof \rangle$

lemma *domains-char*:

shows $H.domains\ t = \{a. V.arr\ t \wedge mkobj\ (Dom\ t) = a\}$

$\langle proof \rangle$

lemma *codomains-char*:

shows $H.codomains\ t = \{a. V.arr\ t \wedge mkobj\ (Cod\ t) = a\}$

$\langle proof \rangle$

lemma *H-arr-char*:

shows $H.arr\ t \longleftrightarrow t \neq Null \wedge Dom\ t \in Obj \wedge Cod\ t \in Obj \wedge$
residuation.arr $(Hom\ (Dom\ t)\ (Cod\ t))\ (Trn\ t)$

$\langle proof \rangle$

lemma *H-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 $H.arr\ t$

$\langle proof \rangle$

lemma *H-seq-char*:

shows $H.seq\ t\ u \longleftrightarrow V.arr\ t \wedge V.arr\ u \wedge Dom\ t = Cod\ u$

$\langle proof \rangle$

lemma *H-seqI* [*intro*]:

assumes $V.arr\ t$ **and** $V.arr\ u$ **and** $Dom\ t = Cod\ u$

shows $H.seq\ t\ u$

$\langle proof \rangle$

sublocale H : *category hcomp*

$\langle proof \rangle$

lemma *is-category*:

shows *category hcomp*

$\langle proof \rangle$

lemma *arr-coincidence_{CRC}* [*simp*]:

shows $H.arr = V.arr$

$\langle proof \rangle$

lemma *dom-char*:

shows $H.dom = (\lambda t. \text{if } H.arr\ t \text{ then } mkobj\ (Dom\ t) \text{ else } V.null)$
 $\langle proof \rangle$

lemma *dom-mkobj* [*simp*]:
assumes $A \in Obj$
shows $H.dom\ (mkobj\ A) = mkobj\ A$
 $\langle proof \rangle$

lemma *mkobj-Dom* [*simp*]:
assumes $H.ide\ a$
shows $mkobj\ (Dom\ a) = a$
 $\langle proof \rangle$

lemma *cod-char*:
shows $H.cod = (\lambda t. \text{if } H.arr\ t \text{ then } mkobj\ (Cod\ t) \text{ else } V.null)$
 $\langle proof \rangle$

lemma *cod-mkobj* [*simp*]:
assumes $A \in Obj$
shows $H.cod\ (mkobj\ A) = mkobj\ A$
 $\langle proof \rangle$

lemma *Dom-dom* [*simp*]:
assumes $V.arr\ t$
shows $Dom\ (H.dom\ t) = Dom\ t$
 $\langle proof \rangle$

lemma *Dom-cod* [*simp*]:
assumes $V.arr\ t$
shows $Dom\ (H.cod\ t) = Cod\ t$
 $\langle proof \rangle$

lemma *Cod-dom* [*simp*]:
assumes $V.arr\ t$
shows $Cod\ (H.dom\ t) = Dom\ t$
 $\langle proof \rangle$

lemma *Cod-cod* [*simp*]:
assumes $V.arr\ t$
shows $Cod\ (H.cod\ t) = Cod\ t$
 $\langle proof \rangle$

lemma *con-implies-par*:
assumes $V.con\ t\ u$
shows $H.par\ t\ u$
 $\langle proof \rangle$

lemma *par-resid*:
assumes $V.con\ t\ u$

shows $H.par\ t\ (resid\ t\ u)$
⟨*proof*⟩

lemma *simulation-dom*:
shows *simulation resid resid H.dom*
⟨*proof*⟩

lemma *simulation-cod*:
shows *simulation resid resid H.cod*
⟨*proof*⟩

sublocale *dom: simulation resid resid H.dom*
⟨*proof*⟩

sublocale *cod: simulation resid resid H.cod*
⟨*proof*⟩

sublocale $VV: fibred-product-rts\ resid\ resid\ resid\ H.dom\ H.cod$ ⟨*proof*⟩

sublocale $H_{VV}: simulation\ VV.resid\ resid$
⟨ $\lambda t. if\ VV.arr\ t\ then\ fst\ t\ \star\ snd\ t\ else\ V.null$ ⟩
⟨*proof*⟩

lemma *simulation-hcomp*:
shows *simulation VV.resid resid* (⟨ $\lambda t. if\ VV.arr\ t\ then\ fst\ t\ \star\ snd\ t\ else\ V.null$ ⟩)
⟨*proof*⟩

lemma *Dom-src [simp]*:
assumes $V.arr\ t$
shows $Dom\ (V.src\ t) = Dom\ t$
⟨*proof*⟩

lemma *Dom-trg [simp]*:
assumes $V.arr\ t$
shows $Dom\ (V.trg\ t) = Dom\ t$
⟨*proof*⟩

lemma *Cod-src [simp]*:
assumes $V.arr\ t$
shows $Cod\ (V.src\ t) = Cod\ t$
⟨*proof*⟩

lemma *Cod-trg [simp]*:
assumes $V.arr\ t$
shows $Cod\ (V.trg\ t) = Cod\ t$
⟨*proof*⟩

lemma *dom-src_{RC} [simp]*:
shows $H.dom\ (V.src\ t) = H.dom\ t$
⟨*proof*⟩

lemma *dom-trg_{CR}* [*simp*]:
shows $H.dom (V.trg t) = H.dom t$
 $\langle proof \rangle$

lemma *cod-src_{CR}* [*simp*]:
shows $H.cod (V.src t) = H.cod t$
 $\langle proof \rangle$

lemma *cod-trg_{CR}* [*simp*]:
shows $H.cod (V.trg t) = H.cod t$
 $\langle proof \rangle$

lemma *src-dom_{CR}* [*simp*]:
shows $V.src (H.dom t) = H.dom t$
 $\langle proof \rangle$

lemma *trg-dom_{CR}* [*simp*]:
shows $V.trg (H.dom t) = H.dom t$
 $\langle proof \rangle$

lemma *src-cod_{CR}* [*simp*]:
shows $V.src (H.cod t) = H.cod t$
 $\langle proof \rangle$

lemma *trg-cod_{CR}* [*simp*]:
shows $V.trg (H.cod t) = H.cod t$
 $\langle proof \rangle$

sublocale *rts-category resid hcomp*
 $\langle proof \rangle$

proposition *is-rts-category*:
shows *rts-category resid hcomp*
 $\langle proof \rangle$

The elements of the originally given set *Obj* are in bijective correspondence with the objects of the constructed RTS-category.

lemma *bij-mkobj*:
shows $mkobj \in Obj \rightarrow Collect\ obj$
and $Dom \in Collect\ obj \rightarrow Obj$
and $\bigwedge A. Dom (mkobj A) = A$
and $\bigwedge a. obj a \implies mkobj (Dom a) = a$
and *bij-betw* $mkobj\ Obj\ (Collect\ obj)$
and *bij-betw* $Dom\ (Collect\ obj)\ Obj$
 $\langle proof \rangle$

abbreviation $MkArr_{ext}$
where $MkArr_{ext} A B \equiv$

$\lambda t. \text{if residuation.arr (Hom A B) t then MkArr A B t else Null}$

abbreviation Trn_{ext}

where $\text{Trn}_{ext} a b \equiv$

$\lambda t. \text{if residuation.arr (HOM a b) t then Trn t}$
 $\text{else ResiduatedTransitionSystem.partial-magma.null}$
 $(\text{Hom (Dom a) (Dom b)})$

lemma *inverse-simulations-Trn-MkArr*:

assumes $A \in \text{Obj}$ and $B \in \text{Obj}$

shows $\text{inverse-simulations (Hom A B) (HOM (mkobj A) (mkobj B))}$
 $(\text{Trn}_{ext} (\text{mkobj A}) (\text{mkobj B})) (\text{MkArr}_{ext} A B)$

$\langle \text{proof} \rangle$

Each hom-RTS of the constructed RTS-category is isomorphic to the corresponding RTS given by *Hom*.

lemma *isomorphic-rts-Hom-HOM*:

assumes $A \in \text{Obj}$ and $B \in \text{Obj}$

shows $\text{isomorphic-rts (Hom A B) (HOM (mkobj A) (mkobj B))}$

$\langle \text{proof} \rangle$

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*

$\langle \text{TYPE}('A \text{ resid}) \rangle \langle \text{TYPE} (('A, 'A) \text{ exponential-rts.arr}) \rangle$

$\langle \text{Collect } \textit{extensional-rts} \cap \textit{Collect } \textit{small-rts} \rangle$
 $\langle \lambda A B. \textit{exponential-rts.resid } A B \rangle$
 $\langle \lambda A. \textit{exponential-rts.MkArr } (I A) (I A) (I A) \rangle$
 $\langle \lambda A B C f g. \textit{COMP.map } A B C (f, g) \rangle$
 $\langle \textit{proof} \rangle$

type-synonym $'a \textit{arr} =$
 $('a \textit{resid}, ('a, 'a) \textit{exponential-rts.arr}) \textit{concrete-rts-category.arr}$

notation \textit{resid} (**infix** $\langle \rangle$ 70)
notation \textit{hcomp} (**infixr** $\langle \star \rangle$ 53)

The mapping \textit{Trn} that takes arrow $t \in H.\textit{hom } a b$ to the underlying transition of the exponential RTS $[\textit{Dom } a, \textit{Dom } b]$, is injective.

lemma $\textit{inj-Trn}$:
assumes $\textit{obj } a$ **and** $\textit{obj } b$
shows $\textit{Trn} \in H.\textit{hom } a b \rightarrow$
 $\textit{Collect } (\textit{residuation.arr } (\textit{exponential-rts.resid } (\textit{Dom } a) (\textit{Dom } b)))$
and $\textit{inj-on } \textit{Trn } (H.\textit{hom } a b)$
 $\langle \textit{proof} \rangle$

sublocale $\textit{locally-small-rts-category resid hcomp}$
 $\langle \textit{proof} \rangle$

abbreviation $\textit{sta-in-hom}$ ($\langle \langle - : - \rightarrow_{\textit{sta}} - \rangle \rangle$)
where $\textit{sta-in-hom } f a b \equiv H.\textit{in-hom } f a b \wedge \textit{sta } f$

abbreviation $\textit{trn-to}$ ($\langle \langle - : - \Rightarrow - \rangle \rangle$)
where $\textit{trn-to } t f g \equiv \textit{arr } t \wedge \textit{src } t = f \wedge \textit{trg } t = g$

definition $\textit{mkarr} :: 'A \textit{resid} \Rightarrow 'A \textit{resid} \Rightarrow$
 $('A \Rightarrow 'A) \Rightarrow ('A \Rightarrow 'A) \Rightarrow ('A \Rightarrow 'A) \Rightarrow$
 $'A \textit{arr}$
where $\textit{mkarr } A B F G \tau \equiv$
 $\textit{MkArr } A B (\textit{exponential-rts.MkArr } F G \tau)$

abbreviation \textit{mksta}
where $\textit{mksta } A B F \equiv \textit{mkarr } A B F F F$

lemma $\textit{Dom-mkarr}$ [\textit{simp}]:
shows $\textit{Dom } (\textit{mkarr } A B F G \tau) = A$
 $\langle \textit{proof} \rangle$

lemma $\textit{Cod-mkarr}$ [\textit{simp}]:
shows $\textit{Cod } (\textit{mkarr } A B F G \tau) = B$
 $\langle \textit{proof} \rangle$

lemma $\textit{arr-mkarr}$ [\textit{intro}]:
assumes $\textit{small-rts } A$ **and** $\textit{extensional-rts } A$

and *small-rts* B **and** *extensional-rts* B
and *transformation* $A B F G \tau$
shows $\text{arr } (mkarr A B F G \tau)$
and $\text{src } (mkarr A B F G \tau) = mksta A B F$
and $\text{trg } (mkarr A B F G \tau) = mksta A B G$
and $\text{dom } (mkarr A B F G \tau) = mkobj A$
and $\text{cod } (mkarr A B F G \tau) = mkobj B$
 $\langle proof \rangle$

lemma *mkarr-simps* [*simp*]:
assumes $\text{arr } (mkarr A B F G \sigma)$
shows $\text{dom } (mkarr A B F G \sigma) = mkobj A$
and $\text{cod } (mkarr A B F G \sigma) = mkobj B$
and $\text{src } (mkarr A B F G \sigma) = mksta A B F$
and $\text{trg } (mkarr A B F G \sigma) = mksta A B G$
 $\langle proof \rangle$

lemma *mkarr-in-hom* [*intro*]:
assumes *obj* a **and** *obj* b
and $A = \text{Dom } a$ **and** $B = \text{Dom } b$
and *simulation* $A B F$
and *simulation* $A B G$
and *transformation* $A B F G \tau$
shows $\langle mkarr A B F G \tau : a \rightarrow b \rangle$
 $\langle proof \rangle$

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 $\text{sta } (mksta A B F)$
and $\text{dom } (mksta A B F) = mkobj A$ **and** $\text{cod } (mksta A B F) = mkobj B$
 $\langle proof \rangle$

abbreviation *Src*
where $\text{Src} \equiv \text{exponential-rts.Dom} \circ \text{Trn}$

abbreviation *Trg*
where $\text{Trg} \equiv \text{exponential-rts.Cod} \circ \text{Trn}$

abbreviation *Map*
where $\text{Map} \equiv \text{exponential-rts.Map} \circ \text{Trn}$

lemma *Src-mkarr* [*simp*]:
assumes $\text{arr } (mkarr A B F G \sigma)$
shows $\text{Src } (mkarr A B F G \sigma) = F$
 $\langle proof \rangle$

lemma *Trg-mkarr* [*simp*]:

assumes arr ($mkarr$ A B F G σ)
shows Trg ($mkarr$ A B F G σ) = G
 $\langle proof \rangle$

lemma Map - $mkarr$ [$simp$]:
assumes arr ($mkarr$ A B F G σ)
shows Map ($mkarr$ A B F G σ) = σ
 $\langle proof \rangle$

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
 $\langle proof \rangle$

lemma src - $simp$:
assumes arr t
shows src t = $mksta$ (Dom t) (Cod t) (Src t)
 $\langle proof \rangle$

lemma trg - $simp$:
assumes arr t
shows trg t = $mksta$ (Dom t) (Cod t) (Trg t)
 $\langle proof \rangle$

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 \in $Collect$ ($transformation$ A B F G)
 $\rightarrow \{t. \langle t : mksta$ A B F \Rightarrow $mksta$ A B $G \rangle\}$
and Map \in $\{t. \langle t : mksta$ A B F \Rightarrow $mksta$ A B $G \rangle\}$
 \rightarrow $Collect$ ($transformation$ A B F G)
and [$simp$]: Map ($mkarr$ A B F G τ) = τ
and [$simp$]: $t \in \{t. \langle t : mksta$ A B F \Rightarrow $mksta$ A B $G \rangle\}$
 \implies $mkarr$ A B F G (Map t) = t
and bij - $betw$ ($mkarr$ A B F G) ($Collect$ ($transformation$ A B F G))
 $\{t. \langle t : mksta$ A B F \Rightarrow $mksta$ A B $G \rangle\}$
and bij - $betw$ Map $\{t. \langle t : mksta$ A B F \Rightarrow $mksta$ A B $G \rangle\}$
 $(Collect$ ($transformation$ A B F G))
 $\langle proof \rangle$

lemma bij - $mksta$:
assumes $small$ - rts A **and** $extensional$ - rts A

and *small-rts B and extensional-rts B*
shows $mksta\ A\ B \in Collect\ (simulation\ A\ B)$
 $\rightarrow \{t.\ \langle t : mkobj\ A \rightarrow_{sta}\ mkobj\ B \rangle\}$
and $Map \in \{t.\ \langle t : mkobj\ A \rightarrow_{sta}\ mkobj\ B \rangle\}$
 $\rightarrow Collect\ (simulation\ A\ B)$
and $[simp]: Map\ (mksta\ A\ B\ F) = F$
and $[simp]: t \in \{t.\ \langle t : mkobj\ A \rightarrow_{sta}\ mkobj\ B \rangle\}$
 $\implies mksta\ A\ B\ (Map\ t) = t$
and $bij\ betw\ (mksta\ A\ B)\ (Collect\ (simulation\ A\ B))$
 $\{t.\ \langle t : mkobj\ A \rightarrow_{sta}\ mkobj\ B \rangle\}$
and $bij\ betw\ Map\ \{t.\ \langle t : mkobj\ A \rightarrow_{sta}\ mkobj\ B \rangle\}$
 $(Collect\ (simulation\ A\ B))$
 $\langle proof \rangle$

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 $mkarr\ A\ C\ (H \circ F)\ (K \circ G)\ (\tau \circ \sigma) =$
 $mkarr\ B\ C\ H\ K\ \tau \star mkarr\ A\ B\ F\ G\ \sigma$
 $\langle proof \rangle$

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 $mkarr\ A\ B\ F\ G\ \sigma \frown mkarr\ A\ B\ F\ H\ \tau$
and $mkarr\ A\ B\ H\ (consistent\ transformations.\ apex\ A\ B\ H\ \sigma\ \tau)$
 $(consistent\ transformations.\ resid\ A\ B\ H\ \sigma\ \tau) =$
 $mkarr\ A\ B\ F\ G\ \sigma \setminus mkarr\ A\ B\ F\ H\ \tau$
 $\langle proof \rangle$

lemma *Dom-hcomp_X*:
assumes $H.seq\ t\ u$
shows $Dom\ (t \star u) = Dom\ u$
 $\langle proof \rangle$

lemma *Cod-hcomp_X*:
assumes $H.seq\ t\ u$
shows $Cod\ (t \star u) = Cod\ t$
 $\langle proof \rangle$

lemma *Map-hcomp*:
assumes $H.seq\ t\ u$
shows $Map\ (t \star u) = Map\ t \circ Map\ u$
 $\langle proof \rangle$

lemma *Src-hcomp*:
assumes $H.seq\ t\ u$
shows $Src\ (t \star u) = Src\ t \circ Src\ u$
 $\langle proof \rangle$

lemma *Trg-hcomp*:
assumes $H.seq\ t\ u$
shows $Trg\ (t \star u) = Trg\ t \circ Trg\ u$
 $\langle proof \rangle$

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 $Map\ (t \setminus u) =$
 $consistent-transformations.resid\ (Dom\ t)\ (Cod\ t)\ (Trg\ u)$
 $(Map\ t)\ (Map\ u)$
 $\langle proof \rangle$

lemma *Src-resid*:
assumes $V.con\ t\ u$
shows $Src\ (t \setminus u) = Trg\ u$
 $\langle proof \rangle$

lemma *Trg-resid*:
assumes $V.con\ t\ u$
shows $Trg\ (t \setminus u) = consistent-transformations.apex\ (Dom\ t)\ (Cod\ t)$
 $(Trg\ u)\ (Map\ t)\ (Map\ u)$
 $\langle proof \rangle$

lemma *simulation-Map*:
assumes $sta\ f$
shows $simulation\ (Dom\ f)\ (Cod\ f)\ (Map\ f)$
 $\langle proof \rangle$

lemma *transformation-Map*:
assumes $arr\ t$
shows $transformation\ (Dom\ t)\ (Cod\ t)\ (Src\ t)\ (Trg\ t)\ (Map\ t)$
 $\langle proof \rangle$

lemma *arr-eqI'*:
assumes $arr\ t$ **and** $arr\ u$
and $Dom\ t = Dom\ u$ **and** $Cod\ t = Cod\ u$
and $Src\ t = Src\ u$ **and** $Trg\ t = Trg\ u$
and $\bigwedge a. residuation.ide\ (Dom\ t)\ a \implies Map\ t\ a = Map\ u\ a$
shows $t = u$
 $\langle proof \rangle$

lemma *iso-char*:

shows $H.iso\ t \iff arr\ t \wedge Src\ t = Map\ t \wedge Trg\ t = Map\ t \wedge$
 $invertible-simulation\ (Dom\ t)\ (Cod\ t)\ (Map\ t)$

$\langle proof \rangle$

lemma *inverse-arrows-char*:

shows $H.inverse-arrows\ t\ u \iff$
 $sta\ t \wedge sta\ u \wedge H.antipar\ t\ u \wedge$
 $inverse-simulations\ (Dom\ t)\ (Dom\ u)\ (Map\ u)\ (Map\ t)$

$\langle proof \rangle$

lemma *inv-char*:

assumes $H.iso\ t$
shows $H.inv\ t = mksta\ (Cod\ t)\ (Dom\ t)$
 $(inverse-simulation.map\ (Dom\ t)\ (Cod\ t)\ (Map\ t))$

$\langle proof \rangle$

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* $\langle proof \rangle$

interpretation I_1 : *identity-simulation One.resid* $\langle proof \rangle$

abbreviation *one* $\langle 1 \rangle$

where $one \equiv mkobj\ One.resid$

lemma *obj-one*:

shows $obj\ 1$

$\langle proof \rangle$

definition *trm*

where $trm\ a \equiv MkArr\ (Dom\ a)\ One.resid$
 $(exponential-rts.MkIde$
 $(constant-simulation.map\ (Dom\ a)\ One.resid\ One.the-arr))$

lemma *one-universality*:

assumes $obj\ a$

shows $\langle trm\ a : a \rightarrow 1 \rangle$

and $\bigwedge t. \langle t : a \rightarrow 1 \rangle \implies t = trm\ a$

and $\exists! t. \langle t : a \rightarrow 1 \rangle$

$\langle proof \rangle$

lemma *terminal-one*:

shows $H.\text{terminal } \mathbf{1}$

$\langle \text{proof} \rangle$

lemma *trm-in-hom* [*intro, simp*]:

assumes *obj a*

shows $\langle \text{trm } a : a \rightarrow \mathbf{1} \rangle$

$\langle \text{proof} \rangle$

lemma *terminal-arrow-is-sta*:

assumes $\langle t : a \rightarrow \mathbf{1} \rangle$

shows *sta t*

$\langle \text{proof} \rangle$

For any object a we have an RTS isomorphism $\text{Dom } a \cong \text{HOM } \mathbf{1} a$. Note that these are *not* at the same type.

abbreviation $UP :: 'A \text{ arr} \Rightarrow 'A \Rightarrow 'A \text{ arr}$

where $UP a \equiv \text{MkArr}_{\text{ext}} (\backslash_1) (\text{Dom } a) \circ \text{exponential-by-One.Up } (\text{Dom } a)$

abbreviation $DN :: 'A \text{ arr} \Rightarrow 'A \text{ arr} \Rightarrow 'A$

where $DN a \equiv \text{exponential-by-One.Dn } (\text{Dom } a) \circ \text{Trn}_{\text{ext}} \mathbf{1} a$

lemma *inverse-simulations-DN-UP*:

assumes *obj a*

shows *inverse-simulations* $(\text{Dom } a) (\text{HOM } \mathbf{1} a) (DN a) (UP a)$

and *isomorphic-rts* $(\text{Dom } a) (\text{HOM } \mathbf{1} a)$

$\langle \text{proof} \rangle$

lemma *terminal-char*:

shows $H.\text{terminal } x \longleftrightarrow \text{obj } x \wedge (\exists ! t. \text{residuation.arr } (\text{Dom } x) t)$

$\langle \text{proof} \rangle$

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* $\langle \text{proof} \rangle$

definition *one* $\langle \mathbf{1} \rangle$

where $\text{one} \equiv \text{terminal-object-in-rtscat.one}$

definition *trm*

where $\text{trm} = \text{terminal-object-in-rtscat.trm}$

interpretation *Trm*: *terminal-object-in-rtscat* $\langle \text{proof} \rangle$

no-notation *Trm.one* ($\langle 1 \rangle$)

lemma *obj-one* [*intro*, *simp*]:
shows *obj one*
 $\langle proof \rangle$

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)*
 $\langle proof \rangle$

sublocale *category-with-terminal-object hcomp*
 $\langle proof \rangle$

sublocale *elementary-category-with-terminal-object hcomp one trm*
 $\langle proof \rangle$

lemma *is-elementary-category-with-terminal-object*:
shows *elementary-category-with-terminal-object hcomp one trm*
 $\langle proof \rangle$

lemma *terminal-char*:
shows *H.terminal x* \longleftrightarrow *obj x* \wedge ($\exists ! t. \text{residuation.arr (Dom x) t}$)
 $\langle proof \rangle$

lemma *Map-trm*:
assumes *obj a*
shows *Map (trm a) =*
constant-simulation.map (Dom a) One.resid One.the-arr
 $\langle proof \rangle$

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)*
 $\langle proof \rangle$

abbreviation $UP_{rts} :: 'A \text{ arr} \Rightarrow 'A \Rightarrow 'A \text{ arr}$
where $UP_{rts} a \equiv \text{MkArr}_{ext} (\backslash_1) (\text{Dom } a) \circ \text{exponential-by-One.Up } (\text{Dom } a)$

abbreviation $DN_{rts} :: 'A \text{ arr} \Rightarrow 'A \text{ arr} \Rightarrow 'A$
where $DN_{rts} a \equiv \text{exponential-by-One.Dn } (\text{Dom } a) \circ \text{Trn}_{ext} \mathbf{1} a$

lemma *UP-DN-naturality*:
assumes *arr t*
shows $DN_{rts} (\text{cod } t) \circ \text{cov-HOM } \mathbf{1} t = \text{Map } t \circ DN_{rts} (\text{dom } t)$
and $UP_{rts} (\text{cod } t) \circ \text{Map } t = \text{cov-HOM } \mathbf{1} t \circ UP_{rts} (\text{dom } t)$

and $\text{cov-HOM } \mathbf{1} \ t = \text{UP}_{rts} (\text{cod } t) \circ \text{Map } t \circ \text{DN}_{rts} (\text{dom } t)$
and $\text{Map } t = \text{DN}_{rts} (\text{cod } t) \circ \text{cov-HOM } \mathbf{1} \ t \circ \text{UP}_{rts} (\text{dom } t)$
 $\langle \text{proof} \rangle$

Equality of parallel arrows $\langle u : a \rightarrow b \rangle$ and $\langle v : a \rightarrow b \rangle$ is determined by their compositions with global transitions $\langle t : \mathbf{1} \rightarrow a \rangle$.

lemma *arr-extensionality*:

assumes $\langle u : a \rightarrow b \rangle$ **and** $\langle v : a \rightarrow b \rangle$ **and** $\text{src } u = \text{src } v$ **and** $\text{trg } u = \text{trg } v$
shows $u = v \iff (\forall t. \langle t : \mathbf{1} \rightarrow a \rangle \longrightarrow u \star t = v \star t)$
 $\langle \text{proof} \rangle$

lemma *sta-extensionality*:

assumes $\langle f : a \rightarrow_{sta} b \rangle$ **and** $\langle g : a \rightarrow_{sta} b \rangle$
shows $f = g \iff (\forall t. \langle t : \mathbf{1} \rightarrow a \rangle \longrightarrow f \star t = g \star t)$
 $\langle \text{proof} \rangle$

The mapping $\text{HOM } \mathbf{1}$, like Dom , takes each object to a corresponding RTS, but unlike Dom it stays at type $'A \text{ arr}$, rather than decreasing the type from $'A \text{ arr}$ to $'A$.

lemma *HOM1-mapsto*:

shows $\text{HOM } \mathbf{1} \in \text{Collect } \text{obj} \rightarrow \text{Collect } \text{extensional-rts} \cap \text{Collect } \text{small-rts}$
 $\langle \text{proof} \rangle$

The mapping $\text{HOM } \mathbf{1}$ is not necessarily injective, but it is essentially so.

lemma *HOM1-reflects-isomorphic*:

assumes $\text{obj } a$ **and** $\text{obj } b$ **and** $\text{isomorphic-rts } (\text{HOM } \mathbf{1} \ a) \ (\text{HOM } \mathbf{1} \ b)$
shows $H.\text{isomorphic } a \ b$
 $\langle \text{proof} \rangle$

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 $\text{Dom } a \times \text{Dom } b$ of their underlying RTS's and then showing that there exists an object $a \otimes b$ such that $\text{Dom } (a \otimes b)$ is isomorphic to $\text{Dom } a \times \text{Dom } b$. Since $\text{Dom } (a \otimes b)$ will have arrow type $'A$, but $\text{Dom } a \times \text{Dom } b$ has arrow type $'A \times 'A$, we need a way to reduce the arrow type of $\text{Dom } a \otimes \text{Dom } b$ 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 $\text{Dom } a \otimes \text{Dom } b$.

locale *product-in-rtscat* =
 $\text{rtscatx } \text{arr-type}$
for $\text{arr-type} :: 'A \text{ itself}$
and $a :: 'A \text{ rtscatx.arr}$

and $b :: 'A \text{ rtscatx.arr} +$
assumes $\text{obj-a: obj } a$
and $\text{obj-b: obj } b$
begin

notation hcomp (**infixr** $\langle \star \rangle$ 53)

interpretation A : *extensional-rts* $\langle \text{Dom } a \rangle$
 $\langle \text{proof} \rangle$

interpretation A : *small-rts* $\langle \text{Dom } a \rangle$
 $\langle \text{proof} \rangle$

interpretation B : *extensional-rts* $\langle \text{Dom } b \rangle$
 $\langle \text{proof} \rangle$

interpretation B : *small-rts* $\langle \text{Dom } b \rangle$
 $\langle \text{proof} \rangle$

interpretation AB : *exponential-rts* $\langle \text{Dom } a \rangle \langle \text{Dom } b \rangle \langle \text{proof} \rangle$

sublocale $PROD$: *product-rts* $\langle \text{Dom } a \rangle \langle \text{Dom } b \rangle \langle \text{proof} \rangle$
sublocale $PROD$: *product-of-extensional-rts* $\langle \text{Dom } a \rangle \langle \text{Dom } b \rangle \langle \text{proof} \rangle$
sublocale $PROD$: *product-of-small-rts* $\langle \text{Dom } a \rangle \langle \text{Dom } b \rangle \langle \text{proof} \rangle$

sublocale $Prod$: *inj-image-rts pairing.some-pair* $PROD.resid$
 $\langle \text{proof} \rangle$

sublocale $Prod$: *small-rts* $Prod.resid$
 $\langle \text{proof} \rangle$

sublocale $Prod$: *extensional-rts* $Prod.resid$
 $\langle \text{proof} \rangle$

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* $Prod.resid$ $PROD.resid$ $\langle \text{Dom } a \rangle$
 $Unpack$ $PROD.P_1$
 $\langle \text{proof} \rangle$

interpretation P_0 : *composite-simulation* $Prod.resid$ $PROD.resid$ $\langle \text{Dom } b \rangle$
 $Unpack$ $PROD.P_0$
 $\langle \text{proof} \rangle$

abbreviation $prod :: 'A \text{ arr}$
where $prod \equiv mkobj$ $Prod.resid$

lemma obj-prod :
shows $\text{obj } prod$
 $\langle \text{proof} \rangle$

lemma *Dom-prod* [*simp*]:
shows $Dom\ prod = Prod.resid$
 $\langle proof \rangle$

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 *p₀-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 $Trn\ p_0 = exponential-rts.MkIde\ P_0.map$
 $\langle proof \rangle$

lemma *p₁-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 $Trn\ p_1 = exponential-rts.MkIde\ P_1.map$
 $\langle proof \rangle$

lemma *p₀-in-hom* [*intro*]:
shows $\langle p_0 : prod \rightarrow b \rangle$
 $\langle proof \rangle$

lemma *p₁-in-hom* [*intro*]:
shows $\langle p_1 : prod \rightarrow a \rangle$
 $\langle proof \rangle$

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 $\langle h : x \rightarrow a \rangle$ **and** $\langle k : x \rightarrow b \rangle$
shows $\exists! m. p_1 \star m = h \wedge p_0 \star m = k$
 $\langle proof \rangle$

lemma *has-as-binary-product*:
shows $H.has-as-binary-product\ a\ b\ p_1\ p_0$

<proof>

sublocale *binary-product hcomp a b p₁ p₀*
<proof>

lemma *preserves-extensional-rts:*
assumes *extensional-rts (Dom a)* **and** *extensional-rts (Dom b)*
shows *extensional-rts Prod.resid*
<proof>

lemma *preserves-small-rts:*
assumes *small-rts (Dom a)* **and** *small-rts (Dom b)*
shows *small-rts Prod.resid*
<proof>

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>

lemma *Map-p₀:*
shows *Map p₀ = PROD.P₀ ◦ Unpack*
<proof>

lemma *Map-p₁:*
shows *Map p₁ = PROD.P₁ ◦ Unpack*
<proof>

lemma *Map-tuple:*
assumes *«t : x → a»* **and** *«u : x → b»*
shows *Map (tuple t u) = Pack ◦ ⟨⟨Map t, Map u⟩⟩*
<proof>

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)*

⟨proof⟩

lemma *sta-p1*:
assumes *obj a and obj b*
shows *sta (p1 a b)*
⟨proof⟩

lemma *has-binary-products_X*:
assumes *obj a and obj b*
shows *H.has-as-binary-product a b (p1 a b) (p0 a b)*
⟨proof⟩

sublocale *category-with-binary-products hcomp*
⟨proof⟩

proposition *is-category-with-binary-products_X*:
shows *category-with-binary-products hcomp*
⟨proof⟩

lemma *extends-to-elementary-category-with-binary-products_X*:
shows *elementary-category-with-binary-products hcomp p0 p1*
⟨proof⟩

sublocale *elementary-category-with-binary-products hcomp p0 p1*
⟨proof⟩

notation *p0* (⟨p0[-, -]⟩)
notation *p1* (⟨p1[-, -]⟩)
notation *tuple* (⟨⟨-, -⟩⟩)
notation *prod* (**infixr** ⟨⊗⟩ 51)

lemma *prod-eq*:
assumes *obj a and obj b*
shows *a ⊗ b = product-in-rtscat.prod a b*
⟨proof⟩

lemma *sta-tuple [simp]*:
assumes *H.span t u and sta t and sta u*
shows *sta ⟨t, u⟩*
⟨proof⟩

lemma *sta-prod*:
assumes *sta t and sta u*
shows *sta (t ⊗ u)*
⟨proof⟩

The “product constraints” *Pack a b* and *Unpack a b* were originally derived from the arbitrarily chosen “type-reducing injection” *pairing.some-pair*.

However, now that the existence of products has been shown, we can express these constraints in terms of *Map* and the projections. We use this for the final definitions of *Pack a b* and *Unpack a b* to emphasize the significance of the chosen product structure over the *ad hoc* function *pairing.some-pair* used to show its existence.

definition *Unpack* :: 'A arr ⇒ 'A arr ⇒ 'A ⇒ 'A × 'A
where *Unpack a b* ≡ ⟨⟨*Map* (*p*₁ a b), *Map* (*p*₀ a b)⟩⟩

definition *Pack* :: 'A arr ⇒ 'A arr ⇒ 'A × 'A ⇒ 'A
where *Pack a b* ≡ *inverse-simulation.map* (*Dom* (a ⊗ b)) (*Dom* a ⊗ *Dom* b)
(*Unpack a b*)

lemma *Map-p*₀:
assumes *obj a* and *obj b*
shows *Map* p₀[a, b] = *product-rts.P*₀ (*Dom* a) (*Dom* b) ∘ *Unpack a b*
⟨*proof*⟩

lemma *Map-p*₁:
assumes *obj a* and *obj b*
shows *Map* p₁[a, b] = *product-rts.P*₁ (*Dom* a) (*Dom* b) ∘ *Unpack a b*
⟨*proof*⟩

lemma *Unpack-eq*:
assumes *obj a* and *obj b*
shows *Unpack a b* = *product-in-rtscat.Unpack a b*
⟨*proof*⟩

lemma *invertible-simulation-Unpack*:
assumes *obj a* and *obj b*
shows *invertible-simulation* (*Dom* (a ⊗ b)) (*Dom* a ⊗ *Dom* b)
⟨⟨*Map* (*p*₁ a b), *Map* (*p*₀ a b)⟩⟩
⟨*proof*⟩

lemma *inverse-simulations-Pack-Unpack*:
assumes *obj a* and *obj b*
shows *inverse-simulations* (*Dom* (a ⊗ b)) (*Dom* a ⊗ *Dom* b)
(*Pack a b*) (*Unpack a b*)
⟨*proof*⟩

lemma *simulation-Pack*:
assumes *obj a* and *obj b*
shows *simulation* (*Dom* a ⊗ *Dom* b) (*Dom* (a ⊗ b)) (*Pack a b*)
⟨*proof*⟩

lemma *simulation-Unpack*:
assumes *obj a* and *obj b*
shows *simulation* (*Dom* (a ⊗ b)) (*Dom* a ⊗ *Dom* b) (*Unpack a b*)
⟨*proof*⟩

lemma *Pack-o-Unpack*:
assumes *obj a* **and** *obj b*
shows $\text{Pack } a \ b \circ \text{Unpack } a \ b = I \ (\text{Dom } (a \otimes b))$
 $\langle \text{proof} \rangle$

lemma *Unpack-o-Pack*:
assumes *obj a* **and** *obj b*
shows $\text{Unpack } a \ b \circ \text{Pack } a \ b = I \ (\text{Dom } a \ \otimes \ \text{Dom } b)$
 $\langle \text{proof} \rangle$

lemma *Pack-Unpack [simp]*:
assumes *obj a* **and** *obj b*
and *residuation.arr* $(\text{Dom } (a \otimes b)) \ t$
shows $\text{Pack } a \ b \ (\text{Unpack } a \ b \ t) = t$
 $\langle \text{proof} \rangle$

lemma *Unpack-Pack [simp]*:
assumes *obj a* **and** *obj b*
and *residuation.arr* $(\text{Dom } a \ \otimes \ \text{Dom } b) \ t$
shows $\text{Unpack } a \ b \ (\text{Pack } a \ b \ t) = t$
 $\langle \text{proof} \rangle$

lemma *src-tuple [simp]*:
assumes *H.span* $t \ u$
shows $\text{src } \langle t, u \rangle = \langle \text{src } t, \text{src } u \rangle$
 $\langle \text{proof} \rangle$

lemma *trg-tuple [simp]*:
assumes *H.span* $t \ u$
shows $\text{trg } \langle t, u \rangle = \langle \text{trg } t, \text{trg } u \rangle$
 $\langle \text{proof} \rangle$

lemma *Map-tuple*:
assumes $\langle t : x \rightarrow a \rangle$ **and** $\langle u : x \rightarrow b \rangle$
shows $\text{Map } \langle t, u \rangle = \text{Pack } a \ b \circ \langle \langle \text{Map } t, \text{Map } u \rangle \rangle$
 $\langle \text{proof} \rangle$

lemma *Map-prod*:
assumes $\langle t : x \rightarrow a \rangle$ **and** $\langle u : y \rightarrow b \rangle$
shows $\text{Map } (t \ \otimes \ u) =$
 $\text{Pack } a \ b \ \circ$
 $\text{product-transformation.map}$
 $(\text{Dom } t) \ (\text{Dom } u) \ (\text{Cod } t) \ (\text{Cod } u) \ (\text{Src } t) \ (\text{Src } u) \ (\text{Map } t) \ (\text{Map } u) \ \circ$
 $\text{Unpack } x \ y$
 $\langle \text{proof} \rangle$

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]⟩
  ⟨proof⟩

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⟩

```

```

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

```

$\langle proof \rangle$

notation $hcomp$ (**infixr** $\langle \star \rangle$ 53)
notation p_0 ($\langle p_0[-, -] \rangle$)
notation p_1 ($\langle p_1[-, -] \rangle$)
notation $tuple$ ($\langle \langle -, - \rangle \rangle$)
notation $prod$ (**infixr** $\langle \otimes \rangle$ 51)

sublocale B : *extensional-rts* $\langle Dom\ b \rangle$

$\langle proof \rangle$

sublocale B : *small-rts* $\langle Dom\ b \rangle$

$\langle proof \rangle$

sublocale C : *extensional-rts* $\langle Dom\ c \rangle$

$\langle proof \rangle$

sublocale C : *small-rts* $\langle Dom\ c \rangle$

$\langle proof \rangle$

sublocale EXP : *exponential-rts* $\langle Dom\ b \rangle \langle Dom\ c \rangle \langle proof \rangle$

sublocale EXP : *exponential-of-small-rts* $\langle Dom\ b \rangle \langle Dom\ c \rangle \langle proof \rangle$

lemma *small-function-Map*:

assumes $EXP.arr\ t$

shows *small-function* $(EXP.Dom\ t)$ **and** *small-function* $(EXP.Cod\ t)$

and *small-function* $(EXP.Map\ t)$

$\langle proof \rangle$

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* $inj-exp\ EXP.resid$

$\langle proof \rangle$

sublocale Exp : *extensional-rts* $Exp.resid$

$\langle proof \rangle$

sublocale Exp : *small-rts* $Exp.resid$

$\langle proof \rangle$

lemma *is-extensional-rts*:

shows *extensional-rts* $Exp.resid$

$\langle proof \rangle$

lemma *is-small-rts*:

shows *small-rts* $Exp.resid$

$\langle proof \rangle$

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 $obj\ exp$

$\langle proof \rangle$

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 $Dom\ exp = Exp.resid$

$\langle proof \rangle$

sublocale $EXPxB$: *product-rts* $EXP.resid \langle Dom\ b \rangle \langle proof \rangle$

sublocale $ExpxB$: *product-rts* $Exp.resid \langle Dom\ b \rangle \langle proof \rangle$

sublocale B : *identity-simulation* $\langle Dom\ b \rangle \langle proof \rangle$

sublocale B : *simulation-as-transformation* $\langle Dom\ b \rangle \langle Dom\ b \rangle B.map \langle proof \rangle$

sublocale B : *transformation-to-extensional-rts*
 $\langle Dom\ b \rangle \langle Dom\ b \rangle B.map\ B.map\ B.map \langle proof \rangle$

sublocale $UnfuncxB$: *product-simulation*
 $EXP.resid \langle Dom\ b \rangle Exp.resid \langle Dom\ b \rangle Unfunc' B.map \langle proof \rangle$

sublocale $FuncxB$: *product-simulation*
 $Exp.resid \langle Dom\ b \rangle EXP.resid \langle Dom\ b \rangle Func' B.map \langle proof \rangle$

sublocale *inverse-simulations* $EXPxB.resid\ ExpxB.resid$

$FuncxB.map\ UnfuncxB.map$

$\langle proof \rangle$

lemma $obj-expxb$:

shows $obj\ (exp \otimes b)$

$\langle proof \rangle$

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*

$\langle Dom\ (exp \otimes b) \rangle ExpxB.resid\ EXPxB.resid$

$\langle Unpack\ exp\ b \rangle FuncxB.map$

⟨proof⟩

We construct the evaluation map associated with $ExpB$ by composing the evaluation map $Eval.map$ from $ExpB$ to C , derived from the exponential RTS construction, with the isomorphism $FuncB-o-Unpack$ from $Dom\ expb.prod$ to $ExpB$ and then obtain the corresponding arrow of the category.

sublocale $Eval$: *evaluation-map* ⟨ $Dom\ b$ ⟩ ⟨ $Dom\ c$ ⟩ ⟨proof⟩

sublocale $Eval$: *evaluation-map-between-extensional-rts* ⟨ $Dom\ b$ ⟩ ⟨ $Dom\ c$ ⟩
⟨proof⟩

sublocale $Eval-o-FuncB-o-Unpack$:

composite-simulation

⟨ $Dom\ (exp \otimes b)$ ⟩ $ExpB.resid$ ⟨ $Dom\ c$ ⟩

$FuncB-o-Unpack.map$ $Eval.map$

⟨proof⟩

lemma $EvalFuncB-o-Unpack-is-simulation$:

shows *simulation* ($Dom\ (exp \otimes b)$) ($Dom\ c$) $Eval-o-FuncB-o-Unpack.map$

⟨proof⟩

definition $eval$

where $eval \equiv mksta\ (Dom\ (exp \otimes b))\ (Dom\ c)\ Eval-o-FuncB-o-Unpack.map$

lemma $eval-simps$ [*simp*]:

shows $sta\ eval$ **and** $dom\ eval = exp \otimes b$ **and** $cod\ eval = c$

and $Dom\ eval = Dom\ (exp \otimes b)$ **and** $Cod\ eval = Dom\ c$

and $Trn\ eval = exponential-rts.MkIde\ Eval-o-FuncB-o-Unpack.map$

⟨proof⟩

lemma $eval-in-hom$ [*intro*]:

shows $\llbracket eval : exp \otimes b \rightarrow c \rrbracket$

⟨proof⟩

lemma $Map-eval$:

shows $Map\ eval = Eval.map \circ (FuncB.map \circ Unpack\ exp\ b)$

⟨proof⟩

lemma $inverse-simulations-Func-Unfunc$:

assumes $obj\ b$ **and** $obj\ c$

shows *inverse-simulations*

($exponential-rts.resid\ (Dom\ b)\ (Dom\ c)$) ($Dom\ exp$) $Func'$ $Unfunc'$

⟨proof⟩

end

Now we transfer the definitions and facts we want to $rtscatx$.

context $rtscatx$

begin

interpretation *elementary-category-with-binary-products* $hcomp$ p_0 p_1
 ⟨*proof*⟩

notation *prod* (infixr ⟨ \otimes ⟩ 51)

definition *exp*
 where $exp\ b\ c \equiv exponential-in-rtscat.exp\ b\ c$

lemma *obj-exp*:
assumes *obj b and obj c*
shows *obj (exp b c)*
 ⟨*proof*⟩

definition *eval*
 where $eval\ b\ c \equiv exponential-in-rtscat.eval\ b\ c$

lemma *eval-simps* [*simp*]:
assumes *obj b and obj c*
shows *sta (eval b c)*
and $dom\ (eval\ b\ c) = exp\ b\ c\ \otimes\ b$
and $cod\ (eval\ b\ c) = c$
 ⟨*proof*⟩

lemma *eval-in-hom_{RCC}* [*intro*]:
assumes *obj b and obj c*
shows « $eval\ b\ c : exp\ b\ c\ \otimes\ b \rightarrow c$ »
 ⟨*proof*⟩

As we did for *Pack a b* and *Unpack a b*, we now express the final definition of the “exponential constraints” *Func b c* and *Unfunc b c* in terms of the product and exponential structure.

definition *Func* :: $'A\ arr \Rightarrow 'A\ arr \Rightarrow 'A \Rightarrow ('A, 'A)\ exponential-rts.arr$
where $Func\ b\ c \equiv Currying.Curry3\ (Dom\ (exp\ b\ c))\ (Dom\ b)\ (Dom\ c)$
 $(Map\ (eval\ b\ c)\ \circ\ Pack\ (exp\ b\ c)\ b)$

definition *Unfunc* :: $'A\ arr \Rightarrow 'A\ arr \Rightarrow ('A, 'A)\ exponential-rts.arr \Rightarrow 'A$
where $Unfunc\ b\ c \equiv inverse-simulation.map$
 $(Dom\ (exp\ b\ c))\ (exponential-rts.resid\ (Dom\ b)\ (Dom\ c))$
 $(Func\ b\ c)$

lemma *Func-eq*:
assumes *obj b and obj c*
shows $simulation\ (Dom\ (exp\ b\ c))\ (exponential-rts.resid\ (Dom\ b)\ (Dom\ c))$
 $(Func\ b\ c)$
and $Func\ b\ c = exponential-in-rtscat.Func'\ b\ c$
 ⟨*proof*⟩

lemma *inverse-simulations-Func-Unfunc*:
assumes *obj b and obj c*

shows *invertible-simulation*
 $(Dom (exp\ b\ c)) (exponential\text{-}rts.resid\ (Dom\ b)\ (Dom\ c)) (Func\ b\ c)$
and *inverse-simulations*
 $(exponential\text{-}rts.resid\ (Dom\ b)\ (Dom\ c)) (Dom\ (exp\ b\ c))$
 $(Func\ b\ c) (Unfunc\ b\ c)$
 $\langle proof \rangle$

lemma *simulation-Func*:
assumes *obj b and obj c*
shows *simulation* $(Dom\ (exp\ b\ c)) (exponential\text{-}rts.resid\ (Dom\ b)\ (Dom\ c))$
 $(Func\ b\ c)$
 $\langle proof \rangle$

lemma *simulation-Unfunc*:
assumes *obj b and obj c*
shows *simulation* $(exponential\text{-}rts.resid\ (Dom\ b)\ (Dom\ c)) (Dom\ (exp\ b\ c))$
 $(Unfunc\ b\ c)$
 $\langle proof \rangle$

lemma *invertible-simulation-Func*:
assumes *obj b and obj c*
shows *invertible-simulation* $(Dom\ (exp\ b\ c)) (exponential\text{-}rts.resid\ (Dom\ b)\ (Dom\ c))$
 $(Dom\ c)$
 $(Currying.Curry3\ (Dom\ (exp\ b\ c)) (Dom\ b)\ (Dom\ c))$
 $(Map\ (eval\ b\ c) \circ Pack\ (exp\ b\ c)\ b))$
 $\langle proof \rangle$

lemma *Unfunc-eq*:
assumes *obj b and obj c*
shows $Unfunc\ b\ c = exponential\text{-}in\text{-}rtscat.Unfunc'\ b\ c$
 $\langle proof \rangle$

lemma *Func-o-Unfunc*:
assumes *obj b and obj c*
shows $Func\ b\ c \circ Unfunc\ b\ c = I\ (exponential\text{-}rts.resid\ (Dom\ b)\ (Dom\ c))$
 $\langle proof \rangle$

lemma *Unfunc-o-Func*:
assumes *obj b and obj c*
shows $Unfunc\ b\ c \circ Func\ b\ c = I\ (Dom\ (exp\ b\ c))$
 $\langle proof \rangle$

lemma *Func-Unfunc [simp]*:
assumes *obj b and obj c*
and *residuation.arr* $(exponential\text{-}rts.resid\ (Dom\ b)\ (Dom\ c))\ t$
shows $Func\ b\ c\ (Unfunc\ b\ c\ t) = t$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

lemma *Map-eval:*
assumes *obj b and obj c*
shows *Map (eval b c) =*
evaluation-map.map (Dom b) (Dom c) \circ
(product-simulation.map
(Dom (exp b c)) (Dom b) (Func b c) (I (Dom b)) \circ
Unpack (exp b c) b)
 $\langle proof \rangle$

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 $\langle Dom a \rangle$*
 $\langle proof \rangle$

sublocale *A: small-rts $\langle Dom a \rangle$*
 $\langle proof \rangle$

sublocale *B: extensional-rts $\langle Dom b \rangle$*
 $\langle proof \rangle$

sublocale *B: small-rts $\langle Dom b \rangle$*
 $\langle proof \rangle$

sublocale *AxB: product-of-extensional-rts $\langle Dom a \rangle \langle Dom b \rangle \langle proof \rangle$*

sublocale *A-Exp: exponential-rts $\langle Dom a \rangle Exp.resid \langle proof \rangle$*

sublocale *aXb: extensional-rts $\langle Dom (a \otimes b) \rangle$*
 $\langle proof \rangle$

sublocale *aXb: small-rts $\langle Dom (a \otimes b) \rangle$*
 $\langle proof \rangle$

sublocale *expXb: exponential-rts *Exp.resid* $\langle Dom b \rangle \langle proof \rangle$*

sublocale *aXb-C: exponential-rts $\langle Dom (a \otimes b) \rangle \langle Dom c \rangle \langle proof \rangle$*

sublocale *Currying $\langle Dom a \rangle \langle Dom b \rangle \langle Dom c \rangle \langle proof \rangle$*

definition *curry :: 'A arr \Rightarrow 'A arr*
where *curry f = mkarr (Dom a) Exp.resid*

$$\begin{aligned}
& (\text{Unfunc}' \circ \text{Curry3} \ (aXb\text{-}C.\text{Dom} \ (\text{Trn} \ f) \circ \text{Pack} \ a \ b)) \\
& (\text{Unfunc}' \circ \text{Curry3} \ (aXb\text{-}C.\text{Cod} \ (\text{Trn} \ f) \circ \text{Pack} \ a \ b)) \\
& (\text{Unfunc}' \circ \text{Curry} \ (aXb\text{-}C.\text{Dom} \ (\text{Trn} \ f) \circ \text{Pack} \ a \ b) \\
& \quad (aXb\text{-}C.\text{Cod} \ (\text{Trn} \ f) \circ \text{Pack} \ a \ b) \\
& \quad (aXb\text{-}C.\text{Map} \ (\text{Trn} \ f) \circ \text{Pack} \ a \ b))
\end{aligned}$$

lemma *curry-in-hom* [intro]:

assumes $\langle f : a \otimes b \rightarrow c \rangle$

shows $\langle \text{curry} \ f : a \rightarrow \text{exp} \rangle$

$\langle \text{proof} \rangle$

lemma *curry-simps* [simp]:

assumes $\langle t : a \otimes b \rightarrow c \rangle$

shows $\text{arr} \ (\text{curry} \ t)$ **and** $\text{dom} \ (\text{curry} \ t) = a$ **and** $\text{cod} \ (\text{curry} \ t) = \text{exp}$

and $\text{Dom} \ (\text{curry} \ t) = \text{Dom} \ a$ **and** $\text{Cod} \ (\text{curry} \ t) = \text{Exp.resid}$

and $\text{src} \ (\text{curry} \ t) = \text{curry} \ (\text{src} \ t)$ **and** $\text{trg} \ (\text{curry} \ t) = \text{curry} \ (\text{trg} \ t)$

and $\text{Map} \ (\text{curry} \ t) =$

$$\begin{aligned}
& (\text{Unfunc}' \circ \text{Curry} \ (aXb\text{-}C.\text{Dom} \ (\text{Trn} \ t) \circ \text{Pack} \ a \ b) \\
& \quad (aXb\text{-}C.\text{Cod} \ (\text{Trn} \ t) \circ \text{Pack} \ a \ b) \\
& \quad (aXb\text{-}C.\text{Map} \ (\text{Trn} \ t) \circ \text{Pack} \ a \ b))
\end{aligned}$$

$\langle \text{proof} \rangle$

lemma *sta-curry*:

assumes $\langle f : a \otimes b \rightarrow c \rangle$ **and** *sta* f

shows *sta* $(\text{curry} \ f)$

$\langle \text{proof} \rangle$

definition *uncurry* :: $'A \text{ arr} \Rightarrow 'A \text{ arr}$

where $\text{uncurry} \ g = \text{mkarr} \ (\text{Dom} \ (a \otimes b)) \ (\text{Dom} \ c)$

$$(\text{Uncurry} \ (\text{Func}' \circ \text{exponential-rts}.\text{Dom} \ (\text{Trn} \ g)) \circ \text{Unpack} \ a \ b)$$

$$(\text{Uncurry} \ (\text{Func}' \circ \text{exponential-rts}.\text{Cod} \ (\text{Trn} \ g)) \circ \text{Unpack} \ a \ b)$$

$$(\text{Uncurry} \ (\text{Func}' \circ \text{exponential-rts}.\text{Map} \ (\text{Trn} \ g)) \circ \text{Unpack} \ a \ b)$$

lemma *uncurry-in-hom* [intro]:

assumes $\langle g : a \rightarrow \text{exp} \rangle$

shows $\langle \text{uncurry} \ g : a \otimes b \rightarrow c \rangle$

$\langle \text{proof} \rangle$

lemma *uncurry-simps* [simp]:

assumes $\langle u : a \rightarrow \text{exp} \rangle$

shows $\text{arr} \ (\text{uncurry} \ u)$

and $\text{dom} \ (\text{uncurry} \ u) = a \otimes b$ **and** $\text{cod} \ (\text{uncurry} \ u) = c$

and $\text{Dom} \ (\text{uncurry} \ u) = \text{Dom} \ (a \otimes b)$ **and** $\text{Cod} \ (\text{uncurry} \ u) = \text{Dom} \ c$

and $\text{Map} \ (\text{uncurry} \ u) =$

$$\text{Uncurry} \ (\text{Func}' \circ \text{exponential-rts}.\text{Map} \ (\text{Trn} \ u)) \circ \text{Unpack} \ a \ b$$

and $\text{src} \ (\text{uncurry} \ u) = \text{uncurry} \ (\text{src} \ u)$

and $\text{trg} \ (\text{uncurry} \ u) = \text{uncurry} \ (\text{trg} \ u)$

$\langle \text{proof} \rangle$

lemma *sta-uncurry*:
assumes $\langle g : a \rightarrow \text{exp} \rangle$ **and** *sta g*
shows *sta (uncurry g)*
 $\langle \text{proof} \rangle$

lemma *uncurry-curry*:
assumes *obj a* **and** *obj b*
and $\langle t : a \otimes b \rightarrow c \rangle$
shows *uncurry (curry t) = t*
 $\langle \text{proof} \rangle$

lemma *curry-uncurry*:
assumes $\langle u : a \rightarrow \text{exp} \rangle$
shows *curry (uncurry u) = u*
 $\langle \text{proof} \rangle$

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 $\langle u : a \rightarrow \text{exp} \rangle$
shows *uncurry u = eval \star $\langle u \star p_1[a, b], p_0[a, b] \rangle$*
 $\langle \text{proof} \rangle$

end

Once again, we transfer the things we want to *rtscatx*.

context *rtscatx*
begin

interpretation *elementary-category-with-binary-products hcomp p₀ p₁*
 $\langle \text{proof} \rangle$

notation *hcomp* (**infixr** $\langle \star \rangle$ 53)

notation *p₀* ($\langle p_0[-, -] \rangle$)

notation *p₁* ($\langle p_1[-, -] \rangle$)

notation *tuple* ($\langle \langle -, - \rangle \rangle$)

notation *prod* (**infixr** $\langle \otimes \rangle$ 51)

definition *curry* :: $'A \text{ arr} \Rightarrow 'A \text{ arr} \Rightarrow 'A \text{ arr} \Rightarrow 'A \text{ arr} \Rightarrow 'A \text{ arr}$
where *curry* \equiv *currying-in-rtscat.curry*

definition *uncurry* :: $'A \text{ arr} \Rightarrow 'A \text{ arr} \Rightarrow 'A \text{ arr} \Rightarrow 'A \text{ arr} \Rightarrow 'A \text{ arr}$
where *uncurry* \equiv *currying-in-rtscat.uncurry*

lemma *curry-in-hom* [*intro, simp*]:
assumes *obj a* **and** *obj b*
and $\langle f : a \otimes b \rightarrow c \rangle$
shows $\langle \text{curry } a \ b \ c \ f : a \rightarrow \text{exp } b \ c \rangle$

$\langle proof \rangle$

lemma *curry-simps* [*simp*]:

assumes *obj a* **and** *obj b*

and $\langle f : a \otimes b \rightarrow c \rangle$

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*)

$\langle proof \rangle$

lemma *sta-curry*:

assumes *obj a* **and** *obj b*

and $\langle f : a \otimes b \rightarrow c \rangle$ **and** *sta f*

shows *sta* (*curry a b c f*)

$\langle proof \rangle$

lemma *uncurry-in-hom* [*intro*, *simp*]:

assumes *obj b* **and** *obj c*

and $\langle g : a \rightarrow exp\ b\ c \rangle$

shows $\langle uncurry\ a\ b\ c\ g : a \otimes b \rightarrow c \rangle$

$\langle proof \rangle$

lemma *uncurry-simps* [*simp*]:

assumes *obj b* **and** *obj c*

and $\langle g : a \rightarrow exp\ b\ c \rangle$

shows *arr* (*uncurry a b c g*)

and *dom* (*uncurry a b c g*) = *a* \otimes *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*)

$\langle proof \rangle$

lemma *sta-uncurry*:

assumes *obj b* **and** *obj c*

and $\langle g : a \rightarrow exp\ b\ c \rangle$ **and** *sta g*

shows *sta* (*uncurry a b c g*)

$\langle proof \rangle$

lemma *uncurry-curry*:

assumes *obj a* **and** *obj b*

and $\langle t : a \otimes b \rightarrow c \rangle$

shows *uncurry a b c* (*curry a b c t*) = *t*

$\langle proof \rangle$

lemma *curry-uncurry*:

assumes *obj b* **and** *obj c*

and $\langle u : a \rightarrow exp\ b\ c \rangle$

shows *curry a b c* (*uncurry a b c u*) = *u*

$\langle proof \rangle$

lemma *uncurry-expansion*:
assumes *obj b and obj c*
and $\langle u : a \rightarrow \text{exp } b \text{ } c \rangle$
shows $\text{uncurry } a \text{ } b \text{ } c \text{ } u = \text{eval } b \text{ } c \star (u \otimes b)$
 $\langle \text{proof} \rangle$

lemma *Map-curry*:
assumes *obj a and obj b and obj c*
shows $\text{Map } (\text{curry } a \text{ } b \text{ } c \text{ } f) =$
 $\text{Unfunc } b \text{ } c \circ$
 $\text{Currying.Curry } (Dom \ a) \ (Dom \ b) \ (Dom \ c)$
 $(Src \ f \circ Pack \ a \ b) \ (Trg \ f \circ Pack \ a \ b) \ (\text{Map } f \circ Pack \ a \ b)$
 $\langle \text{proof} \rangle$

lemma *Map-uncurry*:
assumes *obj a and obj b and obj c*
shows $\text{Map } (\text{uncurry } a \text{ } b \text{ } c \text{ } g) =$
 $\text{Currying.Uncurry } (Dom \ a) \ (Dom \ b) \ (Dom \ c)$
 $(Func \ b \ c \circ \text{exponential-rts.Map } (Trn \ g)) \circ \text{Unpack } a \ b$
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

notation *prod* (infixr $\langle \otimes \rangle$ 51)

interpretation *elementary-cartesian-closed-category*
 $\text{hcomp } p_0 \ p_1 \ \text{one trm exp eval curry}$
 $\langle \text{proof} \rangle$

lemma *is-elementary-cartesian-closed-category*:
shows *elementary-cartesian-closed-category*
 $\text{hcomp } p_0 \ p_1 \ \text{one trm exp eval curry}$
 $\langle \text{proof} \rangle$

theorem *is-cartesian-closed-category*:
shows *cartesian-closed-category hcomp*
 $\langle \text{proof} \rangle$

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 $'A \text{ arr}$. So one thing that needs to be done in order to carry out this program is to express the “object repleteness” of \mathbf{RTS}^\dagger in terms of small extensional RTS's with arrow type $'A \text{ arr}$, as opposed to small extensional RTS's with arrow type $'A$. However, the type $'A \text{ arr}$ 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 $'A \text{ arr}$ 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 $'A \text{ arr}$, 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 $'A \text{ arr}$ to $'A$.

```
fun inj-arr :: 'A arr  $\Rightarrow$  '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 $\text{Collect } (\text{residuation.}arr A) \cup$
 $\{ResiduatedTransitionSystem.partial\text{-magma.null } A\} \subseteq$
 $\text{Collect } arr \cup \{Null\}$
shows $inj\text{-on } inj\text{-arr}$
 $(\text{Collect } (\text{residuation.}arr A) \cup$
 $\{ResiduatedTransitionSystem.partial\text{-magma.null } A\})$
 $\langle proof \rangle$

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 $HOM \mathbf{1} a$ 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* $\Rightarrow 'A$, then applying *bij-mkobj* to obtain the corresponding object a , and finally using the isomorphism $Dom a \cong HOM \mathbf{1} a$ to conclude that $HOM \mathbf{1} a \cong A$.

lemma *obj-replete*:
fixes $A :: 'A$ *arr resid*
assumes $small\text{-rts } A \wedge extensional\text{-rts } A$
and $\text{Collect } (\text{residuation.}arr A) \cup$
 $\{ResiduatedTransitionSystem.partial\text{-magma.null } A\}$
 $\subseteq \text{Collect } arr \cup \{null\}$
shows $\exists a. obj a \wedge isomorphic\text{-rts } A (HOM \mathbf{1} a)$
 $\langle proof \rangle$

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 $HOM \mathbf{1} a$ to $HOM \mathbf{1} b$ and the arrows from a to b correspond bijectively to the transformations between such simulations.

lemma *HOM1-faithful-for-sta*:
assumes $\langle f : a \rightarrow_{sta} b \rangle$ **and** $\langle g : a \rightarrow_{sta} b \rangle$
and $cov\text{-HOM } \mathbf{1} f = cov\text{-HOM } \mathbf{1} g$
shows $f = g$
 $\langle proof \rangle$

lemma *HOM1-faithful-for-arr*:
assumes $arr t$ **and** $arr u$ **and** $src t = src u$ **and** $trg t = trg u$
and $cov\text{-HOM } \mathbf{1} t = cov\text{-HOM } \mathbf{1} u$
shows $t = u$
 $\langle proof \rangle$

lemma *HOM1-full-for-sta*:
assumes $obj a$ **and** $obj b$ **and** $simulation (HOM \mathbf{1} a) (HOM \mathbf{1} b) F$
shows $\exists f. \langle f : a \rightarrow b \rangle \wedge sta f \wedge cov\text{-HOM } \mathbf{1} f = F$
 $\langle proof \rangle$

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 $\exists t. \text{arr } t \wedge \text{src } t = f \wedge \text{trg } t = g \wedge \text{cov-HOM } 1 \ t = T$
 $\langle \text{proof} \rangle$

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)))
 $\langle \text{proof} \rangle$

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)))
 $\langle \text{proof} \rangle$

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 \mathbf{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 \mathbf{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* $\langle \text{proof} \rangle$

interpretation *RTSx*: *rtscatx arr-type* $\langle \text{proof} \rangle$

interpretation *RTSx*: *elementary-category-with-binary-products*
RTSx.hcomp RTSx.p₀ RTSx.p₁

$\langle \text{proof} \rangle$

interpretation *RTS_S*: *subcategory RTSx.hcomp RTSx.sta*

$\langle \text{proof} \rangle$

type-synonym *'a arr* = *'a rtscatx.arr*

definition *comp* :: *'A arr comp* (**infixr** $\langle \cdot \rangle$ 53)

where *comp* \equiv *subcategory.comp RTSx.hcomp RTSx.sta*

sublocale *category comp*

$\langle \text{proof} \rangle$

notation *in-hom* ($\langle \ll - : - \rightarrow - \gg \rangle$)

lemma *ide-iff-RTS-obj*:

shows *ide a* \longleftrightarrow *RTSx.obj a*

$\langle \text{proof} \rangle$

lemma *arr-iff-RTS-sta*:

shows *arr f* \longleftrightarrow *RTSx.sta f*

$\langle \text{proof} \rangle$

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 $Map :: 'A \text{ arr} \Rightarrow 'A \Rightarrow 'A$
where $Map \equiv RTSx.Map$

lemma $ideD_{RTSC}$ [*intro, simp*]:
assumes $ide\ a$
shows $Obj\ (Rts\ a)$
<proof>

lemma $ide\mkide$ [*intro, simp*]:
assumes $Obj\ A$
shows $ide\ (mkide\ A)$
<proof>

lemma $Rts\mkide$ [*simp*]:
shows $Rts\ (mkide\ A) = A$
<proof>

lemma $mkide\ Rts$ [*simp*]:
assumes $ide\ a$
shows $mkide\ (Rts\ a) = a$
<proof>

lemma $Dom\mkide$ [*simp*]:
assumes $ide\ (mkide\ A)$
shows $Dom\ (mkide\ A) = A$
<proof>

lemma $Cod\mkide$ [*simp*]:
assumes $ide\ (mkide\ A)$
shows $Cod\ (mkide\ A) = A$
<proof>

lemma $Map\mkide$ [*simp*]:
assumes $ide\ (mkide\ A)$
shows $Map\ (mkide\ A) = I\ A$
<proof>

lemma $bij\mkide$:
shows $mkide \in Collect\ Obj \rightarrow Collect\ ide$
and $Rts \in Collect\ ide \rightarrow Collect\ Obj$
and $Rts\ (mkide\ A) = A$
and $ide\ a \implies mkide\ (Rts\ a) = a$
and $bij\text{-betw}\ mkide\ (Collect\ Obj)\ (Collect\ ide)$
and $bij\text{-betw}\ Rts\ (Collect\ ide)\ (Collect\ Obj)$
<proof>

lemma $arrD$:
assumes $arr\ f$

shows $Obj (Rts (dom f))$ **and** $Obj (Rts (cod f))$
and $simulation (Rts (dom f)) (Rts (cod f)) (Map f)$
 $\langle proof \rangle$

lemma *arr-mkarr* [*intro, simp*]:
assumes $Obj A$ **and** $Obj B$ **and** $simulation A B F$
shows $arr (mkarr A B F)$
 $\langle proof \rangle$

lemma *arrI_{RTSC}*:
assumes $f \in mkarr A B \text{ ' } Collect (simulation A B)$
and $Obj A$ **and** $Obj B$
shows $arr f$
 $\langle proof \rangle$

lemma *Dom-mkarr* [*simp*]:
assumes $arr (mkarr A B F)$
shows $Dom (mkarr A B F) = A$
 $\langle proof \rangle$

lemma *Cod-mkarr* [*simp*]:
assumes $arr (mkarr A B F)$
shows $Cod (mkarr A B F) = B$
 $\langle proof \rangle$

lemma *Map-mkarr* [*simp*]:
assumes $arr (mkarr A B F)$
shows $Map (mkarr A B F) = F$
 $\langle proof \rangle$

lemma *mkarr-Map* [*simp*]:
assumes $Obj A$ **and** $Obj B$ **and** $t \in \{t. \langle t : mkide A \rightarrow mkide B \rangle\}$
shows $mkarr A B (Map t) = t$
 $\langle proof \rangle$

lemma *dom-mkarr* [*simp*]:
assumes $arr (mkarr A B F)$
shows $dom (mkarr A B F) = mkide A$
 $\langle proof \rangle$

lemma *cod-mkarr* [*simp*]:
assumes $arr (mkarr A B F)$
shows $cod (mkarr A B F) = mkide B$
 $\langle proof \rangle$

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 $\langle mkarr A B F : a \rightarrow b \rangle$

$\langle proof \rangle$

lemma *bij-mkarr*:

assumes *Obj A and Obj B*

shows $mkarr\ A\ B \in Collect\ (simulation\ A\ B)$

$\rightarrow \{t. \llbracket t : mkide\ A \rightarrow mkide\ B \rrbracket\}$

and $Map \in \{t. \llbracket t : mkide\ A \rightarrow mkide\ B \rrbracket\} \rightarrow Collect\ (simulation\ A\ B)$

and $Map\ (mkarr\ A\ B\ F) = F$

and $t \in \{t. \llbracket t : mkide\ A \rightarrow mkide\ B \rrbracket\} \implies mkarr\ A\ B\ (Map\ t) = t$

and $bij\text{-}betw\ (mkarr\ A\ B)\ (Collect\ (simulation\ A\ B))$

$\{t. \llbracket t : mkide\ A \rightarrow mkide\ B \rrbracket\}$

and $bij\text{-}betw\ Map\ \{t. \llbracket t : mkide\ A \rightarrow mkide\ B \rrbracket\}$

$(Collect\ (simulation\ A\ B))$

$\langle proof \rangle$

lemma *arr-eqI*:

assumes *par f g and Map f = Map g*

shows $f = g$

$\langle proof \rangle$

lemma *Map-ide*:

assumes *ide a*

shows $Map\ a = I\ (Rts\ a)$

$\langle proof \rangle$

lemma *Map-comp*:

assumes *seq g f*

shows $Map\ (g \cdot f) = Map\ g \circ Map\ f$

$\langle proof \rangle$

lemma *comp-mkarr*:

assumes *arr (mkarr A B F) and arr (mkarr B C G)*

shows $mkarr\ B\ C\ G \cdot mkarr\ A\ B\ F = mkarr\ A\ C\ (G \circ F)$

$\langle proof \rangle$

lemma *iso-char*:

shows $iso\ t \iff arr\ t \wedge invertible\text{-}simulation\ (Dom\ t)\ (Cod\ t)\ (Map\ t)$

$\langle proof \rangle$

lemma *isomorphic-char*:

shows $isomorphic\ a\ b \iff$

$ide\ a \wedge ide\ b \wedge (\exists F. invertible\text{-}simulation\ (Rts\ a)\ (Rts\ b)\ F)$

$\langle proof \rangle$

4.4.1 Terminal Object

definition *one* $\quad (\langle 1 \rangle)$

where $one \equiv RTSx.one$

no-notation $RTSx.one \quad (\langle 1 \rangle)$

definition *trm*
where $trm \equiv RTSx.trm$

lemma *Rts-one* [*simp*]:
shows $Rts \mathbf{1} = One.resid$
 $\langle proof \rangle$

lemma *mkide-One* [*simp*]:
shows $mkide One.resid = \mathbf{1}$
 $\langle proof \rangle$

sublocale *elementary-category-with-terminal-object comp one trm*
 $\langle proof \rangle$

lemma *Map-trm*:
assumes *ide a*
shows $Map (trm a) = constant-simulation.map (Rts a) One.resid One.the-arr$
 $\langle proof \rangle$

lemma *terminal-char*:
shows $terminal\ x \longleftrightarrow ide\ x \wedge (\exists!t. residuation.arr (Rts\ x)\ t)$
 $\langle proof \rangle$

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 $RTSx.p_0$ ($\langle p_0[-, -] \rangle$)

no-notation $RTSx.p_1$ ($\langle p_1[-, -] \rangle$)

notation p_0 ($\langle p_0[-, -] \rangle$)

notation p_1 ($\langle p_1[-, -] \rangle$)

sublocale *elementary-cartesian-category comp p_0 p_1 one trm*
 $\langle proof \rangle$

notation *prod* (**infixr** $\langle \otimes \rangle$ 51)

notation *tuple* ($\langle \langle -, - \rangle \rangle$)

notation *dup* ($\langle d[-] \rangle$)

notation *assoc* ($\langle a[-, -, -] \rangle$)

notation *assoc'* ($\langle a^{-1}[-, -, -] \rangle$)

notation *lunit* ($\langle l[-] \rangle$)

notation *lunit'* ($\langle l^{-1}[-] \rangle$)

notation *runit* ($\langle r[-] \rangle$)

notation *runit'* ($\langle r^{-1}[-] \rangle$)

lemma *tuple-char*:
shows $\text{tuple} = (\lambda f g. \text{if span } f g \text{ then } RTSx.\text{tuple } f g \text{ else null})$
 $\langle \text{proof} \rangle$

lemma *prod-char*:
shows $(\otimes) = (\lambda f g. \text{if arr } f \wedge \text{arr } g \text{ then } RTSx.\text{prod } f g \text{ else null})$
 $\langle \text{proof} \rangle$

definition $\text{Pack} :: 'A \text{ arr} \Rightarrow 'A \text{ arr} \Rightarrow 'A \times 'A \Rightarrow 'A$
where $\text{Pack} \equiv RTSx.\text{Pack}$

definition $\text{Unpack} :: 'A \text{ arr} \Rightarrow 'A \text{ arr} \Rightarrow 'A \Rightarrow 'A \times 'A$
where $\text{Unpack} \equiv RTSx.\text{Unpack}$

lemma *inverse-simulations-Pack-Unpack*:
assumes *ide a and ide b*
shows $\text{inverse-simulations } (Rts (a \otimes b)) (Rts a \otimes Rts b)$
 $(\text{Pack } a b) (\text{Unpack } a b)$
 $\langle \text{proof} \rangle$

lemma *simulation-Pack*:
assumes *ide a and ide b*
shows $\text{simulation } (Rts a \otimes Rts b) (Rts (a \otimes b)) (\text{Pack } a b)$
 $\langle \text{proof} \rangle$

lemma *simulation-Unpack*:
assumes *ide a and ide b*
shows $\text{simulation } (Rts (a \otimes b)) (Rts a \otimes Rts b) (\text{Unpack } a b)$
 $\langle \text{proof} \rangle$

lemma *Pack-o-Unpack*:
assumes *ide a and ide b*
shows $\text{Pack } a b \circ \text{Unpack } a b = I (Rts (a \otimes b))$
 $\langle \text{proof} \rangle$

lemma *Unpack-o-Pack*:
assumes *ide a and ide b*
shows $\text{Unpack } a b \circ \text{Pack } a b = I (Rts a \otimes Rts b)$
 $\langle \text{proof} \rangle$

lemma *Pack-Unpack [simp]*:
assumes *ide a and ide b*
and *residuation.arr (Rts (a ⊗ b)) t*
shows $\text{Pack } a b (\text{Unpack } a b t) = t$
 $\langle \text{proof} \rangle$

lemma *Unpack-Pack [simp]*:
assumes *ide a and ide b*

and *residuation.arr* $(Rts\ a \otimes Rts\ b)\ t$
shows $Unpack\ a\ b\ (Pack\ a\ b\ t) = t$
 $\langle proof \rangle$

lemma *Map-p₀*:
assumes *ide a* **and** *ide b*
shows $Map\ p_0[a, b] = product\ rts.P_0\ (Rts\ a)\ (Rts\ b) \circ Unpack\ a\ b$
 $\langle proof \rangle$

lemma *Map-p₁*:
assumes *ide a* **and** *ide b*
shows $Map\ p_1[a, b] = product\ rts.P_1\ (Rts\ a)\ (Rts\ b) \circ Unpack\ a\ b$
 $\langle proof \rangle$

lemma *Map-tuple*:
assumes $\langle f : x \rightarrow a \rangle$ **and** $\langle g : x \rightarrow b \rangle$
shows $Map\ \langle f, g \rangle = Pack\ a\ b \circ \langle \langle Map\ f, Map\ g \rangle \rangle$
 $\langle proof \rangle$

corollary *Map-dup*:
assumes *ide a*
shows $Map\ d[a] = Pack\ a\ a \circ \langle \langle Map\ a, Map\ a \rangle \rangle$
 $\langle proof \rangle$

proposition *Map-lunit*:
assumes *ide a*
shows $Map\ l[a] = product\ rts.P_0\ (Rts\ \mathbf{1})\ (Rts\ a) \circ Unpack\ \mathbf{1}\ a$
 $\langle proof \rangle$

proposition *Map-runit*:
assumes *ide a*
shows $Map\ r[a] = product\ rts.P_1\ (Rts\ a)\ (Rts\ \mathbf{1}) \circ Unpack\ a\ \mathbf{1}$
 $\langle proof \rangle$

lemma *assoc-expansion*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows $a[a, b, c] =$
 $\langle p_1[a, b] \cdot p_1[a \otimes b, c], \langle p_0[a, b] \cdot p_1[a \otimes b, c], p_0[a \otimes b, c] \rangle \rangle$
 $\langle proof \rangle$

lemma *Unpack-naturality*:
assumes *arr f* **and** *arr g*
shows $Unpack\ (cod\ f)\ (cod\ g) \circ Map\ (f \otimes g) =$
 $product\ simulation.map\ (Rts\ (dom\ f))\ (Rts\ (dom\ g))\ (Map\ f)\ (Map\ g) \circ$
 $Unpack\ (dom\ f)\ (dom\ g)$
 $\langle proof \rangle$

lemma *Map-prod*:
assumes *arr f* **and** *arr g*

shows $Map (f \otimes g) =$
 $Pack (cod f) (cod g) \circ$
 $product-simulation.map (Rts (dom f)) (Rts (dom g)) (Map f) (Map g) \circ$
 $Unpack (dom f) (dom g)$
 $\langle proof \rangle$

4.4.3 Exponentials

definition $exp :: 'A arr \Rightarrow 'A arr \Rightarrow 'A arr$
where $exp \equiv RTSx.exp$

definition $eval :: 'A arr \Rightarrow 'A arr \Rightarrow 'A arr$
where $eval \equiv RTSx.eval$

definition $curry :: 'A arr \Rightarrow 'A arr \Rightarrow 'A arr \Rightarrow 'A arr \Rightarrow 'A arr$
where $curry \equiv RTSx.curry$

definition $uncurry :: 'A arr \Rightarrow 'A arr \Rightarrow 'A arr \Rightarrow 'A arr \Rightarrow 'A arr$
where $uncurry \equiv RTSx.uncurry$

definition $Func :: 'A arr \Rightarrow 'A arr \Rightarrow 'A \Rightarrow ('A, 'A) exponential-rts.arr$
where $Func \equiv RTSx.Func$

definition $Unfunc :: 'A arr \Rightarrow 'A arr \Rightarrow ('A, 'A) exponential-rts.arr \Rightarrow 'A$
where $Unfunc \equiv RTSx.Unfunc$

lemma *inverse-simulations-Func-Unfunc*:
assumes *ide b and ide c*
shows *inverse-simulations*
 $(exponential-rts.resid (Rts b) (Rts c)) (Rts (exp b c))$
 $(Func b c) (Unfunc b c)$
 $\langle proof \rangle$

lemma *simulation-Func*:
assumes *ide b and ide c*
shows *simulation*
 $(Rts (exp b c)) (exponential-rts.resid (Rts b) (Rts c)) (Func b c)$
 $\langle proof \rangle$

lemma *simulation-Unfunc*:
assumes *ide b and ide c*
shows *simulation*
 $(exponential-rts.resid (Rts b) (Rts c)) (Rts (exp b c)) (Unfunc b c)$
 $\langle proof \rangle$

lemma *Func-o-Unfunc*:
assumes *ide b and ide c*
shows $Func b c \circ Unfunc b c = I (exponential-rts.resid (Rts b) (Rts c))$
 $\langle proof \rangle$

lemma *Unfunc-o-Func*:
assumes *ide b* **and** *ide c*
shows $\text{Unfunc } b \ c \circ \text{Func } b \ c = I \ (Rts \ (exp \ b \ c))$
 $\langle proof \rangle$

lemma *Func-Unfunc [simp]*:
assumes *ide b* **and** *ide c*
and *residuation.arr (exponential-rts.resid (Rts b) (Rts c)) t*
shows $\text{Func } b \ c \ (\text{Unfunc } b \ c \ t) = t$
 $\langle proof \rangle$

lemma *Unfunc-Func [simp]*:
assumes *ide b* **and** *ide c*
and *residuation.arr (Rts (exp b c)) t*
shows $\text{Unfunc } b \ c \ (\text{Func } b \ c \ t) = t$
 $\langle proof \rangle$

lemma *Map-eval*:
assumes *ide b* **and** *ide c*
shows $\text{Map} \ (eval \ b \ c) =$
 $evaluation\text{-map.map} \ (Rts \ b) \ (Rts \ c) \circ$
 $product\text{-simulation.map} \ (Rts \ (exp \ b \ c)) \ (Rts \ b) \ (\text{Func } b \ c) \ (I \ (Rts \ b)) \circ$
 $Unpack \ (exp \ b \ c) \ b$
 $\langle proof \rangle$

lemma *Map-curry*:
assumes *ide a* **and** *ide b* **and** $\langle f : a \otimes b \rightarrow c \rangle$
shows $\text{Map} \ (curry \ a \ b \ c \ f) =$
 $\text{Unfunc } b \ c \circ$
 $Currying.Curry3 \ (Rts \ a) \ (Rts \ b) \ (Rts \ c) \ (\text{Map } f \circ \text{Pack } a \ b)$
 $\langle proof \rangle$

lemma *Map-uncurry*:
assumes *ide b* **and** *ide c* **and** $\langle g : a \rightarrow exp \ b \ c \rangle$
shows $\text{Map} \ (uncurry \ a \ b \ c \ g) =$
 $Currying.Uncurry \ (Rts \ a) \ (Rts \ b) \ (Rts \ c) \ (\text{Func } b \ c \circ \text{Map } g) \circ$
 $Unpack \ a \ b$
 $\langle proof \rangle$

4.4.4 Cartesian Closure

sublocale *elementary-cartesian-closed-category*
 $comp \ p_0 \ p_1 \ one \ trm \ exp \ eval \ curry$
 $\langle proof \rangle$

theorem *is-elementary-cartesian-closed-category*:
shows *elementary-cartesian-closed-category*
 $comp \ p_0 \ p_1 \ one \ trm \ exp \ eval \ curry$

<proof>

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* $\langle Rts\ a \rangle$
<proof>

interpretation *B: extensional-rts* $\langle Rts\ b \rangle$
<proof>

interpretation *C: extensional-rts* $\langle Rts\ c \rangle$
<proof>

interpretation *A: identity-simulation* $\langle Rts\ a \rangle$ *<proof>*

interpretation *B: identity-simulation* $\langle Rts\ b \rangle$ *<proof>*

interpretation *C: identity-simulation* $\langle Rts\ c \rangle$ *<proof>*

interpretation *AxB: product-rts* $\langle Rts\ a \rangle$ $\langle Rts\ b \rangle$ *<proof>*

interpretation *AxB-xC: product-rts* *AxB.resid* $\langle Rts\ c \rangle$ *<proof>*

interpretation *BxC: product-rts* $\langle Rts\ b \rangle$ $\langle Rts\ c \rangle$ *<proof>*

interpretation *Ax-BxC: product-rts* $\langle Rts\ a \rangle$ *BxC.resid* *<proof>*

interpretation *AxB: extensional-rts* *AxB.resid*
<proof>

interpretation *BxC: extensional-rts* *BxC.resid*
<proof>

interpretation *AxB-xC: extensional-rts* *AxB-xC.resid*
<proof>

interpretation *Ax-BxC: extensional-rts* *Ax-BxC.resid*
<proof>

sublocale *ASSOC: ASSOC* $\langle Rts\ a \rangle$ $\langle Rts\ b \rangle$ $\langle Rts\ c \rangle$ *<proof>*

interpretation *axb: extensional-rts* $\langle Rts\ (a \otimes b) \rangle$
<proof>

interpretation *bxc: extensional-rts* $\langle Rts\ (b \otimes c) \rangle$

$\langle \text{proof} \rangle$
interpretation $axb-xc$: *extensional-rts* $\langle \text{Rts } ((a \otimes b) \otimes c) \rangle$
 $\langle \text{proof} \rangle$
interpretation $ax-bxc$: *extensional-rts* $\langle \text{Rts } (a \otimes (b \otimes c)) \rangle$
 $\langle \text{proof} \rangle$

interpretation $PU-ab$: *inverse-simulations*
 $\langle \text{Rts } (a \otimes b) \rangle \text{ AxB.resid } \langle \text{Pack } a \ b \rangle \langle \text{Unpack } a \ b \rangle$
 $\langle \text{proof} \rangle$

interpretation $PU-bc$: *inverse-simulations*
 $\langle \text{Rts } (b \otimes c) \rangle \text{ BxC.resid } \langle \text{Pack } b \ c \rangle \langle \text{Unpack } b \ c \rangle$
 $\langle \text{proof} \rangle$

interpretation $Axbc$: *product-rts* $\langle \text{Rts } a \rangle \langle \text{Rts } (b \otimes c) \rangle \langle \text{proof} \rangle$
interpretation $Axbc$: *identity-simulation* $\text{Axbc.resid } \langle \text{proof} \rangle$
interpretation $abxC$: *product-rts* $\langle \text{Rts } (a \otimes b) \rangle \langle \text{Rts } c \rangle \langle \text{proof} \rangle$
interpretation $abxC$: *identity-simulation* $\text{abxC.resid } \langle \text{proof} \rangle$

interpretation $AxPack-bc$:
product-simulation $\langle \text{Rts } a \rangle \text{ BxC.resid } \langle \text{Rts } a \rangle \langle \text{Rts } (b \otimes c) \rangle$
 $\langle I (\text{Rts } a) \rangle \langle \text{Pack } b \ c \rangle \langle \text{proof} \rangle$

interpretation $AxUnpack-bc$:
product-simulation $\langle \text{Rts } a \rangle \langle \text{Rts } (b \otimes c) \rangle \langle \text{Rts } a \rangle \text{ BxC.resid}$
 $\langle I (\text{Rts } a) \rangle \langle \text{Unpack } b \ c \rangle \langle \text{proof} \rangle$

interpretation $Unpack-abxC$:
product-simulation $\langle \text{Rts } (a \otimes b) \rangle \langle \text{Rts } c \rangle \text{ AxB.resid } \langle \text{Rts } c \rangle$
 $\langle \text{Unpack } a \ b \rangle \langle I (\text{Rts } c) \rangle \langle \text{proof} \rangle$

sublocale $PU-Axbc$: *inverse-simulations* $\text{Axbc.resid } \text{Ax-BxC.resid}$
 $\text{AxPack-bc.map } \text{AxUnpack-bc.map}$
 $\langle \text{proof} \rangle$

The following shows that the simulation $\text{Map } a[a, b, c]$ 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 $(\text{AxUnpack-bc.map} \circ \text{Unpack } a \ (b \otimes c)) \circ \text{Map } a[a, b, c] =$
 $\text{ASSOC.map} \circ (\text{Unpack-abxC.map} \circ \text{Unpack } (a \otimes b) \ c)$
 $\langle \text{proof} \rangle$

As a corollary, we obtain an explicit formula for $\text{Map } a[a, b, c]$ in terms of the external associator ASSOC.map and packing and unpacking isomorphisms.

corollary *Map-assoc*:
shows $\text{Map } a[a, b, c] =$
 $(\text{Pack } a \ (b \otimes c) \circ \text{AxPack-bc.map}) \circ$
 $\text{ASSOC.map} \circ$
 $(\text{Unpack-abxC.map} \circ \text{Unpack } (a \otimes b) \ c)$

⟨proof⟩

end

context *rtscat*

begin

proposition *Map-assoc*:

assumes *ide a and ide b and ide c*

shows $\text{Map } a[a, b, c] =$

$\text{Pack } a (b \otimes c) \circ$

$\text{product-simulation.map } (Rts \ a) (Rts \ b \otimes Rts \ c)$

$(I (Rts \ a)) (Pack \ b \ c) \circ$

$\text{ASSOC.map } (Rts \ a) (Rts \ b) (Rts \ c) \circ$

$(\text{product-simulation.map}$

$(Rts \ (a \otimes b)) (Rts \ c) (\text{Unpack } a \ b) (I (Rts \ c)) \circ$

$\text{Unpack } (a \otimes b) \ c)$

⟨proof⟩

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 $\alpha \iota$*

⟨proof⟩

sublocale *ECMC: elementary-closed-monoidal-category comp Prod $\alpha \iota$
exp eval curry*

⟨proof⟩

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* $\langle Rts a \rangle$
 $\langle proof \rangle$

interpretation B : *extensional-rts* $\langle Rts b \rangle$
 $\langle proof \rangle$

interpretation C : *extensional-rts* $\langle Rts c \rangle$
 $\langle proof \rangle$

interpretation AxB : *product-rts* $\langle Rts a \rangle \langle Rts b \rangle \langle proof \rangle$

interpretation BxC : *product-rts* $\langle Rts b \rangle \langle Rts c \rangle \langle proof \rangle$

interpretation AB : *exponential-rts* $\langle Rts a \rangle \langle Rts b \rangle \langle proof \rangle$

interpretation BC : *exponential-rts* $\langle Rts b \rangle \langle Rts c \rangle \langle proof \rangle$

interpretation AC : *exponential-rts* $\langle Rts a \rangle \langle Rts c \rangle \langle proof \rangle$

interpretation $ABxA$: *product-rts* $AB.resid \langle Rts a \rangle \langle proof \rangle$

interpretation $BCxAB$: *product-rts* $BC.resid AB.resid \langle proof \rangle$

interpretation $BCxAB$: *extensional-rts* $BCxAB.resid$
 $\langle proof \rangle$

interpretation $BCxAB-x-A$: *product-rts* $BCxAB.resid \langle Rts a \rangle \langle proof \rangle$

interpretation ab : *extensional-rts* $\langle EXP a b \rangle$
 $\langle proof \rangle$

interpretation bc : *extensional-rts* $\langle EXP b c \rangle$
 $\langle proof \rangle$

interpretation $bcxab$: *product-of-extensional-rts* $\langle EXP b c \rangle \langle EXP a b \rangle \langle proof \rangle$

interpretation $abxA$: *product-rts* $\langle EXP a b \rangle \langle Rts a \rangle \langle proof \rangle$

interpretation $bcxB$: *product-rts* $\langle EXP b c \rangle \langle Rts b \rangle \langle proof \rangle$

interpretation $bc-x-abxA$: *product-rts* $\langle EXP b c \rangle abxA.resid \langle proof \rangle$

interpretation $bcxab-x-A$: *product-rts* $bcxab.resid \langle Rts a \rangle \langle proof \rangle$

interpretation $ASSOC$: $ASSOC BC.resid AB.resid \langle Rts a \rangle \langle proof \rangle$

interpretation $COMP$: $COMP \langle Rts a \rangle \langle Rts b \rangle \langle Rts c \rangle \langle proof \rangle$

interpretation $Assoc-abc$: *Association arr-type* $a b c$
 $\langle proof \rangle$

interpretation $Assoc-bc-ab-a$: *Association arr-type* $\langle exp b c \rangle \langle exp a b \rangle a$
 $\langle proof \rangle$

interpretation $Func-Unfunc-ab$: *inverse-simulations* $AB.resid \langle EXP a b \rangle$
 $\langle Func a b \rangle \langle Unfunc a b \rangle$
 $\langle proof \rangle$

interpretation $Func-Unfunc-bc$: *inverse-simulations* $BC.resid \langle EXP b c \rangle$
 $\langle Func b c \rangle \langle Unfunc b c \rangle$
 $\langle proof \rangle$

interpretation $Func-bcxFunc-ab$: *product-simulation*
 $\langle EXP b c \rangle \langle EXP a b \rangle BC.resid AB.resid$
 $\langle Func b c \rangle \langle Func a b \rangle \langle proof \rangle$

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 $Func\ a\ c \circ Map\ (ECMC.Comp\ a\ b\ c) =$
 $COMP.map \circ (Func\text{-}bcxFunc\text{-}ab.map \circ Unpack\ (exp\ b\ c)\ (exp\ a\ b))$
 $\langle proof \rangle$

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 $Map\ (ECMC.Comp\ a\ b\ c) =$
 $Unfunc\ a\ c \circ COMP.map \circ$
 $(Func\text{-}bcxFunc\text{-}ab.map \circ Unpack\ (exp\ b\ c)\ (exp\ a\ b))$
 $\langle proof \rangle$

end

context *rtscat*

begin

abbreviation *EXP*

where $EXP \equiv \lambda a\ b. Rts\ (exp\ a\ b)$

proposition *Map-Comp*:

assumes *ide a* **and** *ide b* **and** *ide c*
shows $Map\ (ECMC.Comp\ a\ b\ c) =$
 $Unfunc\ a\ c \circ COMP.map\ (Rts\ a)\ (Rts\ b)\ (Rts\ c) \circ$
 $(product\text{-}simulation.map\ (EXP\ b\ c)\ (EXP\ a\ b)$
 $(Func\ b\ c)\ (Func\ a\ b) \circ$
 $Unpack\ (exp\ b\ c)\ (exp\ a\ b))$
 $\langle proof \rangle$

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* $\langle TYPE(V) \rangle$
\langle proof \rangle

interpretation *RTS: rtscat* $\langle TYPE(V) \rangle$
\langle proof \rangle

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  $\alpha$   $\iota$ 
    <proof>
```

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 $CMC.tensor\ f\ g = f \otimes g$
 $\langle proof \rangle$

The situation with tupling and “duplicators” is similar.

lemma *tuple-agreement*:
assumes *span f g*
shows $CMC.tuple\ f\ g = \langle f, g \rangle$
 $\langle proof \rangle$

lemma *dup-agreement*:
assumes *arr f*
shows $CMC.dup\ f = dup\ f$
 $\langle proof \rangle$

sublocale *elementary-cartesian-closed-monoidal-category*
comp Prod α ι exp eval curry
 $\langle proof \rangle$

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 $CMC.assoc\ a\ b\ c =$
 $\langle p_1[a, b] \cdot p_1[a \otimes b, c], \langle p_0[a, b] \cdot p_1[a \otimes b, c], p_0[a \otimes b, c] \rangle \rangle$
 $\langle proof \rangle$

lemma *extends-to-enriched-category*:
shows *enriched-category comp Prod α ι*
(Collect ide) exp ECMC.Id ECMC.Comp
 $\langle proof \rangle$

end

locale *rts-enriched-category* =
universe arr-type +
RTS: rtsat 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 ⇒ 'O ⇒ 'A rtscatx.arr
and Id :: 'O ⇒ 'A rtscatx.arr
and Comp :: 'O ⇒ 'O ⇒ 'O ⇒ '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  $\langle HOM_{EC} a b \rangle$ 
     $\langle proof \rangle$ 

    sublocale small-rts  $\langle HOM_{EC} a b \rangle$ 
     $\langle proof \rangle$ 

  end

locale rts-enriched-functor =
  RTS: rtscat arr-type +
  A: rts-enriched-category arr-type ObjA HomA IdA CompA +
  B: rts-enriched-category arr-type ObjB HomB IdB CompB +
  enriched-functor RTS.comp RTS.Prod RTS.α RTS.ι
  for arr-type :: 'A itself
  begin

    lemma is-local-simulation:
    assumes a ∈  $Obj_A$  and b ∈  $Obj_A$ 
    shows simulation ( $A.HOM_{EC} a b$ ) ( $B.HOM_{EC} (F_o a) (F_o b)$ )
    ( $RTS.Map (F_a a b)$ )
     $\langle proof \rangle$ 

  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     (infix ⟨⊗⟩ 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 (HOMEC (Dom t) (Cod t)) (Trn t)

```

abbreviation $Ide :: ('O, 'A) arr \Rightarrow bool$
where $Ide \equiv \lambda t. t \neq Null \wedge Dom\ t \in Obj \wedge Cod\ t \in Obj \wedge$
 $residuation.ide\ (HOM_{EC}\ (Dom\ t)\ (Cod\ t))\ (Trn\ t)$

definition $Con :: ('O, 'A) arr \Rightarrow ('O, 'A) arr \Rightarrow bool$
where $Con\ t\ u \equiv Arr\ t \wedge Arr\ u \wedge Dom\ t = Dom\ u \wedge Cod\ t = Cod\ u \wedge$
 $residuation.con\ (HOM_{EC}\ (Dom\ t)\ (Cod\ t))\ (Trn\ t)\ (Trn\ u)$

The global residuation is obtained by combining the local residuations of each of the hom-RTS's.

fun $resid$ (**infix** $\langle \rangle$ 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 : $ResiduatedTransitionSystem.partial\magma\ resid$
 $\langle proof \rangle$

lemma $null\ char$:
shows $V.null = Null$
 $\langle proof \rangle$

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 $Con\ t\ u$
 $\langle proof \rangle$

lemma $ConE$ [*elim*]:
assumes $Con\ t\ u$
and $\llbracket Arr\ t; Arr\ u; Dom\ t = Dom\ u; Cod\ t = Cod\ u;$
 $residuation.con\ (HOM_{EC}\ (Dom\ t)\ (Cod\ t))\ (Trn\ t)\ (Trn\ u) \rrbracket \Longrightarrow T$
shows T
 $\langle proof \rangle$

lemma $Con\ sym$:
assumes $Con\ t\ u$
shows $Con\ u\ t$
 $\langle proof \rangle$

lemma $resid\ ne\ Null\ imp\ Con$:
assumes $t \setminus u \neq Null$
shows $Con\ t\ u$
 $\langle proof \rangle$

sublocale V : *residuation resid*
 $\langle proof \rangle$

notation $V.con$ (**infix** $\langle \frown \rangle$ 50)

lemma *con-char*:
shows $t \frown u \longleftrightarrow Con\ t\ u$
 $\langle proof \rangle$

lemma *arr-char*:
shows $V.arr\ t \longleftrightarrow Arr\ t$
 $\langle proof \rangle$

lemma *ide-char*:
shows $V.ide\ t \longleftrightarrow Ide\ t$
 $\langle proof \rangle$

lemma *con-implies-Par*:
assumes $t \frown u$
shows $Dom\ t = Dom\ u$ **and** $Cod\ t = Cod\ u$
 $\langle proof \rangle$

lemma *Dom-resid [simp]*:
assumes $t \frown u$
shows $Dom\ (t \setminus u) = Dom\ t$
 $\langle proof \rangle$

lemma *Cod-resid [simp]*:
assumes $t \frown u$
shows $Cod\ (t \setminus u) = Cod\ t$
 $\langle proof \rangle$

lemma *Trn-resid [simp]*:
assumes $t \frown u$
shows $Trn\ (t \setminus u) = HOM_{EC}\ (Dom\ t)\ (Cod\ t)\ (Trn\ t)\ (Trn\ u)$
 $\langle proof \rangle$

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 $V.trg\ t = (if\ V.arr\ t$
 $then\ MkArr\ (Dom\ t)\ (Cod\ t)$
 $(residuation.trg\ (HOM_{EC}\ (Dom\ t)\ (Cod\ t))\ (Trn\ t))$
 $else\ Null)$
 $\langle proof \rangle$

sublocale V : *rts resid*
 $\langle proof \rangle$

lemma *is-rts*:
shows *rts resid*
 ⟨*proof*⟩

sublocale *V*: *extensional-rts resid*
 ⟨*proof*⟩

lemma *is-extensional-rts*:
shows *extensional-rts resid*
 ⟨*proof*⟩

lemma *arr-MkArr* [*intro*]:
assumes $a \in \text{Obj}$ **and** $b \in \text{Obj}$
and *residuation.arr* ($\text{HOM}_{EC} a b$) t
shows $V.\text{arr} (\text{MkArr } a b t)$
 ⟨*proof*⟩

lemma *arr-eqI*:
assumes $t \neq V.\text{null}$ **and** $u \neq V.\text{null}$
and $\text{Dom } t = \text{Dom } u$ **and** $\text{Cod } t = \text{Cod } u$ **and** $\text{Trn } t = \text{Trn } u$
shows $t = u$
 ⟨*proof*⟩

lemma *MkArr-Trn*:
assumes $V.\text{arr } t$
shows $t = \text{MkArr} (\text{Dom } t) (\text{Cod } t) (\text{Trn } t)$
 ⟨*proof*⟩

lemma *src-char*:
shows $V.\text{src } t =$ (if $V.\text{arr } t$
 then $\text{MkArr} (\text{Dom } t) (\text{Cod } t)$
 (rts.src
 ($\text{HOM}_{EC} (\text{Dom } t) (\text{Cod } t)) (\text{Trn } t)$)
 else Null)
 ⟨*proof*⟩

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** $\langle \star \rangle$ 53)
where $t \star u \equiv$
 if $V.\text{arr } t \wedge V.\text{arr } u \wedge \text{Dom } t = \text{Cod } u$
 then $\text{MkArr} (\text{Dom } u) (\text{Cod } t)$
 ($\text{RTS.Map} (\text{Comp} (\text{Dom } u) (\text{Cod } u) (\text{Cod } t))$
 ($\text{RTS.Pack} (\text{Hom} (\text{Dom } t) (\text{Cod } t))$
 ($\text{Hom} (\text{Dom } u) (\text{Cod } u)$)
 ($\text{Trn } t, \text{Trn } u$)))

else V.null

lemma *arr-hcomp*:

assumes $V.arr\ t$ **and** $V.arr\ u$ **and** $Dom\ t = Cod\ u$
shows $V.arr\ (t \star u)$

<proof>

lemma *Dom-hcomp [simp]*:

assumes $V.arr\ t$ **and** $V.arr\ u$ **and** $Dom\ t = Cod\ u$
shows $Dom\ (t \star u) = Dom\ u$

<proof>

lemma *Cod-hcomp [simp]*:

assumes $V.arr\ t$ **and** $V.arr\ u$ **and** $Dom\ t = Cod\ u$
shows $Cod\ (t \star u) = Cod\ t$

<proof>

lemma *Trn-hcomp [simp]*:

assumes $V.arr\ t$ **and** $V.arr\ u$ **and** $Dom\ t = Cod\ u$
shows $Trn\ (t \star u) =$

$RTS.Map\ (Comp\ (Dom\ u)\ (Cod\ u)\ (Cod\ t))$
 $(RTS.Pack\ (Hom\ (Cod\ u)\ (Cod\ t))\ (Hom\ (Dom\ u)\ (Cod\ u))$
 $(Trn\ t,\ Trn\ u))$

<proof>

lemma *hcomp-Null [simp]*:

shows $t \star Null = Null$ **and** $Null \star u = Null$

<proof>

sublocale H : *Category.partial-magma hcomp*

<proof>

lemma *H-null-char*:

shows $H.null = V.null$

<proof>

sublocale H : *partial-composition hcomp* *<proof>*

lemma *H-composable-char*:

shows $t \star u \neq V.null \iff V.arr\ t \wedge V.arr\ u \wedge Dom\ t = Cod\ u$

<proof>

definition *horizontal-unit*

where *horizontal-unit* $a \equiv$

$V.arr\ a \wedge Dom\ a = Cod\ a \wedge$
 $(\forall t. (V.arr\ t \wedge Dom\ t = Cod\ a \implies t \star a = t) \wedge$
 $(V.arr\ t \wedge Dom\ a = Cod\ t \implies a \star t = t))$

lemma *H-ide-char*:

shows $H.ide\ a \longleftrightarrow horizontal\text{-}unit\ a$
 ⟨*proof*⟩

Each $A \in Obj$ determines a corresponding identity for horizontal composition; namely, the transition of $HOM_{EC}\ A\ A$ obtained by evaluating the simulation $\langle Id\ A : One \rightarrow Hom\ A\ A \rangle$ at the unique arrow $RTS.One.the\text{-}arr$ of the underlying one-arrow RTS of One .

abbreviation $mkobj$
where $mkobj\ A \equiv MkArr\ A\ A\ (RTS.Map\ (Id\ A)\ RTS.One.the\text{-}arr)$

lemma *Id-yields-horiz-ide*:
assumes $A \in Obj$
shows $H.ide\ (mkobj\ A)$
 ⟨*proof*⟩

lemma *H-ide-is-V-ide*:
assumes $H.ide\ a$
shows $V.ide\ a$
 ⟨*proof*⟩

lemma *H-domains-char*:
shows $H.domains\ t = \{a. V.arr\ t \wedge a = mkobj\ (Dom\ t)\}$
 ⟨*proof*⟩

lemma *H-codomains-char*:
shows $H.codomains\ t = \{a. V.arr\ t \wedge a = mkobj\ (Cod\ t)\}$
 ⟨*proof*⟩

lemma *H-arr-char*:
shows $H.arr\ t \longleftrightarrow t \neq Null \wedge Dom\ t \in Obj \wedge Cod\ t \in Obj \wedge$
 $residuation.arr\ (HOM_{EC}\ (Dom\ t)\ (Cod\ t))\ (Trn\ t)$
 ⟨*proof*⟩

lemma *H-seq-char*:
shows $H.seq\ t\ u \longleftrightarrow V.arr\ t \wedge V.arr\ u \wedge Dom\ t = Cod\ u$
 ⟨*proof*⟩

sublocale H : *category hcomp*
 ⟨*proof*⟩

lemma *is-category*:
shows *category hcomp*
 ⟨*proof*⟩

lemma *H-dom-char*:
shows $H.dom =$
 (λ*t*. if $H.arr\ t$
 then $MkArr\ (Dom\ t)\ (Dom\ t)$
 ($RTS.Map\ (Id\ (Dom\ t))\ RTS.One.the\text{-}arr$)

else $V.null$)
⟨proof⟩

lemma *H-dom-simp*:

assumes $V.arr\ t$

shows $H.dom\ t = MkArr\ (Dom\ t)\ (Dom\ t)$
 $(RTS.Map\ (Id\ (Dom\ t))\ RTS.One.the-arr)$

⟨proof⟩

lemma *H-cod-char*:

shows $H.cod =$

$(\lambda t. \text{if } H.arr\ t$
then $MkArr\ (Cod\ t)\ (Cod\ t)$
 $(RTS.Map\ (Id\ (Cod\ t))\ RTS.One.the-arr)$
else $V.null$)

⟨proof⟩

lemma *H-cod-simp*:

assumes $V.arr\ t$

shows $H.cod\ t = MkArr\ (Cod\ t)\ (Cod\ t)$
 $(RTS.Map\ (Id\ (Cod\ t))\ RTS.One.the-arr)$

⟨proof⟩

lemma *con-implies-H-par*:

assumes $V.con\ t\ u$

shows $H.par\ t\ u$

⟨proof⟩

lemma *H-par-resid*:

assumes $V.con\ t\ u$

shows $H.par\ t\ (resid\ t\ u)$

⟨proof⟩

lemma *simulation-dom*:

shows $simulation\ resid\ resid\ H.dom$

⟨proof⟩

lemma *simulation-cod*:

shows $simulation\ resid\ resid\ H.cod$

⟨proof⟩

sublocale *dom*: $simulation\ resid\ resid\ H.dom$

⟨proof⟩

sublocale *cod*: $simulation\ resid\ resid\ H.cod$

⟨proof⟩

sublocale *RR*: $fibred-product-rts\ resid\ resid\ resid\ H.dom\ H.cod$ ⟨proof⟩

sublocale *H*: $simulation\ RR.resid\ resid$

$\langle \lambda t. \text{if } RR.arr\ t \text{ then } fst\ t \star snd\ t \text{ else } V.null \rangle$

<proof>

lemma *simulation-hcomp*:

shows *simulation RR.resid resid*

$(\lambda t. \text{if } RR.\text{arr } t \text{ then } \text{fst } t \star \text{snd } t \text{ else } V.\text{null})$

<proof>

lemma *Dom-src [simp]*:

assumes *V.arr t*

shows $Dom (V.\text{src } t) = Dom t$

<proof>

lemma *Dom-trg [simp]*:

assumes *V.arr t*

shows $Dom (V.\text{trg } t) = Dom t$

<proof>

lemma *Cod-src [simp]*:

assumes *V.arr t*

shows $Cod (V.\text{src } t) = Cod t$

<proof>

lemma *Cod-trg [simp]*:

assumes *V.arr t*

shows $Cod (V.\text{trg } t) = Cod t$

<proof>

lemma *null-coincidence [simp]*:

shows $H.\text{null} = V.\text{null}$

<proof>

lemma *arr-coincidence [simp]*:

shows $H.\text{arr} = V.\text{arr}$

<proof>

lemma *dom-src [simp]*:

shows $H.\text{dom} (V.\text{src } t) = H.\text{dom } t$

<proof>

lemma *src-dom [simp]*:

shows $V.\text{src} (H.\text{dom } t) = H.\text{dom } t$

<proof>

lemma *small-homs*:

shows $\text{small} (H.\text{hom } a \ b)$

<proof>

Note that the arrow type of the RTS-category given by the following is $(\text{'O}, \text{'A}) \text{ arr}$, 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 $('O, 'A) \text{ arr}$, and then we try to go back to an RTS-enriched category, the hom-RTS's will have arrow type $('O, 'A) \text{ arr}$, 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 $('O, 'A) \text{ arr}$ back to $'A$.

sublocale *rts-category resid hcomp*
 ⟨*proof*⟩

proposition *is-rts-category:*
shows *rts-category resid hcomp*
 ⟨*proof*⟩

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 $('O, 'A) \text{ arr}$, 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: $\exists \varphi :: 'O \Rightarrow 'A. \text{inj-on } \varphi \text{ Obj}$*
begin

We will use R to refer to the RTS constructed from the given enriched category.

abbreviation $R :: ('O, 'A) \text{ arr resid}$
where $R \equiv \text{resid}$

The locale assumptions are sufficient to allow us to uniquely encode each element of $\text{Collect arr} \cup \{\text{null}\}$ as single element of $'A$.

lemma *ex-arrow-injection*:

shows $\exists i :: ('O, 'A) \text{arr} \Rightarrow 'A. \text{inj-on } i \text{ (Collect arr} \cup \{\text{null}\})$
 $\langle \text{proof} \rangle$

lemma *bij-betw-Obj-horiz-ide*:

shows $\text{bij-betw mkobj Obj (Collect H.ide)}$
 $\langle \text{proof} \rangle$

lemma *ex-isomorphic-image-rts*:

shows $\exists R' (UP :: 'A \Rightarrow ('O, 'A) \text{arr}) (DN :: ('O, 'A) \text{arr} \Rightarrow 'A).$
 $\text{small-rts } R' \wedge \text{extensional-rts } R' \wedge \text{inverse-simulations } R R' UP DN$
 $\langle \text{proof} \rangle$

We now choose some RTS with the properties asserted by the previous lemma, along with the invertible simulations that relate it to R .

definition $R' :: 'A \text{ resid}$

where $R' \equiv \text{SOME } R'. \exists UP DN. \text{small-rts } R' \wedge \text{extensional-rts } R' \wedge$
 $\text{inverse-simulations resid } R' UP DN$

definition $UP :: 'A \Rightarrow ('O, 'A) \text{arr}$

where $UP \equiv \text{SOME } UP. \exists DN. \text{small-rts } R' \wedge \text{extensional-rts } R' \wedge$
 $\text{inverse-simulations resid } R' UP DN$

definition $DN :: ('O, 'A) \text{arr} \Rightarrow 'A$

where $DN \equiv \text{SOME } DN. \text{small-rts } R' \wedge \text{extensional-rts } R' \wedge$
 $\text{inverse-simulations resid } R' UP DN$

lemma $R' \text{-prop}$:

shows $\exists UP DN. \text{small-rts } R' \wedge \text{extensional-rts } R' \wedge$
 $\text{inverse-simulations } R R' UP DN$
 $\langle \text{proof} \rangle$

sublocale $R': \text{extensional-rts } R'$

$\langle \text{proof} \rangle$

sublocale $R': \text{small-rts } R'$

$\langle \text{proof} \rangle$

lemma $\text{extensional-rts-}R'$:

shows $\text{extensional-rts } R'$
 $\langle \text{proof} \rangle$

lemma $\text{small-rts-}R'$:

shows $\text{small-rts } R'$
 $\langle \text{proof} \rangle$

sublocale $UP\text{-}DN: \text{inverse-simulations } R R' UP DN$

$\langle \text{proof} \rangle$

lemma *inverse-simulations-UP-DN*:
shows *inverse-simulations resid R' UP DN*
 ⟨*proof*⟩

lemma *R'-src-char*:
shows $R'.src = DN \circ src \circ UP$
 ⟨*proof*⟩

lemma *R'-trg-char*:
shows $R'.trg = DN \circ trg \circ UP$
 ⟨*proof*⟩

We transport the horizontal composition (\star) to R' via the isomorphisms UP and DN .

abbreviation $hcomp' :: 'A \text{ resid } (\text{infixr } \langle \star' \rangle 53)$
where $t \star' u \equiv DN (UP t \star UP u)$

interpretation H' : *Category.partial-magma hcomp'*
 ⟨*proof*⟩

lemma *H'-null-char*:
shows $H'.null = DN \text{ null}$
 ⟨*proof*⟩

interpretation H' : *partial-composition $\langle \lambda t u. DN (hcomp (UP t) (UP u)) \rangle$*
 ⟨*proof*⟩

lemma *H'-ide-char*:
shows $H'.ide t \longleftrightarrow H.ide (UP t)$
 ⟨*proof*⟩

lemma *H'-domains-char*:
shows $H'.domains t = DN \text{ ' } H.domains (UP t)$
 ⟨*proof*⟩

lemma *H'-codomains-char*:
shows $H'.codomains t = DN \text{ ' } H.codomains (UP t)$
 ⟨*proof*⟩

lemma *H'-arr-char*:
shows $H'.arr t = H.arr (UP t)$
 ⟨*proof*⟩

lemma *H'-seq-char*:
shows $H'.seq t u \longleftrightarrow H.seq (UP t) (UP u)$
 ⟨*proof*⟩

sublocale H' : *category hcomp'*

<proof>

lemma *hcomp'-is-category*:

shows *category hcomp'*

<proof>

lemma *H'-dom-char*:

shows $H'.dom = DN \circ H.dom \circ UP$

<proof>

lemma *H'-cod-char*:

shows $H'.cod = DN \circ H.cod \circ UP$

<proof>

lemma *null'-coincidence [simp]*:

shows $H'.null = R'.null$

<proof>

lemma *arr'-coincidence [simp]*:

shows $H'.arr = R'.arr$

<proof>

lemma *H'-hom-char*:

shows $H'.hom\ a\ b = DN \circ H.hom\ (UP\ a)\ (UP\ b)$

<proof>

interpretation *dom'*: *simulation R' R' H'.dom*

<proof>

interpretation *cod'*: *simulation R' R' H'.cod*

<proof>

lemma *R'-con-char*:

shows $R'.con\ t\ u \longleftrightarrow V.con\ (UP\ t)\ (UP\ u)$

<proof>

sublocale *R'R'*: *fibred-product-rts R' R' R' H'.dom H'.cod* *<proof>*

sublocale *H'*: *simulation R'R'.resid R'*

*<\t. if R'R'.arr t then fst t *' snd t else R'.null>*

<proof>

proposition *is-locally-small-rts-category*:

shows *locally-small-rts-category R' hcomp'*

<proof>

end

5.2.2 Functoriality

```

locale rts-functor-of-enriched-functor =
  universe arr-type +
  RTS: rtscat arr-type +
  A: rts-enriched-category arr-type ObjA HomA IdA CompA +
  B: rts-enriched-category arr-type ObjB HomB IdB CompB +
  EF: rts-enriched-functor
    ObjA HomA IdA CompA ObjB HomB IdB CompB Fo Fa
for ObjA :: 'a set
and HomA :: 'a ⇒ 'a ⇒ 'A rtscatx.arr
and IdA :: 'a ⇒ 'A rtscatx.arr
and CompA :: 'a ⇒ 'a ⇒ 'a ⇒ 'A rtscatx.arr
and ObjB :: 'b set
and HomB :: 'b ⇒ 'b ⇒ 'A rtscatx.arr
and IdB :: 'b ⇒ 'A rtscatx.arr
and CompB :: 'b ⇒ 'b ⇒ 'b ⇒ 'A rtscatx.arr
and Fo :: 'a ⇒ 'b
and Fa :: 'a ⇒ 'a ⇒ 'A rtscatx.arr
begin

```

```

  interpretation A: rts-category-of-enriched-category
    arr-type ObjA HomA IdA CompA

```

⟨*proof*⟩

```

  interpretation B: rts-category-of-enriched-category
    arr-type ObjB HomB IdB CompB

```

⟨*proof*⟩

definition *F*

```

where F t ≡ if residuation.arr A.resid t
  then B.MkArr (Fo (A.Dom t)) (Fo (A.Cod t))
    (RTS.Map (Fa (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*⟩

sublocale *rts-functor A.resid A.hcomp B.resid B.hcomp F*

⟨*proof*⟩

lemma *is-rts-functor:*

shows *rts-functor A.resid A.hcomp B.resid B.hcomp F*

⟨*proof*⟩

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 (infixr <\> 70)
and hcomp :: 'A comp (infixr <*> 53)
begin

  sublocale RTS: rtscat arr-type <proof>

  no-notation V.comp (infixr <\> 55)
  no-notation H.in-hom (infixr <«- : - → -»>)
  no-notation RTS.prod (infixr <⊗> 51)

  notation RTS.in-hom (infixr <«- : - → -»>)
  notation RTS.CMC.tensor (infixr <⊗> 51)
  notation RTS.CMC.unity (infixr <1>)
  notation RTS.CMC.assoc (infixr <a[-, -, -]>)
  notation RTS.CMC.lunit (infixr <l[-]>)
  notation RTS.CMC.runit (infixr <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))
    (λt. 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))

```

$(RTS.Rts (Hom a c))$
 $(\lambda t. (\lambda x. fst x \star snd x) (RTS.Unpack (Hom b c) (Hom a b) t))$

lemma *ide-Hom* [*intro, simp*]:
assumes $a \in Obj$ **and** $b \in Obj$
shows $RTS.ide (Hom a b)$
 $\langle proof \rangle$

lemma
assumes $a \in Obj$ **and** $b \in Obj$
shows $HOM-null-char: ResiduatedTransitionSystem.partial-magma.null$
 $(RTS.Rts (Hom a b)) =$
 $null$
and $HOM-arr-char:$
 $residuation.arr (RTS.Rts (Hom a b)) t \longleftrightarrow H.in-hom t a b$
 $\langle proof \rangle$

lemma *Id-in-hom* [*intro*]:
assumes $a \in Obj$
shows $\langle Id a : \mathbf{1} \rightarrow Hom a a \rangle$
 $\langle proof \rangle$

lemma *Id-simps* [*simp*]:
assumes $a \in Obj$
shows $RTS.arr (Id a)$
and $RTS.dom (Id a) = \mathbf{1}$
and $RTS.cod (Id a) = Hom a a$
 $\langle proof \rangle$

lemma *Comp-in-hom* [*intro, simp*]:
assumes $a \in Obj$ **and** $b \in Obj$ **and** $c \in Obj$
shows $\langle Comp a b c : Hom b c \otimes Hom a b \rightarrow Hom a c \rangle$
 $\langle proof \rangle$

lemma *Comp-simps* [*simp*]:
assumes $a \in Obj$ **and** $b \in Obj$ **and** $c \in Obj$
shows $RTS.arr (Comp a b c)$
and $RTS.dom (Comp a b c) = Hom b c \otimes Hom a b$
and $RTS.cod (Comp a b c) = Hom a c$
 $\langle proof \rangle$

lemma *Map-Comp-Pack*:
assumes $a \in Obj$ **and** $b \in Obj$ **and** $c \in Obj$
and $residuation.arr$
 $(product-rts.resid (RTS.Rts (Hom b c)) (RTS.Rts (Hom a b))) x$
shows $RTS.Map (Comp a b c) (RTS.Pack (Hom b c) (Hom a b) x) =$
 $fst x \star snd x$

<proof>

sublocale *rts-enriched-category arr-type Obj Hom Id Comp*
<proof>

proposition *is-rts-enriched-category:*
shows *rts-enriched-category Obj Hom Id Comp*
<proof>

lemma *HOM-agreement:*
assumes *H.ide a and H.ide b*
shows *HOM_{EC} a b = HOM a b*
<proof>

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*
<proof>

interpretation *B: enriched-category-of-rts-category arr-type resid_B comp_B*
<proof>

definition F_o
where $F_o a \equiv$ if $A.H.ide a$ then $F a$ else $B.null$

definition F_a
where $F_a a b \equiv$ if $A.H.ide a \wedge A.H.ide b$
 then $RTS.mkarr (A.HOM_{EC} a b) (B.HOM_{EC} (F_o a) (F_o b))$
 ($\lambda t.$ 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 $a \in A.Obj$ **and** $b \in A.Obj$
shows $sub-rts.resid resid_A (\lambda t. A.H.in-hom t a b) = A.HOM_{EC} a b$
and $sub-rts.resid resid_B (\lambda t. B.H.in-hom t (F_o a) (F_o b)) =$
 $B.HOM_{EC} (F_o a) (F_o b)$
 $\langle proof \rangle$

sublocale *rts-enriched-functor*
 $\langle Collect A.H.ide \rangle A.Hom A.Id A.Comp$
 $\langle Collect B.H.ide \rangle B.Hom B.Id B.Comp$
 $F_o F_a$
 $\langle proof \rangle$

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*
 $Obj Hom Id Comp \langle proof \rangle$

no-notation $RTS.prod$ (**infixr** $\langle \otimes \rangle$ 51)

interpretation Trn : *simulation RC.resid resid*
 $\langle \lambda t. if RC.arr t then RC.Trn t else null \rangle$
 $\langle proof \rangle$

interpretation *MkArr: simulation resid RC.resid*
 $\langle \lambda t. \text{if arr } t \text{ then } RC.MkArr (H.dom t) (H.cod t) t$
 $\text{else } RC.null \rangle$

$\langle \text{proof} \rangle$

interpretation *Trn-MkArr: inverse-simulations resid RC.resid*
 $\langle \lambda t. \text{if } RC.arr t \text{ then } RC.Trn t \text{ else null} \rangle$
 $\langle \lambda t. \text{if arr } t \text{ then } RC.MkArr (H.dom t) (H.cod t) t$
 $\text{else } RC.null \rangle$

$\langle \text{proof} \rangle$

lemma *inverse-simulations-Trn-MkArr:*
shows *inverse-simulations resid RC.resid*
 $(\lambda t. \text{if } RC.arr t \text{ then } RC.Trn t \text{ else null})$
 $(\lambda t. \text{if arr } t \text{ then } RC.MkArr (H.dom t) (H.cod t) t \text{ else } RC.null)$
 $\langle \text{proof} \rangle$

interpretation *Trn: functor RC.hcomp hcomp*
 $\langle \lambda t. \text{if } RC.arr t \text{ then } RC.Trn t \text{ else null} \rangle$

$\langle \text{proof} \rangle$

interpretation *MkArr: functor hcomp RC.hcomp*
 $\langle \lambda t. \text{if arr } t \text{ then } RC.MkArr (H.dom t) (H.cod t) t$
 $\text{else } RC.null \rangle$

$\langle \text{proof} \rangle$

interpretation *Trn-MkArr: inverse-functors hcomp RC.hcomp*
 $\langle \lambda t. \text{if } RC.arr t \text{ then } RC.Trn t \text{ else null} \rangle$
 $\langle \lambda t. \text{if arr } t$
 $\text{then } RC.MkArr (H.dom t) (H.cod t) t$
 $\text{else } RC.null \rangle$

$\langle \text{proof} \rangle$

lemma *inverse-functors-Trn-MkArr:*
shows *inverse-functors hcomp RC.hcomp*
 $(\lambda t. \text{if } RC.arr t \text{ then } RC.Trn t \text{ else null})$
 $(\lambda t. \text{if arr } t \text{ then } RC.MkArr (H.dom t) (H.cod t) t \text{ else } RC.null)$
 $\langle \text{proof} \rangle$

proposition *induces-rts-category-isomorphism:*
shows *rts-category-isomorphism resid hcomp RC.resid RC.hcomp*
 $(\lambda t. \text{if arr } t \text{ then } RC.MkArr (H.dom t) (H.cod t) t \text{ else } RC.null)$
 $\langle \text{proof} \rangle$

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'*
 $\langle proof \rangle$

interpretation R'' : *enriched-category-of-rts-category* *arr-type* R' *hcomp'*
 $\langle proof \rangle$

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 $\langle F_a\ a\ b : Hom\ a\ b \rightarrow R''.Hom\ (F_o\ a)\ (F_o\ b) \rangle$ 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 $F_a \equiv \lambda a\ b. \text{if } a \in Obj \wedge b \in Obj$
 then $RTS.mkarr\ (HOM_{EC}\ a\ b)\ (R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b))$
 ($\lambda t. \text{if } residuation.arr\ (HOM_{EC}\ a\ b)\ t$
 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 $a \in \text{Obj}$

shows $\text{DN} (\text{MkArr } a \ a \ (\text{RTS.Map } (\text{Id } a) \ \text{RTS.One.the-arr})) \in \text{Collect } H'.\text{ide}$
<proof>

lemma *bij-F_o*:

shows *bij-betw* $F_o \ \text{Obj } R''.\text{Obj}$

<proof>

lemma *F_a-in-hom* [*intro, simp*]:

assumes $a \in \text{Obj}$ **and** $b \in \text{Obj}$

shows $\langle F_a \ a \ b : \text{Hom } a \ b \rightarrow R''.\text{Hom } (F_o \ a) \ (F_o \ b) \rangle$

<proof>

lemma *F_a-simps* [*simp*]:

assumes $a \in \text{Obj}$ **and** $b \in \text{Obj}$

shows $\text{RTS.arr } (F_a \ a \ b)$

and $\text{RTS.dom } (F_a \ a \ b) = \text{Hom } a \ b$

and $\text{RTS.cod } (F_a \ a \ b) = R''.\text{Hom } (F_o \ a) \ (F_o \ b)$

<proof>

lemma *Map-F_a-simp* [*simp*]:

assumes $a \in \text{Obj}$ **and** $b \in \text{Obj}$ **and** *residuation.arr* ($\text{HOM}_{EC} \ a \ b$) t

shows $\text{RTS.Map } (F_a \ a \ b) \ t = \text{DN} (\text{MkArr } a \ b \ t)$

<proof>

interpretation Φ : *rts-enriched-functor*

$\text{Obj Hom Id Comp } R''.\text{Obj } R''.\text{Hom } R''.\text{Id } R''.\text{Comp}$
 $F_o \ F_a$

<proof>

lemma *induces-rts-enriched-functor*:

shows *rts-enriched-functor*

$\text{Obj Hom Id Comp } R''.\text{Obj } R''.\text{Hom } R''.\text{Id } R''.\text{Comp } F_o \ F_a$

<proof>

proposition *induces-fully-faithful-rts-enriched-functor*:

shows *fully-faithful-rts-enriched-functor*

$\text{Obj Hom Id Comp } R''.\text{Obj } R''.\text{Hom } R''.\text{Id } R''.\text{Comp } F_o \ F_a$

<proof>

end

5.5 \mathbf{RTS}^\dagger Determined by its Underlying Category

In this section we show that the category \mathbf{RTS}^\dagger is fully determined by its subcategory \mathbf{RTS} comprising the arrows that are identities for the residuation. Specifically, we show that there is an invertible RTS-functor from \mathbf{RTS}^\dagger to the RTS-category obtained from the category \mathbf{RTS} regarded as a category enriched in itself.

context *rtscat*
begin

The following produces a stand-alone instance of the category \mathbf{RTS}^\dagger , independent of the current context. Arrows of \mathbf{RTS}^\dagger 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* \langle *proof* \rangle

In the current locale context, *comp* is the composition for the ordinary category \mathbf{RTS} . As a cartesian closed category, this category determines a category enriched in itself.

interpretation *enriched-category comp Prod α ι*
Collect ide \langle *exp ECMC.Id ECMC.Comp*
 \langle *proof* \rangle

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 \mathbf{RC} . Arrows of \mathbf{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 \mathbf{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 \langle *Collect ide* \langle *exp ECMC.Id ECMC.Comp*
 \langle *proof* \rangle

We now define the mapping Φ which we will show to be an RTS-category isomorphism from \mathbf{RC} to \mathbf{RTS} . In order to map an arrow *MkArr* a b t of \mathbf{RC} to an arrow of \mathbf{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 $\Phi :: ('A$ *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 $RTSx.arr\ (\Phi\ t)$
and $RTSx.dom\ (\Phi\ t) = RTSx.mkobj\ (RTSx.Dom\ (RC.Dom\ t))$
and $RTSx.cod\ (\Phi\ t) = RTSx.mkobj\ (RTSx.Dom\ (RC.Cod\ t))$
 $\langle proof \rangle$

lemma Φ -in-hom [intro]:
assumes $RC.arr\ t$
shows $RTSx.H.in-hom\ (\Phi\ t)$
 $(RTSx.mkobj\ (RTSx.Dom\ (RC.Dom\ t)))\ (RTSx.mkobj\ (RTSx.Dom\ (RC.Cod\ t)))$
 $\langle proof \rangle$

interpretation Φ : simulation $RC.resid\ RTSx.resid\ \Phi$
 $\langle proof \rangle$

The following fact is key to showing that Φ is functorial.

lemma *Func-Trn-obj*:
assumes $RC.obj\ a$
shows $Func\ (RC.Dom\ a)\ (RC.Cod\ a)\ (RC.Trn\ a) =$
 $exponential-rts.MkIde\ (I\ (Rts\ (RC.Dom\ a)))$
 $\langle proof \rangle$

lemma *obj- Φ -obj*:
assumes $RC.obj\ a$
shows $RTSx.obj\ (\Phi\ a)$
 $\langle proof \rangle$

interpretation Φ : functor $RC.hcomp\ RTSx.hcomp\ \Phi$
 $\langle proof \rangle$

interpretation Φ : rts-functor $RC.resid\ RC.hcomp$
 $RTSx.resid\ RTSx.hcomp\ \Phi$
 $\langle proof \rangle$

interpretation Φ : fully-faithful-functor $RC.hcomp\ RTSx.hcomp\ \Phi$
 $\langle proof \rangle$

interpretation Φ : full-embedding-functor $RC.hcomp\ RTSx.hcomp\ \Phi$
 $\langle proof \rangle$

interpretation Φ : invertible-functor $RC.hcomp\ RTSx.hcomp\ \Phi$
 $\langle proof \rangle$

interpretation Φ : invertible-simulation $RC.resid\ RTSx.resid\ \Phi$
 $\langle proof \rangle$

theorem *rts-category-isomorphism* *RC.resid RC.hcomp*
RTSx.resid RTSx.hcomp Φ

<proof>

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.