Residuated Transition Systems II: Categorical Properties

Eugene W. Stark

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

June 17, 2024

Abstract

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

Contents

Introduction

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

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

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

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

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

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

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

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

Chapter 1

Preliminaries

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

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

lemma (**in** *extensional-rts*) *divisors-of-ide*:
assumes *composite-of t u v* **and** *ide v*
shows *ide t* **and** *ide u*
⟨*proof*⟩

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** $\backslash_A$ *70*)
and *B* :: *'b resid* (**infix** $\backslash_B$ *70*)
and *b* :: *'b* +
assumes *ide-b*: *B.ide b*
begin

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

  sublocale *simulation A B map*
    ⟨*proof*⟩

end

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

  lemma *inv-simp* [*simp*]:
  assumes *B.arr y*
  shows *G* (*F y*) = *y*
    ⟨*proof*⟩

  lemma *inv'-simp* [*simp*]:
  assumes *A.arr x*
  shows *F* (*G x*) = *x*
    ⟨*proof*⟩

  lemma *induce-bij-betw-arr-sets*:
  shows *bij-betw F* (*Collect B.arr*) (*Collect A.arr*)
    ⟨*proof*⟩

end

lemma *inverse-simulations-sym*:
assumes *inverse-simulations A B F G*

shows *inverse-simulations B A G F*
⟨*proof*⟩

locale *invertible-simulation* =
simulation +
assumes *invertible*: ∃ *G*. *inverse-simulations A B G F*

lemma *invertible-simulation-def′*:
shows *invertible-simulation A B F* ⟷ (∃ *G*. *inverse-simulations A B G F*)
⟨*proof*⟩

lemma *invertible-simulation-iff*:
shows *invertible-simulation A B F* ⟷
simulation A B F ∧
bij-betw F (*Collect* (*residuation.arr A*)) (*Collect* (*residuation.arr B*)) ∧
(∀ *t u*. *residuation.con B* (*F t*) (*F u*) ⟶ *residuation.con A t u*)
⟨*proof*⟩

context *invertible-simulation*
begin

lemma *is-bijection-betw-arr-sets*:
shows *bij-betw F* (*Collect A.arr*) (*Collect B.arr*)
⟨*proof*⟩

lemma *reflects-con*:
assumes *residuation.con B* (*F t*) (*F u*)
shows *residuation.con A t u*
⟨*proof*⟩

end

context *inverse-simulations*
begin

sublocale *F*: *invertible-simulation B A F*
⟨*proof*⟩

sublocale *G*: *invertible-simulation A B G*
⟨*proof*⟩

end

lemma *inverse-simulation-unique*:
assumes *inverse-simulations A B G F*
and *inverse-simulations A B G′ F*
shows *G* = *G′*
⟨*proof*⟩

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

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

  interpretation *inverse-simulations A B map F*
    ⟨*proof*⟩

  sublocale *simulation B A map* ⟨*proof*⟩

  lemma *is-simulation*:
  shows *simulation B A map*
    ⟨*proof*⟩

  sublocale *inverse-simulations A B map F* ⟨*proof*⟩

  lemma *map-simp*:
  assumes *B.arr x*
  shows *map x* = *inv-into* (*Collect A.arr*) *F x*
    ⟨*proof*⟩

  lemma *map-eq*:
  shows *map* = (λ*x*. *if B.arr x then inv-into* (*Collect A.arr*) *F x else A.null*)
    ⟨*proof*⟩

end

lemma *invertible-simulation-identity*:
assumes *rts A*
shows [*intro*]: *invertible-simulation A A* (*I A*)
and *inverse-simulations A A* (*I A*) (*I A*)
and *inverse-simulation.map A A* (*I A*) = *I A*
⟨*proof*⟩

lemma *inverse-simulations-compose*:
assumes *inverse-simulations A B F′ F* **and** *inverse-simulations B C G′ G*
shows *inverse-simulations A C* (*F′* ∘ *G′*) (*G* ∘ *F*)
⟨*proof*⟩

lemma *invertible-simulation-comp* [*intro*]:
assumes *invertible-simulation A B F* **and** *invertible-simulation B C G*
shows *invertible-simulation A C* (*G* ∘ *F*)
and *inverse-simulations A C*
      (*inverse-simulation.map A B F* ∘ *inverse-simulation.map B C G*) (*G* ∘ *F*)

and *inverse-simulation.map A C* $(G \circ F) =$
inverse-simulation.map A B F $\circ$ *inverse-simulation.map B C G*
⟨*proof*⟩

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

lemma *invertible-simulation-cancel-left*:
assumes *invertible-simulation A B H*
shows $[\![$*simulation C A F*; *simulation C A G*; $H \circ F = H \circ G]\!] \Longrightarrow F = G$
⟨*proof*⟩

lemma *invertible-simulation-cancel-right*:
assumes *invertible-simulation A B H*
shows $[\![$*simulation B C F*; *simulation B C G*; $F \circ H = G \circ H]\!] \Longrightarrow F = G$
⟨*proof*⟩

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

lemma *isomorphic-rts-reflexive*:
assumes *rts A*
shows *isomorphic-rts A A*
⟨*proof*⟩

lemma *isomorphic-rts-symmetric*:
assumes *isomorphic-rts A B*
shows *isomorphic-rts B A*
⟨*proof*⟩

lemma *isomorphic-rts-transitive* [*trans*]:
assumes *isomorphic-rts A B* **and** *isomorphic-rts B C*
shows *isomorphic-rts A C*
⟨*proof*⟩

lemma (**in** *simulation*) *simulation-inv-intoI*:
assumes *inj-on F* (*Collect A.arr*) **and** *F* ‘ (*Collect A.arr*) = *Collect B.arr*
and $\bigwedge t.\ \neg$ *B.arr t* $\Longrightarrow$ *inv-into* (*Collect A.arr*) *F t* = *A.null*

and $\bigwedge t\ u.\ t \frown_B u \Longrightarrow$
$\quad$ *inv-into* (*Collect A.arr*) *F t* $\frown_A$ *inv-into* (*Collect A.arr*) *F u*
shows *simulation B A* (*inv-into* (*Collect A.arr*) *F*)
⟨*proof*⟩

1.2 Transformations

lemma (**in** *transformation*) *preserves-arr*:
assumes *A.arr t*
shows *B.arr* (τ *t*)
⟨*proof*⟩

lemma (**in** *transformation*) *preserves-con*:
assumes *t* $\frown_A$ *u*
shows τ *t* $\frown_B$ τ *u* **and** τ *t* $\frown_B$ *F u*
⟨*proof*⟩

lemma (**in** *transformation*) *naturality1'*:
assumes *A.arr t*
shows *B.composite-of* (*F t*) (τ (*A.trg t*)) (τ *t*)
⟨*proof*⟩

lemma (**in** *transformation*) *naturality2'*:
assumes *A.arr t*
shows *B.composite-of* (τ (*A.src t*)) (*G t*) (τ *t*)
⟨*proof*⟩

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

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

lemma *naturality1'*$_E$:
shows *F t* $\cdot_B$ τ (*A.trg t*) = τ *t*
and *A.arr t* $\Longrightarrow$ *B.composable* (*F t*) (τ (*A.trg t*))
⟨*proof*⟩

lemma *naturality2'*$_E$:
shows τ (*A.src t*) $\cdot_B$ *G t* = τ *t*
and *A.arr t* $\Longrightarrow$ *B.composable* (τ (*A.src t*)) (*G t*)
⟨*proof*⟩

lemma *naturality3'*$_E$:
shows τ (*A.src t*) $\sqcup_B$ *F t* = τ *t*
and *A.arr t* $\Longrightarrow$ *B.joinable* (τ (*A.src t*)) (*F t*)
⟨*proof*⟩

lemma *naturality*$_E$:
shows $\tau\ (A.src\ t) \cdot_B G\ t = F\ t \cdot_B \tau\ (A.trg\ t)$
⟨*proof*⟩

lemma *general-naturality*:
assumes *A.con x y*
shows $\tau\ x \setminus_B F\ y = \tau\ (x \setminus_A y)$
and $F\ x \setminus_B \tau\ y = G\ (x \setminus_A y)$
⟨*proof*⟩

lemma *preserves-prfx*:
assumes $t \lesssim_A u$
shows $\tau\ t \lesssim_B \tau\ u$
⟨*proof*⟩

end

locale *transformation-by-components* =
A: *weakly-extensional-rts A* +
B: *extensional-rts B* +
F: *simulation A B F* +
G: *simulation A B G*
for *A* :: *'a resid* (**infix** $\setminus_A$ *55*)
and *B* :: *'b resid* (**infix** $\setminus_B$ *55*)
and *F* :: $'a \Rightarrow 'b$
and *G* :: $'a \Rightarrow 'b$
and τ :: $'a \Rightarrow 'b$ +
assumes *preserves-src*: $A.ide\ a \Longrightarrow B.src\ (\tau\ a) = F\ a$
and *preserves-trg*: $A.ide\ a \Longrightarrow B.trg\ (\tau\ a) = G\ a$
and *naturality1*: $A.arr\ t \Longrightarrow \tau\ (A.src\ t) \setminus_B F\ t = \tau\ (A.trg\ t)$
and *naturality2*: $A.arr\ t \Longrightarrow F\ t \setminus_B \tau\ (A.src\ t) = G\ t$
and *joinable*: $A.arr\ t \Longrightarrow B.joinable\ (\tau\ (A.src\ t))\ (F\ t)$
begin

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

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

lemma *map-eq*:
shows $map\ t = (if\ A.arr\ t\ then\ \tau\ (A.src\ t) \sqcup_B F\ t\ else\ B.null)$
⟨*proof*⟩

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

assumes *A.ide a*
shows *map a* = *τ a*
⟨*proof*⟩

sublocale *transformation A B F G map*
⟨*proof*⟩

lemma *is-transformation*:
shows *transformation A B F G map*
⟨*proof*⟩

end

locale *identity-transformation* =
transformation +
assumes *identity*: *A.ide a* ⟹ *B.ide* (*τ a*)
begin

lemma *src-eq-trg*:
shows *F* = *G*
⟨*proof*⟩

sublocale *simulation* ⟨*proof*⟩

end

lemma *comp-identity-transformation*:
assumes *transformation A B F G T*
shows *I B* ∘ *T* = *T*
⟨*proof*⟩

lemma *comp-transformation-identity*:
assumes *transformation A B F G T*
shows *T* ∘ *I A* = *T*
⟨*proof*⟩

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

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

abbreviation *F*

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

abbreviation G
where $G \equiv$ *constant-simulation.map* A B $(B.trg\ t)$

interpretation F: *simulation* A B F
⟨*proof*⟩

interpretation G: *simulation* A B G
⟨*proof*⟩

sublocale *transformation* A B F G *map*
⟨*proof*⟩

end

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

sublocale *transformation* A B F F F
⟨*proof*⟩

sublocale *identity-transformation* A B F F F
⟨*proof*⟩

end

lemma *transformation-eqI*:
assumes *transformation* A B F G σ **and** *transformation* A B F G τ
and *extensional-rts* B
and $\bigwedge a.$ *residuation.ide* A $a \Longrightarrow \sigma\ a = \tau\ a$
shows $\sigma = \tau$
⟨*proof*⟩

lemma *invertible-simulation-cancel-left'*:
assumes *invertible-simulation* A B H
shows ⟦*transformation* C A F G S; *transformation* C A F G T;
$H \circ S = H \circ T$⟧
$\Longrightarrow S = T$
⟨*proof*⟩

lemma *invertible-simulation-cancel-right'*:
assumes *invertible-simulation* A B H
shows ⟦*transformation* B C F G S; *transformation* B C F G T;
$S \circ H = T \circ H$⟧
$\Longrightarrow S = T$

⟨*proof*⟩

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* ⟨*proof*⟩
  sublocale *simulation-to-weakly-extensional-rts A.resid B F* ⟨*proof*⟩

  lemma *fixing-arr-gives-transformation-1*:
  assumes *A1.arr t1*
  shows *transformation A0 B*
        (λ*t0*. *F* (*A1.src t1*, *t0*)) (λ*t0*. *F* (*A1.trg t1*, *t0*))
        (λ*t0*. *F* (*t1*, *t0*))
  ⟨*proof*⟩

  lemma *fixing-arr-gives-transformation-0*:
  assumes *A0.arr t2*
  shows *transformation A1 B*
        (λ*t1*. *F* (*t1*, *A0.src t2*)) (λ*t1*. *F* (*t1*, *A0.trg t2*))
        (λ*t1*. *F* (*t1*, *t2*))
  ⟨*proof*⟩

end

locale *transformation-of-binary-simulations* =
  A1: *weakly-extensional-rts A1* +
  A0: *weakly-extensional-rts A0* +
  B: *weakly-extensional-rts B* +
  A1xA0: *product-rts A1 A0* +
  F: *binary-simulation A1 A0 B F* +
  G: *binary-simulation A1 A0 B G* +
  transformation A1xA0.resid B F G τ
for *A1* :: *'a1 resid* (**infix** $\setminus_{A1}$ *55*)
and *A0* :: *'a0 resid* (**infix** $\setminus_{A0}$ *55*)
and *B* :: *'b resid* (**infix** $\setminus_{B}$ *55*)
and *F* :: *'a1* ∗ *'a0* ⇒ *'b*
and *G* :: *'a1* ∗ *'a0* ⇒ *'b*
and τ :: *'a1* ∗ *'a0* ⇒ *'b*
begin

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

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

notation *B.con* (**infix** $\frown_{B}$ *50*)
notation *B.prfx* (**infix** $\lesssim_{B}$ *50*)
notation *B.cong* (**infix** $\sim_{B}$ *50*)

notation *A1xA0.resid* (**infix** $\setminus_{A1xA0}$ *55*)

sublocale *A1xA0*: *product-of-weakly-extensional-rts A1 A0* ⟨*proof*⟩

lemma *fixing-ide-gives-transformation-1*:
assumes *A1.ide a1*
shows *transformation A0 B* (λ*f0*. *F* (*a1*, *f0*)) (λ*f0*. *G* (*a1*, *f0*))
(λ*f0*. τ (*a1*, *f0*))
⟨*proof*⟩

lemma *fixing-ide-gives-transformation-0*:
assumes *A0.ide a0*
shows *transformation A1 B* (λ*f1*. *F* (*f1*, *a0*)) (λ*f1*. *G* (*f1*, *a0*))
(λ*f1*. τ (*f1*, *a0*))
⟨*proof*⟩

end

1.4 Horizontal Composite of Transformations

lemma *transformation-whisker-left*:
assumes *transformation A B F G* τ **and** *simulation B C H*
and *weakly-extensional-rts C*
shows *transformation A C* (*H* ∘ *F*) (*H* ∘ *G*) (*H* ∘ τ)
⟨*proof*⟩

lemma *transformation-whisker-right*:
assumes *transformation B C F G* τ **and** *simulation A B H*
and *weakly-extensional-rts A*
shows *transformation A C* (*F* ∘ *H*) (*G* ∘ *H*) (τ ∘ *H*)
⟨*proof*⟩

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

lemma *horizontal-composite*:
assumes *transformation B C F G* σ **and** *transformation A B H K* τ
and *extensional-rts A* **and** *extensional-rts B* **and** *extensional-rts C*
shows *transformation A C* (*F* ∘ *H*) (*G* ∘ *K*) (σ ∘ τ)
⟨*proof*⟩

end

Chapter 2

RTS's as Categories

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

2.1 Categories with Bounded Pushouts

2.1.1 Bounded Spans

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

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

context *category*
begin

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

lemma *bounded-spanI* [*intro*]:
assumes *commutative-square f g h k*
shows *bounded-span h k*
⟨*proof*⟩

lemma *bounded-spanE* [*elim*]:
assumes *bounded-span h k*
obtains *f g* **where** *commutative-square f g h k*
⟨*proof*⟩

lemma *bounded-span-sym*:
shows *bounded-span h k* $\Longrightarrow$ *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* ≡
commutative-square f g h k ∧
(∀ *f′ g′*. *commutative-square f′ g′ h k* ⟶ (∃!*l*. *l* · *f* = *f′* ∧ *l* · *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 ⋀*f′ g′*. *commutative-square f′ g′ h k* $\Longrightarrow$ ∃!*l*. *l* · *f* = *f′* ∧ *l* · *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 *elementary-category-with-bounded-pushouts* =
category C
for *C* :: *'a comp* (**infixr** · *55*)
and *inj0* :: *'a* ⇒ *'a* ⇒ *'a* (i_0[-, -])
and *inj1* :: *'a* ⇒ *'a* ⇒ *'a* (i_1[-, -]) +
assumes *inj0-ext*: ¬ *bounded-span h k* $\Longrightarrow$ $\mathrm{i}_0[h, k]$ = *null*
and *inj1-ext*: ¬ *bounded-span h k* $\Longrightarrow$ $\mathrm{i}_1[h, k]$ = *null*
and *pushout-commutes* [*intro*]:
bounded-span h k $\Longrightarrow$ *commutative-square* $\mathrm{i}_1[h, k]$ $\mathrm{i}_0[h, k]$ *h k*
and *pushout-universal*:
commutative-square f g h k $\Longrightarrow$ ∃!*l*. *l* · $\mathrm{i}_1[h, k]$ = *f* ∧ *l* · $\mathrm{i}_0[h, k]$ = *g*
begin

lemma *dom-inj* [*simp*]:
assumes *bounded-span h k*
shows *dom* $\mathrm{i}_0[h,\ k]$ = *cod k* **and** *dom* $\mathrm{i}_1[h,\ k]$ = *cod h*
⟨*proof*⟩

lemma *cod-inj*:
assumes *bounded-span h k*
shows *cod* $\mathrm{i}_1[h,\ k]$ = *cod* $\mathrm{i}_0[h,\ k]$
⟨*proof*⟩

lemma *has-bounded-pushouts*:
assumes *bounded-span h k*
shows *pushout-square* $\mathrm{i}_1[h,\ k]$ $\mathrm{i}_0[h,\ k]$ *h k*
⟨*proof*⟩

end

end

2.2 Categories of Transitions

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

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

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

lemma *commutative-square-sym*:
shows *commutative-square f g h k* ⟹ *commutative-square g f k h*
⟨*proof*⟩

lemma *inj-sym*:
shows $\mathrm{i}_0[k,\ h]$ = $\mathrm{i}_1[h,\ k]$
⟨*proof*⟩

In this setting, pushouts are uniquely determined.

lemma *pushouts-unique*:
assumes *pushout-square f g h k*
shows *f* = $\mathrm{i}_1[h,\ k]$ **and** *g* = $\mathrm{i}_0[h,\ k]$
⟨*proof*⟩

lemma *inj-arr-self*:
assumes *arr t*
shows $\mathrm{i}_0[t,\ t] = cod\ t$ **and** $\mathrm{i}_1[t,\ t] = cod\ t$
⟨*proof*⟩

lemma *inj-arr-dom*:
assumes *arr t*
shows $\mathrm{i}_0[t,\ dom\ t] = t$ **and** $\mathrm{i}_1[t,\ dom\ t] = cod\ t$
⟨*proof*⟩

lemma *eq-iff-ide-inj*:
assumes *span t u*
shows $t = u \longleftrightarrow ide\ \mathrm{i}_0[t,\ u] \wedge ide\ \mathrm{i}_0[u,\ t]$
⟨*proof*⟩

lemma *inj-comp*:
assumes *bounded-span t* $(v \cdot u)$
shows $\mathrm{i}_0[t,\ v \cdot u] = \mathrm{i}_0[\mathrm{i}_0[t,\ u],\ v]$
and $\mathrm{i}_0[v \cdot u,\ t] = \mathrm{i}_0[v,\ \mathrm{i}_0[t,\ u]] \cdot \mathrm{i}_0[u,\ t]$
⟨*proof*⟩

lemma *inj-prefix*:
assumes *arr* $(u \cdot t)$
shows $\mathrm{i}_0[u \cdot t,\ t] = u$ **and** $\mathrm{i}_0[t,\ u \cdot t] = cod\ u$
⟨*proof*⟩

end

2.3 Extensional RTS's with Composites as Categories

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

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

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

interpretation *Category.partial-magma* ‹$\lambda u\ t.\ t \cdot u$›
⟨*proof*⟩
interpretation *Category.partial-composition* ‹$\lambda u\ t.\ t \cdot u$› ⟨*proof*⟩

lemma *null-char*:
shows *null* = *A.null*

⟨*proof*⟩

lemma *ide-char*:
shows *ide a* ⟷ *A.ide a*
⟨*proof*⟩

lemma *src-in-domains*:
assumes *A.arr t*
shows *A.src t* ∈ *domains t*
⟨*proof*⟩

lemma *trg-in-codomains*:
assumes *A.arr t*
shows *A.trg t* ∈ *codomains t*
⟨*proof*⟩

lemma *arr-char*:
shows *arr* = *A.arr*
⟨*proof*⟩

lemma *seq-char*:
shows *seq u t* ⟷ *A.seq t u*
⟨*proof*⟩

sublocale *category* ‹λ*u t*. *t* · *u*›
⟨*proof*⟩

proposition *is-category*:
shows *category* (λ*u t*. *t* · *u*)
⟨*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

2.4 Characterization

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

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

sublocale *category-of-transitions* ‹$\lambda u\ t.\ t \cdot u$› ‹$\lambda h\ k.\ h \setminus k$› ‹$\lambda h\ k.\ k \setminus h$›
⟨*proof*⟩

proposition *is-category-of-transitions*:
shows *category-of-transitions* ($\lambda u\ t.\ t \cdot u$) ($\lambda h\ k.\ h \setminus k$) ($\lambda h\ k.\ k \setminus h$)
⟨*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

interpretation *ResiduatedTransitionSystem.partial-magma* ‹$\lambda h\ k.\ \mathrm{i}_0[h, k]$›
⟨*proof*⟩

lemma *null-char*:
shows *null* = *C.null*
⟨*proof*⟩

interpretation *residuation* ‹$\lambda h\ k.\ \mathrm{i}_0[h, k]$›
⟨*proof*⟩

lemma *con-char*:
shows *con t u* ⟷ *C.bounded-span t u*
⟨*proof*⟩

lemma *arr-char*:
shows *arr t* ⟷ *C.arr t*
⟨*proof*⟩

lemma *ide-char*:
shows *ide a* ⟷ *C.ide a*
⟨*proof*⟩

interpretation *rts* ‹$\lambda h\ k.\ \mathrm{i}_0[h, k]$›
⟨*proof*⟩

interpretation *extensional-rts* ‹$\lambda h\ k.\ \mathrm{i}_0[h,\ k]$›
⟨*proof*⟩

lemma *src-char*:
assumes *arr t*
shows *src t* = *C.dom t*
⟨*proof*⟩

lemma *trg-char*:
assumes *arr t*
shows *trg t* = *C.cod t*
⟨*proof*⟩

lemma *seq-char*:
shows *seq t u* ⟷ *C.seq u t*
⟨*proof*⟩

lemma *comp-char*:
shows *comp t u* = *u* ⋅ *t*
⟨*proof*⟩

sublocale *extensional-rts-with-composites* ‹$\lambda h\ k.\ \mathrm{i}_0[h,\ k]$›
⟨*proof*⟩

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

end

end

Chapter 3

RTS Constructions

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

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

3.1 Notation

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

notation involving the symbols at issue.

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

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

3.2 Some Constraints on a Type

3.2.1 Nondegenerate

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

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

3.2.2 Lifting

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

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

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

lemma *inj-some-lift*:
shows *inj some-lift*
⟨*proof*⟩

A type that admits lifting is obviously nondegenerate.

sublocale *nondegenerate*
⟨*proof*⟩

end

3.2.3 Pairing

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

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

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

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

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

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

lemma *inj-some-pair*:
shows *inj some-pair*
⟨*proof*⟩

lemma *first-conv*:
shows *first* (*some-pair* $(x, y)) = x$
⟨*proof*⟩

lemma *second-conv*:
shows *second* (*some-pair* $(x, y)) = y$
⟨*proof*⟩

lemma *pair-conv*:
assumes *is-pair x*
shows *some-pair* (*first* x, *second* $x) = x$
⟨*proof*⟩

end

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

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

sublocale *lifting type*
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *inj-some-inj*:
shows *inj-on some-inj* (*Collect small-function*)
⟨*proof*⟩

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

lemma *app-some-inj*:
assumes *small-function F*
shows *app* (*some-inj F*) = *F*
⟨*proof*⟩

lemma *some-inj-lam-app*:
assumes *f* ∈ *some-inj* ‘ *Collect small-function*
shows *some-inj* (λ*x*. *app f x*) = *f*
⟨*proof*⟩

end

context
begin

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

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

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

end

3.2.5 Universe

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

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

context
begin

interpretation *nondegenerate* ‹*TYPE*(*V*)›
⟨*proof*⟩

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 $\bigwedge t$. *small-function* (*A t*)
⟨*proof*⟩

context *exponentiation*
begin

lemma *small-function-some-inj-resid*:
fixes *A* :: *'a resid*
assumes *small-rts A*
shows *small-function* (λ*t*. *some-inj* (*A t*))
⟨*proof*⟩

fun *some-inj-resid* :: *'a resid* ⇒ *'a*
where *some-inj-resid A* = (*some-inj* (λ*t*. *some-inj* (*A t*)))

lemma *inj-on-some-inj-resid*:
shows *inj-on some-inj-resid* {*A* :: *'a resid*. *small-rts A*}
⟨*proof*⟩

end

3.4 Injective Images of RTS's

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

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

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

abbreviation *Null*
where *Null* ≡ *map A.null*

abbreviation *Arr*
where *Arr t* ≡ *t* ∈ *map* ' *Collect A.arr*

abbreviation *map'*

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

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

lemma *inj-map'*:
shows $inj\text{-}on\ map\ (Collect\ A.arr)$
⟨*proof*⟩

lemma *map-null*:
shows $map\ A.null \notin map\ `\ Collect\ A.arr$
⟨*proof*⟩

lemma *map'-map* [*simp*]:
assumes $A.arr\ t$
shows $map'\ (map\ t) = t$
⟨*proof*⟩

lemma *map-map'* [*simp*]:
assumes $Arr\ t$
shows $map\ (map'\ t) = t$
⟨*proof*⟩

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

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

sublocale *residuation resid*
⟨*proof*⟩

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

lemma *con-char*:
shows $t \frown u \longleftrightarrow Arr\ t \wedge Arr\ u \wedge map'\ t \frown_A map'\ u$
⟨*proof*⟩

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

lemma *ide-char*$_{II}$:
shows $ide\ t \longleftrightarrow Arr\ t \wedge A.ide\ (map'\ t)$
⟨*proof*⟩

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

⟨*proof*⟩

sublocale *rts resid*
⟨*proof*⟩

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

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

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

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

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 map'_{ext} ‘ (*Collect arr*) = *Collect A.arr*
⟨*proof*⟩

lemma $cong\text{-}char_{II}$:
shows $t \sim u \longleftrightarrow Arr\ t \wedge Arr\ u \wedge map'\ t \sim_A map'\ u$
⟨*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* ⟷ *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 $\mathit{inj\text{-}image\text{-}rts.map}_{ext}$ $(G \circ F)$ $X =$
$\mathit{inj\text{-}image\text{-}rts.map}_{ext}$ *G* (*inj-image-rts.resid F X*) ∘
($\mathit{inj\text{-}image\text{-}rts.map}_{ext}$ *F X*)
and $\mathit{inj\text{-}image\text{-}rts.map'}_{ext}$ $(G \circ F)$ $X =$
$\mathit{inj\text{-}image\text{-}rts.map'}_{ext}$ *F X* ∘
$\mathit{inj\text{-}image\text{-}rts.map'}_{ext}$ *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* ⟷ *False*
⟨*proof*⟩

lemma *ide-char*$_{ERTS}$:
shows *ide t* ⟷ *False*
⟨*proof*⟩

lemma *con-char*:
shows *con t u* ⟷ *False*
⟨*proof*⟩

lemma *trg-char*:
shows *trg t* = *null*
⟨*proof*⟩

sublocale *rts resid*
⟨*proof*⟩

lemma *cong-char*$_{ERTS}$:
shows *cong t u* ⟷ *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* ⟷ *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 resid* ⇒ *'a* ⇒ *'e*
where *initiator A* ≡ (λ*t*. *ResiduatedTransitionSystem.partial-magma.null A*)

lemma *initiator-is-simulation*:
assumes *rts A*
shows *simulation resid A* (*initiator A*)
⟨*proof*⟩

lemma *universality*:
assumes *rts A*
shows ∃!*F*. *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* ≡ *SOME t. t* ≠ *undefined*

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

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

  lemma *null-char*:
  shows *null* = *undefined*
    ⟨*proof*⟩

  sublocale *residuation resid*
    ⟨*proof*⟩

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

  lemma *arr-char*:
  shows *arr t* ⟷ *t* = *the-arr*
    ⟨*proof*⟩

  lemma *ide-char*$_{1RTS}$:
  shows *ide t* ⟷ *t* = *the-arr*
    ⟨*proof*⟩

  lemma *con-char*:
  shows *con t u* ⟷ *arr t* ∧ *arr u*
    ⟨*proof*⟩

  lemma *trg-char*:
  shows *trg t* = (*if arr t then t else null*)
    ⟨*proof*⟩

  sublocale *rts resid*
    ⟨*proof*⟩

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

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

lemma *cong-char*$_{1\,RTS}$:
shows $t \lesssim_1 u \longleftrightarrow arr\ t \wedge arr\ u$
⟨*proof*⟩

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

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

lemma *src-char*:
shows *src t* = (*if t* = *the-arr then t else null*)
⟨*proof*⟩

lemma *prfx-char*:
shows $t \lesssim_1 u \longleftrightarrow arr\ t \wedge arr\ u$
⟨*proof*⟩

lemma *composite-of-char*:
shows *composite-of t u v* $\longleftrightarrow arr\ t \wedge arr\ u \wedge arr\ v$
⟨*proof*⟩

lemma *composable-char*:
shows *composable t u* $\longleftrightarrow arr\ t \wedge arr\ u$
⟨*proof*⟩

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

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

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

lemma *terminator-is-simulation*:
assumes *rts A*
shows *simulation A resid* (*terminator A*)
⟨*proof*⟩

lemma *universality*:
assumes *rts A*
shows ∃!*F*. *simulation A 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* ⟷
(∀ *Q R T*. *transformation resid A Q R T* ⟶ *F* ∘ *T* = *G* ∘ *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* ⟷
(∀ *Q R T*. *transformation resid A Q R T* ⟶ *U* ∘ *T* = *V* ∘ *T*)
⟨*proof*⟩

end

3.7 Sub-RTS

A sub-RTS of an RTS R may be determined by specifying a subset of the transitions of R that is closed under residuation and in addition includes some common source for every consistent pair of transitions contained in it.

locale *sub-rts* =

R: *rts R*
for *R* :: *'a resid* (**infix** $\setminus_R$ *70*)
and *Arr* :: *'a* $\Rightarrow$ *bool* +
assumes *inclusion*: *Arr t* $\Longrightarrow$ *R.arr t*
and *resid-closed*: $[\![$*Arr t*; *Arr u*; *R.con t u*$]\!] \Longrightarrow$ *Arr* (*t* $\setminus_R$ *u*)
and *enough-sources*: $[\![$*Arr t*; *Arr u*; *R.con t u*$]\!] \Longrightarrow$
$\exists$ *a*. *Arr a* $\wedge$ *a* $\in$ *R.sources t* $\wedge$ *a* $\in$ *R.sources u*
begin

definition *resid* :: *'a resid* (**infix** $\setminus$ *70*)
where *resid t u* $\equiv$ *if Arr t* $\wedge$ *Arr u* $\wedge$ *R.con t u then t* $\setminus_R$ *u else R.null*

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

lemma *null-char*:
shows *null* = *R.null*
⟨*proof*⟩

sublocale *residuation resid*
⟨*proof*⟩

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

lemma *ide-char*:
shows *ide t* $\longleftrightarrow$ *Arr t* $\wedge$ *R.ide t*
⟨*proof*⟩

lemma *con-char*:
shows *con t u* $\longleftrightarrow$ *Arr t* $\wedge$ *Arr u* $\wedge$ *R.con t u*
⟨*proof*⟩

lemma *trg-char*:
shows *trg* = (λ*t*. *if arr t then R.trg t else null*)
⟨*proof*⟩

sublocale *rts resid*
⟨*proof*⟩

lemma *prfx-char*:
shows *prfx t u* $\longleftrightarrow$ *Arr t* $\wedge$ *Arr u* $\wedge$ *R.prfx t u*
⟨*proof*⟩

lemma *cong-char*:
shows *cong t u* $\longleftrightarrow$ *Arr t* $\wedge$ *Arr u* $\wedge$ *R.cong t u*
⟨*proof*⟩

lemma *composite-of-char*:
shows *composite-of t u v* $\longleftrightarrow$ *Arr t* $\wedge$ *Arr u* $\wedge$ *Arr v* $\wedge$ *R.composite-of t u v*
⟨*proof*⟩

lemma *join-of-char*:
shows *join-of t u v* $\longleftrightarrow$ *Arr t* $\wedge$ *Arr u* $\wedge$ *Arr v* $\wedge$ *R.join-of t u v*
⟨*proof*⟩

lemma *preserves-weakly-extensional-rts*:
assumes *weakly-extensional-rts R*
shows *weakly-extensional-rts resid*
⟨*proof*⟩

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

end

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

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

lemma *src-char*:
shows *src* = (*λt. if arr t then R.src t else null*)
⟨*proof*⟩

lemma *targets-char*:
assumes *arr t*
shows *targets t* = {*R.trg t*}
⟨*proof*⟩

end

locale *sub-rts-of-extensional-rts* =
R: *extensional-rts R* +
sub-rts R Arr
for *R* :: *'a resid* (**infix** $\backslash_R$ *70*)
and *Arr* :: *'a* $\Rightarrow$ *bool*
begin

sublocale *sub-rts-of-weakly-extensional-rts* ⟨*proof*⟩

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

end

3.8 Fibered Product RTS

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

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

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

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

abbreviation *Con*
where *Con* ≡ λ*tu vw*. *fst tu* $\frown_A$ *fst vw* ∧ *snd tu* $\frown_B$ *snd vw* ∧
F (*fst tu*) = *G* (*snd tu*) ∧ *F* (*fst vw*) = *G* (*snd vw*)

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

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

lemma *null-char*:
shows *null* = (*A.null*, *B.null*)
⟨*proof*⟩

sublocale *residuation resid*
⟨*proof*⟩

notation *con* (**infix** ⌢ *50*)

lemma *arr-char*:
shows *arr t* ⟷ *Arr t*
⟨*proof*⟩

lemma *con-char*:
shows *t* ⌢ *u* ⟷ *Con t u*
⟨*proof*⟩

lemma *ide-char*$_{FP}$:
shows *ide t* ⟷ *Ide t*
⟨*proof*⟩

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

sublocale *rts resid*
⟨*proof*⟩

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

lemma *prfx-char*:
shows *t* ≲ *u* ⟷ *F* (*fst t*) = *G* (*snd t*) ∧ *F* (*fst u*) = *G* (*snd u*) ∧
fst t ≲$_A$ *fst u* ∧ *snd t* ≲$_B$ *snd u*
⟨*proof*⟩

lemma *cong-char*$_{FP}$:
shows *t* ∼ *u* ⟷ *F* (*fst t*) = *G* (*snd t*) ∧ *F* (*fst u*) = *G* (*snd u*) ∧
fst t ∼$_A$ *fst u* ∧ *snd t* ∼$_B$ *snd u*
⟨*proof*⟩

lemma *sources-char*:
shows *sources t* =
{*a*. *F* (*fst t*) = *G* (*snd t*) ∧ *F* (*fst a*) = *G* (*snd a*) ∧
fst a ∈ *A.sources* (*fst t*) ∧ *snd a* ∈ *B.sources* (*snd t*)}
⟨*proof*⟩

lemma *targets-char*$_{FP}$:
shows *targets t* =

$\{a.\ F\ (fst\ t) = G\ (snd\ t) \wedge F\ (fst\ a) = G\ (snd\ a) \wedge$
$fst\ a \in A.targets\ (fst\ t) \wedge snd\ a \in B.targets\ (snd\ t)\}$
⟨*proof*⟩

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
⟨*proof*⟩

lemma P_0-*is-simulation*:
shows *simulation resid B* P_0
⟨*proof*⟩

sublocale P_1: *simulation resid A* P_1
⟨*proof*⟩

lemma P_1-*is-simulation*:
shows *simulation resid A* P_1
⟨*proof*⟩

lemma *commutativity*:
shows $F\ o\ P_1 = G\ o\ P_0$
⟨*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 $P_1 \circ tuple\ X\ H\ K = H$ **and** $P_0 \circ tuple\ X\ H\ K = K$
and $\exists!HK.\ simulation\ X\ resid\ HK \wedge P_1\ o\ HK = H \wedge P_0\ o\ HK = K$
⟨*proof*⟩

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

sublocale *weakly-extensional-rts resid*
⟨*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 *fibered-product-of-extensional-rts* =
A: *extensional-rts A* +
B: *extensional-rts B* +
fibered-product-of-weakly-extensional-rts
begin

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

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

end

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

sublocale *small-rts resid*
⟨*proof*⟩

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

end

3.9 Product RTS

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

definition *pointwise-tuple* :: $('x \Rightarrow 'a) \Rightarrow ('x \Rightarrow 'b) \Rightarrow 'x \Rightarrow 'a \times 'b$ $(\langle\langle\text{-}, \text{-}\rangle\rangle)$
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
⟨*proof*⟩

lemma P_0*-is-simulation*:
shows *simulation resid B* P_0
⟨*proof*⟩

sublocale P_1: *simulation resid A* P_1
⟨*proof*⟩

lemma P_1*-is-simulation*:
shows *simulation resid A* P_1
⟨*proof*⟩

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 $\exists!HK.$ *simulation X resid HK* $\wedge$ P_1 *o HK* $= H \wedge P_0$ *o HK* $= K$
⟨*proof*⟩

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*⟩

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* $(P_1 \circ F)$ $(P_1 \circ G)$ *S*
and *transformation X B* $(P_0 \circ F)$ $(P_0 \circ G)$ *T*
shows [*intro*]: *transformation X resid F G* $\langle\langle S,\ T\rangle\rangle$
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 *o ST* $= S \wedge$ P_0 *o 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* = *product-rts.resid A B*
shows *simulation X Y* ⟨⟨*H*, *K*⟩⟩
⟨*proof*⟩

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

lemma *comp-pointwise-tuple*:
shows ⟨⟨*H*, *K*⟩⟩ ∘ *L* = ⟨⟨*H* ∘ *L*, *K* ∘ *L*⟩⟩
⟨*proof*⟩

lemma *comp-product-simulation-tuple2*:
assumes *simulation A A′ F* **and** *simulation B B′ G*
and *transformation X A* H_0 H_1 *H* **and** *transformation X B* K_0 K_1 *K*
shows *product-simulation.map A B F G* ∘ ⟨⟨*H*, *K*⟩⟩ = ⟨⟨*F* ∘ *H*, *G* ∘ *K*⟩⟩
⟨*proof*⟩

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

locale *product-transformation* =
A1: *weakly-extensional-rts A1* +
A0: *weakly-extensional-rts A0* +
B1: *extensional-rts B1* +
B0: *extensional-rts B0* +
A1xA0: *product-rts A1 A0* +
B1xB0: *product-rts B1 B0* +
F1: *simulation A1 B1 F1* +
F0: *simulation A0 B0 F0* +
G1: *simulation A1 B1 G1* +
G0: *simulation A0 B0 G0* +
T1: *transformation A1 B1 F1 G1 T1* +
T0: *transformation A0 B0 F0 G0 T0*
for *A1* :: *'a1 resid* (**infix** $\backslash_{A1}$ *70*)
and *A0* :: *'a0 resid* (**infix** $\backslash_{A0}$ *70*)

and *B1* :: *'b1 resid* (**infix** $\backslash_{B1}$ *70*)
and *B0* :: *'b0 resid* (**infix** $\backslash_{B0}$ *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-of-weakly-extensional-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*) map_0 :: *'a1* × *'a0* ⇒ *'b1* × *'b0*
where $map_0 \equiv (\lambda a.\ (T1\ (fst\ a),\ T0\ (snd\ a)))$

sublocale *TC*: *transformation-by-components*
A1xA0.resid B1xB0.resid F1xF0.map G1xG0.map map_0
⟨*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* (map_0 (*A1xA0.src t*)) (*F1xF0.map t*)
⟨*proof*⟩

lemma *map-simp-ide*:
assumes *A1xA0.ide t*
shows *map t* = map_0 (*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-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-simulation.map A B* $(F' \circ F)$ $(G' \circ G)$ =
product-simulation.map A′ B′ F′ G′ $\circ$ *product-simulation.map A B F G*
⟨*proof*⟩

3.9.1 Associators

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

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

sublocale *AxB*: *product-rts A B* ⟨*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* ≡ *Ax-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-xC.resid Ax-BxC.resid map*
⟨*proof*⟩

lemma *is-simulation*:
shows *simulation AxB-xC.resid Ax-BxC.resid map*
⟨*proof*⟩

The following explicit formula is more convenient for calculations.

lemma *map-eq*:

shows *map* = (λ*x*. *if AxB-xC.arr x*
then (*fst* (*fst x*), (*snd* (*fst x*), *snd x*))
else Ax-BxC.null)
⟨*proof*⟩

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

sublocale *inv*: *simulation Ax-BxC.resid AxB-xC.resid map′*
⟨*proof*⟩

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

lemma *map′-eq*:
shows *map′* =
(λ*x*. *if Ax-BxC.arr x*
then ((*fst x*, *fst* (*snd x*)), *snd* (*snd x*))
else AxB-xC.null)
⟨*proof*⟩

lemma *inverse-simulations-map′-map*:
shows *inverse-simulations AxB-xC.resid Ax-BxC.resid map′ map*
⟨*proof*⟩

end

3.10 Exponential RTS

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

locale *consistent-transformations* =
A: *weakly-extensional-rts A* +
B: *extensional-rts B* +
F: *simulation A B F* +
G: *simulation A B G* +
H: *simulation A B H* +
σ: *transformation A B F G σ* +

```
  τ: transformation A B F H τ
for A :: 'a resid      (infix \_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 B.join (σ (A.src t) \_B τ (A.src t)) (H t)
                 else B.null)

end
```

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

```
context consistent-transformations
begin

  lemma sym-apex-eq:
  shows sym.apex = apex
  ⟨proof⟩
```

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

interpretation *apex*: *simulation A B apex*
⟨*proof*⟩

lemma *simulation-apex*:
shows *simulation A B apex*
⟨*proof*⟩

lemma *resid-ide*:
assumes *A.ide a*
shows *resid a* = σ *a* $\backslash_B$ τ *a*
⟨*proof*⟩

interpretation *resid*: *transformation* ‹($\backslash_A$)› ‹($\backslash_B$)› *H apex resid*
⟨*proof*⟩

lemma *transformation-resid*:
shows *transformation* ($\backslash_A$) ($\backslash_B$) *H apex resid*
⟨*proof*⟩

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

notation *A.con* (**infix** $\frown_A$ *50*)
notation *A.prfx* (**infix** $\lesssim_A$ *50*)
notation *B.con* (**infix** $\frown_B$ *50*)
notation *B.join* (**infix** $\sqcup_B$ *52*)
notation *B.prfx* (**infix** $\lesssim_B$ *50*)

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

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

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

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

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

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

abbreviation *Ide* :: ('*a*, '*b*) *arr* ⇒ *bool*
where *Ide* ≡ λ*τ*. *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* ≡ λ*σ 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* = (λ*t*. *if A.arr t*
then Cod τ t \\$_B$ (*Map σ* (*A.src t*) \\$_B$ *Map τ* (*A.src t*))
else B.null)

lemma *Apex-sym*:
assumes *Con σ τ*
shows *Apex σ τ* = *Apex τ σ*
⟨*proof*⟩

lemma *Apex-is-simulation* [*intro*]:
assumes *Con σ τ*
shows *simulation A B* (*Apex σ τ*)
⟨*proof*⟩

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

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

lemma *Dom-resid'*:
assumes *Con* σ τ
shows *Dom* ($\sigma \setminus \tau$) = *Cod* τ
⟨*proof*⟩

lemma *Cod-resid'*:
assumes *Con* σ τ
shows *Cod* ($\sigma \setminus \tau$) = *Apex* σ τ
⟨*proof*⟩

lemma *Map-resid'*:
assumes *Con* σ τ
shows *Map* ($\sigma \setminus \tau$) = (λt. *if A.arr t*
then Map σ (*A.src t*) $\setminus_B$ *Map* τ (*A.src t*) $\sqcup_B$ *Cod* τ *t*
else B.null)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *transformation-Map-resid*:
assumes *Con* σ τ
shows *transformation* ($\setminus_A$) ($\setminus_B$) (*Cod* τ) (*Apex* σ τ) (*Map* ($\sigma \setminus \tau$))
⟨*proof*⟩

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

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

sublocale *residuation resid*
⟨*proof*⟩

notation *con* (**infix** $\frown$ *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 $Map\ (\sigma \setminus \tau) = (\lambda t.\ if\ A.arr\ t$
$then\ Map\ \sigma\ (A.src\ t) \setminus_B Map\ \tau\ (A.src\ t) \sqcup_B Cod\ \tau\ t$
$else\ B.null)$
⟨*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 $\sigma \setminus \tau =$
$(if\ \sigma \frown \tau$
$then\ MkArr\ (Cod\ \tau)\ (Apex\ \sigma\ \tau)$
$(\lambda t.\ if\ A.arr\ t$
$then\ Map\ \sigma\ (A.src\ t) \setminus_B Map\ \tau\ (A.src\ t) \sqcup_B Cod\ \tau\ t$
$else\ B.null)$
$else\ null)$
⟨*proof*⟩

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

lemma *trg-char*:
shows $trg = (\lambda\tau.\ if\ arr\ \tau\ then\ MkIde\ (Cod\ \tau)\ else\ null)$
⟨*proof*⟩

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

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

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

sublocale *rts resid*
⟨*proof*⟩

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

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

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

lemma $conI_{ERTS}$ [*intro*]:
assumes *coinitial* $\sigma\ \tau$
and $\bigwedge a.\ A.ide\ a \Longrightarrow Map\ \sigma\ a \frown_B Map\ \tau\ a$
shows $\sigma \frown \tau$
⟨*proof*⟩

lemma $conE_{ERTS}$ [*elim*]:
assumes $\sigma \frown \tau$
and $[\![coinitial\ \sigma\ \tau;\ \bigwedge t\ u.\ A.con\ t\ u \Longrightarrow Map\ \sigma\ t \frown_B Map\ \tau\ u]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *arrI* [*intro*]:
assumes $f \neq null$ **and** $transformation\ A\ B\ (Dom\ f)\ (Cod\ f)\ (Map\ f)$
shows $arr\ f$
⟨*proof*⟩

lemma *arrE* [*elim*]:
assumes *arr f*
and $[\![f \neq null;\ transformation\ A\ B\ (Dom\ f)\ (Cod\ f)\ (Map\ f)]\!] \Longrightarrow T$
shows *T*
⟨*proof*⟩

lemma *arr-MkArr* [*iff*]:
shows *arr* (*MkArr F G τ*) ⟷ *transformation A B F G τ*
⟨*proof*⟩

lemma *src-simp*:
assumes *arr τ*
shows *src τ* = *MkIde* (*Dom τ*)
⟨*proof*⟩

lemma *src-char*:
shows *src* = (λτ. *if arr τ then MkIde* (*Dom τ*) *else null*)
⟨*proof*⟩

lemma *Map-src* [*simp*]:
assumes *arr τ*
shows *Map* (*src τ*) = *Dom τ*
⟨*proof*⟩

lemma *ide-MkIde* [*iff*]:
shows *ide* (*MkIde F*) ⟷ *simulation A B F*
⟨*proof*⟩

lemma *MkArr-Map*:
assumes *τ* ≠ *Null*
shows *τ* = *MkArr* (*Dom τ*) (*Cod τ*) (*Map τ*)
⟨*proof*⟩

lemma *MkIde-Dom*:
assumes *arr τ*
shows *MkIde* (*Dom τ*) = *src τ*
⟨*proof*⟩

lemma *MkIde-Cod*:
assumes *arr τ*
shows *MkIde* (*Cod τ*) = *trg τ*
⟨*proof*⟩

lemma *MkIde-Map*:
assumes *ide a*
shows *MkIde* (*Map a*) = *a*
⟨*proof*⟩

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

lemma *seq-char*:
shows *seq* σ τ $\longleftrightarrow$ *Arr* σ $\wedge$ *Arr* τ $\wedge$ *Cod* σ = *Dom* τ
⟨*proof*⟩

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

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

lemma *Map-preserves-prfx*:
assumes $\sigma \lesssim \tau$ **and** *A.arr t*
shows *Map* σ *t* $\lesssim_B$ *Map* τ *t*
⟨*proof*⟩

Joins in an Exponential RTS

notation *join* (**infix** $\sqcup$ *52*)

lemma *join-char*:
shows *joinable* σ τ $\longleftrightarrow$
arr σ $\wedge$ *arr* τ $\wedge$ *Dom* σ = *Dom* τ $\wedge$
($\forall$ *t*. *A.arr t* $\longrightarrow$
B.joinable (*Map* σ (*A.src t*) $\sqcup_B$ *Map* τ (*A.src t*)) (*Dom* σ *t*))
and $\sigma \sqcup \tau$ =
(*if joinable* σ τ
then MkArr (*Dom* τ) (*Apex* σ τ)
(λ*t*. (*Map* σ (*A.src t*) $\sqcup_B$ *Map* τ (*A.src t*)) $\sqcup_B$ *Dom* τ *t*)
else null)
⟨*proof*⟩

lemma *Dom-join*:
assumes *joinable* σ τ
shows *Dom* ($\sigma \sqcup \tau$) = *Dom* σ
⟨*proof*⟩

lemma *Cod-join*:
assumes *joinable* σ τ
shows *Cod* ($\sigma \sqcup \tau$) = *Apex* σ τ
⟨*proof*⟩

lemma *Map-join*:
assumes *joinable* σ τ
shows *Map* $(\sigma \sqcup \tau) =$
$(\lambda t.\ (Map\ \sigma\ (A.src\ t) \sqcup_B Map\ \tau\ (A.src\ t)) \sqcup_B Dom\ \tau\ t)$
⟨*proof*⟩

end

3.10.1 Exponential of Small RTS's

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

lemma *small-Collect-fun*:
shows *small* $\{F.\ F\ `\ Collect\ A.arr \subseteq Collect\ B.arr \wedge$
$F\ `\ (UNIV - Collect\ A.arr) \subseteq \{B.null\}\}$
⟨*proof*⟩

lemma *small-Collect-simulation*:
shows *small* (*Collect* (*simulation A B*))
⟨*proof*⟩

lemma *small-Collect-transformation*:
assumes *simulation A B F* **and** *simulation A B G*
shows *small* (*Collect* (*transformation A B F G*))
⟨*proof*⟩

sublocale *small-rts resid*
⟨*proof*⟩

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

end

3.10.2 Exponential into RTS with Composites

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

interpretation *B*: *extensional-rts B* ⟨*proof*⟩
interpretation *B*: *extensional-rts-with-composites B* ⟨*proof*⟩

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

abbreviation $COMP :: ('a, 'b)\ arr \Rightarrow ('a, 'b)\ arr \Rightarrow ('a, 'b)\ arr$
where $COMP\ t\ u \equiv MkArr\ (Dom\ t)\ (Cod\ u)$
$(\lambda x.\ Map\ t\ (A.src\ x) \cdot_B Map\ u\ (A.src\ x) \sqcup_B Dom\ t\ x)$

lemma *composite-of-iff*:
shows $composite\text{-}of\ t\ u\ v \longleftrightarrow seq\ t\ u \wedge v = COMP\ t\ u$
⟨*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 $arr\ \tau$ **and** $A.arr\ u$
shows $Dom\ \tau\ u \cdot_B Map\ \tau\ (A.trg\ u) = Map\ \tau\ (A.src\ u) \cdot_B Cod\ \tau\ u$
⟨*proof*⟩

lemma *Dom-comp* [*simp*]:
assumes $seq\ \sigma\ \tau$
shows $Dom\ (comp\ \sigma\ \tau) = Dom\ \sigma$
⟨*proof*⟩

lemma *Cod-comp* [*simp*]:
assumes $seq\ \sigma\ \tau$
shows $Cod\ (comp\ \sigma\ \tau) = Cod\ \tau$
⟨*proof*⟩

lemma *Map-comp* [*simp*]:
assumes $seq\ \sigma\ \tau$
shows $Map\ (comp\ \sigma\ \tau) =$
$(\lambda x.\ Map\ \sigma\ (A.src\ x) \cdot_B Map\ \tau\ (A.src\ x) \sqcup_B Dom\ \sigma\ x)$
⟨*proof*⟩

lemma *Map-comp-ide*:
assumes $seq\ \sigma\ \tau$ **and** $A.ide\ x$
shows $Map\ (comp\ \sigma\ \tau)\ x = Map\ \sigma\ x \cdot_B Map\ \tau\ x$
⟨*proof*⟩

end

3.10.3 Exponential by One

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

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

  sublocale *exponential-rts One.resid A* ⟨*proof*⟩
  notation *resid* (**infix** $\backslash$ *70*)
  notation *con* (**infix** $\frown$ *50*)

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

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

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

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

  lemma *inverse-simulations-Dn-Up*:
  shows *inverse-simulations A resid Dn Up*
  ⟨*proof*⟩

end

3.10.4 Evaluation Map

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

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

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

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

lemma *map-simp*:
assumes *ABxA.arr Fg*
shows *map Fg* = *AB.Map* (*fst Fg*) (*snd Fg*)
⟨*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

3.10.5 Currying

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

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

notation *A.con* (**infix** $\frown_A$ *50*)
notation *B.con* (**infix** $\frown_B$ *50*)
notation *C.con* (**infix** $\frown_C$ *50*)
notation *C.prfx* (**infix** $\lesssim_C$ *50*)
notation *C.join* (**infixr** $\sqcup_C$ *52*)
notation *AxB.resid* (**infix** $\setminus_{AxB}$ *55*)
notation *AxB.con* (**infix** $\frown_{AxB}$ *52*)
notation *AxB.prfx* (**infix** $\lesssim_{AxB}$ *52*)
notation *BC.resid* (**infix** $\setminus_{[B,C]}$ *55*)
notation *BC.con* (**infix** $\frown_{[B,C]}$ *52*)
notation *BC.join* (**infixr** $\sqcup_{[B,C]}$ *52*)
notation *BCxB.resid* (**infix** $\setminus_{[B,C]xB}$ *55*)
notation *BCxB.con* (**infix** $\frown_{[B,C]xB}$ *52*)

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

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

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

lemma *Curry-simp*:
assumes *A.arr f*
shows *Curry F G* τ *f* =
BC.MkArr ($\lambda g.$ *F* (*A.src f*, *g*)) ($\lambda g.$ *G* (*A.trg f*, *g*)) ($\lambda g.$ τ (*f*, *g*))
⟨*proof*⟩

lemma *Uncurry-simp*:
assumes *AxB.arr f*
shows *Uncurry* τ *f* = *E.map* (τ (*fst f*), *snd f*)
⟨*proof*⟩

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

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

lemma *Map-Curry*:
assumes *A.arr f*
shows *BC.Map* (*Curry F G τ f*) = (λ*g*. *τ* (*f*, *g*))
⟨*proof*⟩

lemma *Map-simulation-expansion*:
assumes *simulation A BC.resid G* **and** *AxB.arr f*
shows *BC.Map* (*G* (*fst f*)) (*snd f*) =
BC.Map (*G* (*fst f*)) (*B.src* (*snd f*)) ⊔$_C$
BC.Map (*G* (*A.src* (*fst f*))) (*snd f*)
⟨*proof*⟩

lemma *Map-simulation-monotone*:
assumes *simulation A BC.resid G* **and** *f* ≲$_{AxB}$ *g*
shows *BC.Map* (*G* (*fst f*)) (*snd f*) ≲$_C$ *BC.Map* (*G* (*fst g*)) (*snd g*)
⟨*proof*⟩

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

lemma *Uncurry-preserves-simulations* [*intro*]:
assumes *simulation A BC.resid F*
shows *simulation AxB.resid C* (*Uncurry F*)
⟨*proof*⟩

lemma *Curry-preserves-transformations*:
assumes *transformation AxB.resid C F G τ*
shows *transformation A BC.resid* (*Curry3 F*) (*Curry3 G*) (*Curry F G τ*)
⟨*proof*⟩

lemma *Uncurry-preserves-transformations*:
assumes *transformation A BC.resid F G τ*
shows *transformation AxB.resid C* (*Uncurry F*) (*Uncurry G*) (*Uncurry τ*)
⟨*proof*⟩

lemma *Uncurry-Curry*:
assumes *transformation AxB.resid C F G τ*
shows *Uncurry* (*Curry F G τ*) = *τ*
⟨*proof*⟩

lemma *Curry-Uncurry*:
assumes *transformation A BC.resid F G τ*
shows *Curry* (*Uncurry F*) (*Uncurry G*) (*Uncurry τ*) = *τ*
⟨*proof*⟩

lemma *src-Curry*:
assumes *transformation AxB.resid C F G τ* **and** *A.arr f*

shows *BC.src* (*Curry F G τ f*) = *Curry3 F* (*A.src f*)
⟨*proof*⟩

lemma *trg-Curry*:
assumes *transformation AxB.resid C F G τ* **and** *A.arr f*
shows *BC.trg* (*Curry F G τ f*) = *Curry3 G* (*A.trg f*)
⟨*proof*⟩

lemma *src-Uncurry*:
assumes *transformation A BC.resid F G τ* **and** *AxB.arr f*
shows *C.src* (*Uncurry τ f*) = *Uncurry F* (*AxB.src f*)
⟨*proof*⟩

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

end

3.10.6 Currying and Uncurrying as Inverse Simulations

context *Currying*
begin

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

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

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

lemma *Dom-CURRY* [*simp*]:
assumes *AxB-C.arr f*
shows *A-BC.Dom* (*CURRY f*) =
Curry (*AxB-C.Dom f*) (*AxB-C.Dom f*) (*AxB-C.Dom f*)
⟨*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 ¬ *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* $\setminus_{[AxB,C]}$ *u*)) *f*) =
BC.Dom (*A-BC.Map* (*CURRY t* $\setminus_{[A,[B,C]]}$ *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* $\setminus_{[AxB,C]}$ *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* $\setminus_{[AxB,C]}$ *u*)) *f1*) =
BC.Map (*A-BC.Map* (*A-BC.resid* (*CURRY t*) (*CURRY u*)) *f1*)
⟨*proof*⟩

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

⟨*proof*⟩

lemma *Curry-preserves-Apex*:
assumes *AxB-C.con t u*
shows *Curry3* (*AxB-C.Apex t u*) = *A-BC.Apex* (*CURRY t*) (*CURRY u*)
(**is** *?LHS* = *?RHS*)
⟨*proof*⟩

sublocale *CURRY*: *simulation AxB-C.resid A-BC.resid CURRY*
⟨*proof*⟩

lemma *CURRY-is-simulation*:
shows *simulation AxB-C.resid A-BC.resid CURRY*
⟨*proof*⟩

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

lemma *Dom-UNCURRY* [*simp*]:
assumes *A-BC.arr f*
shows *AxB-C.Dom* (*UNCURRY f*) = *Uncurry* (*A-BC.Dom f*)
⟨*proof*⟩

lemma *Cod-UNCURRY* [*simp*]:
assumes *A-BC.arr f*
shows *AxB-C.Cod* (*UNCURRY f*) = *Uncurry* (*A-BC.Cod f*)
⟨*proof*⟩

lemma *Map-UNCURRY* [*simp*]:
assumes *A-BC.arr f*
shows *AxB-C.Map* (*UNCURRY f*) = *Uncurry* (*A-BC.Map f*)
⟨*proof*⟩

lemma *UNCURRY-CURRY* [*simp*]:
assumes *AxB-C.arr t*
shows *UNCURRY* (*CURRY t*) = *t*
⟨*proof*⟩

lemma *CURRY-UNCURRY* [*simp*]:
assumes *A-BC.arr t*
shows *CURRY* (*UNCURRY t*) = *t*
⟨*proof*⟩

lemma *UNCURRY-is-simulation*:

shows *simulation A-BC.resid AxB-C.resid UNCURRY*
⟨*proof*⟩

sublocale *UNCURRY*: *simulation A-BC.resid AxB-C.resid UNCURRY*
⟨*proof*⟩

interpretation *inverse-simulations*
A-BC.resid AxB-C.resid CURRY UNCURRY
⟨*proof*⟩

sublocale *CURRY*: *invertible-simulation AxB-C.resid A-BC.resid CURRY*
⟨*proof*⟩

lemma *invertible-simulation-CURRY*:
shows *invertible-simulation AxB-C.resid A-BC.resid CURRY*
⟨*proof*⟩

sublocale *UNCURRY*: *invertible-simulation*
A-BC.resid AxB-C.resid UNCURRY
⟨*proof*⟩

lemma *invertible-simulation-UNCURRY*:
shows *invertible-simulation A-BC.resid AxB-C.resid UNCURRY*
⟨*proof*⟩

sublocale *inverse-simulations A-BC.resid AxB-C.resid CURRY UNCURRY*
⟨*proof*⟩

lemma *inverse-simulations-CURRY-UNCURRY*:
shows *inverse-simulations A-BC.resid AxB-C.resid CURRY UNCURRY*
⟨*proof*⟩

end

3.10.7 Coextension of a Simulation

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

context *evaluation-map*
begin

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

lemma *Uncurry-simulation-expansion*:
assumes *weakly-extensional-rts X*
and *simulation X AB.resid F*

shows *Currying.Uncurry X A B F* =
map ∘ (*product-simulation.map X A F* (*I A*))
⟨*proof*⟩

lemma *universality*:
assumes *weakly-extensional-rts X*
and *simulation* (*product-rts.resid X A*) *B G*
shows *simulation X AB.resid* (*Currying.Curry3 X A B G*)
and *Currying.Uncurry X A B* (*coext X G*) = *G*
and ∃!*F*. *simulation X AB.resid F* ∧ *Currying.Uncurry X A B F* = *G*
⟨*proof*⟩

lemma *comp-coext-simulation*:
assumes *weakly-extensional-rts X* **and** *weakly-extensional-rts X′*
and *simulation X X′ G*
and *simulation* (*product-rts.resid X′ A*) *B H*
shows *coext X′ H* ∘ *G* = *coext X* (*H* ∘ *product-simulation.map X A G* (*I A*))
⟨*proof*⟩

end

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

lemma *Uncurry-transformation-expansion*:
assumes *weakly-extensional-rts X*
and *transformation X AB.resid F G T*
shows *Currying.Uncurry X A B T* =
map ∘ *product-transformation.map X A AB.resid A F* (*I A*) *T* (*I A*)
⟨*proof*⟩

lemma *universality2*:
assumes *weakly-extensional-rts X*
and *transformation* (*product-rts.resid X A*) *B F G T*
shows *transformation X AB.resid*
(*Currying.Curry3 X A B F*) (*Currying.Curry3 X A B G*)
(*Currying.Curry X A B F G T*)
and *Currying.Uncurry X A B* (*Currying.Curry X A B F G T*) = *T*
and ∃!*T′*. *transformation X AB.resid*
(*Currying.Curry3 X A B F*) (*Currying.Curry3 X A B G*)
T′ ∧
Currying.Uncurry X A B T′ = *T*
⟨*proof*⟩

end

3.10.8 Compositors

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

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

  sublocale *AC*: *exponential-rts A C* ⟨*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* × (*'a*, *'b*) *AB.arr* ⇒ (*'a*, *'c*) *AC.arr*
  where *map* = (λ*F*. *Currying.Curry3 F*)
            (*E-BC.map* ∘ *BCxE-AB.map* ∘ *ASSOC-BC-AB-A.map*)

  sublocale *simulation BCxAB.resid AC.resid map*
    ⟨*proof*⟩

  lemma *is-simulation*:

shows *simulation BCxAB.resid AC.resid map*
⟨*proof*⟩

sublocale *binary-simulation BC.resid AB.resid AC.resid map* ⟨*proof*⟩

lemma *is-binary-simulation*:
shows *binary-simulation BC.resid AB.resid AC.resid map*
⟨*proof*⟩

sublocale *E-BC-o-BCxE-AB*: *composite-simulation*
BC-x-ABxA.resid BCxB.resid C
BCxE-AB.map E-BC.map
⟨*proof*⟩

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* = (λ*gf*. *if BCxAB.arr gf*
then AC.MkArr
(*BC.Dom* (*fst gf*) ∘ *AB.Dom* (*snd gf*))
(*BC.Cod* (*fst gf*) ∘ *AB.Cod* (*snd gf*))
(*BC.Map* (*fst gf*) ∘ *AB.Map* (*snd gf*))
else AC.Null)
⟨*proof*⟩

end

3.10.9 Functoriality of Exponential

Here we show that the covariant and contravariant exponential RTS constructions are "meta-functorial": they preserve identity simulations and compositions of simulations. We say "meta-functorial", rather than "functorial", because we do not have formal categories to serve as the domain and codomain for these constructions.

abbreviation *cov-Exp*
:: *'c resid* ⇒ *'a resid* ⇒ *'b resid* ⇒ (*'a* ⇒ *'b*)
⇒ (*'c*, *'a*) *exponential-rts.arr* ⇒ (*'c*, *'b*) *exponential-rts.arr* (*Exp*→)
where *Exp*→ *X B C* ≡
λ*G t2*. *COMP.map X B C* (*exponential-rts.MkIde G*, *t2*)

abbreviation *cnt-Exp*
:: *'a resid* ⇒ *'b resid* ⇒ *'c resid* ⇒ (*'a* ⇒ *'b*)
⇒ (*'b*, *'c*) *exponential-rts.arr* ⇒ (*'a*, *'c*) *exponential-rts.arr* (*Exp*←)
where *Exp*← *A B X* ≡
λ*F t1*. *COMP.map A B X* (*t1*, *exponential-rts.MkIde F*)

lemma *cov-Exp-eq*:
assumes *extensional-rts X* **and** *extensional-rts B* **and** *extensional-rts C*
shows $Exp^{\rightarrow}\ X\ B\ C\ G =$
$(\lambda t2.\ \textit{if simulation}\ B\ C\ G \wedge \textit{residuation.arr}\ (\textit{exponential-rts.resid}\ X\ B)\ t2$
then exponential-rts.MkArr
$(G \circ \textit{exponential-rts.Dom}\ t2)$
$(G \circ \textit{exponential-rts.Cod}\ t2)$
$(G \circ \textit{exponential-rts.Map}\ t2)$
else exponential-rts.Null)
⟨*proof*⟩

lemma *cnt-Exp-eq*:
assumes *extensional-rts X* **and** *extensional-rts A* **and** *extensional-rts B*
shows $Exp^{\leftarrow}\ A\ B\ X\ F =$
$(\lambda t1.\ \textit{if residuation.arr}\ (\textit{exponential-rts.resid}\ B\ X)\ t1 \wedge \textit{simulation}\ A\ B\ F$
then exponential-rts.MkArr
$(\textit{exponential-rts.Dom}\ t1 \circ F)$
$(\textit{exponential-rts.Cod}\ t1 \circ F)$
$(\textit{exponential-rts.Map}\ t1 \circ F)$
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*)
($Exp^{\rightarrow}\ X\ B\ C\ G$)
⟨*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*)
($Exp^{\leftarrow}\ A\ B\ X\ F$)
⟨*proof*⟩

lemma *cov-Exp-ide*:
assumes *extensional-rts X* **and** *extensional-rts B*
shows $Exp^{\rightarrow}\ X\ B\ B\ (I\ B) = I\ (\textit{exponential-rts.resid}\ X\ B)$
⟨*proof*⟩

lemma *cnt-Exp-ide*:
assumes *extensional-rts X* **and** *extensional-rts B*
shows $Exp^{\leftarrow}\ B\ B\ X\ (I\ B) = I\ (\textit{exponential-rts.resid}\ B\ X)$
⟨*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*)
($Exp^{\leftarrow}\ B\ C\ X\ F$)
⟨*proof*⟩

end

Chapter 4

RTS's in Categories

4.1 RTS-Categories

In this section, we develop the notion of an *RTS-category*, which is analogous to a 2-category, except that the "vertical" structure is that of an RTS, rather than a category. So an RTS-category is a category with respect to a "horizontal" composition, which also has a vertical structure as an RTS.

theory *RTSCategory*
imports *Main RTSConstructions Category3.ConcreteCategory*
Category3.CartesianClosedCategory Category3.EquivalenceOfCategories
begin

4.1.1 Definition and Basic Properties

locale *rts-category* =
V: *extensional-rts resid* +
H: *category hcomp* +
VV: *fibered-product-rts resid resid resid H.dom H.cod* +
H: *simulation VV.resid resid*
‹λ*t*. *if VV.arr t then hcomp* (*fst t*) (*snd t*) *else V.null*›
for *resid* :: ′*a resid* (**infix** \ 70)
and *hcomp* :: ′*a comp* (**infixr** ⋆ 53) +
assumes *null-coincidence* [*simp*]: *H.null* = *V.null*
and *arr-coincidence* [*simp*]: *H.arr* = *V.arr*
and *src-dom* [*simp*]: *V.src* (*H.dom t*) = *H.dom t*
begin

notation *H.in-hom* («- : - → -»)

abbreviation *null*
where *null* ≡ *V.null*

abbreviation *arr*
where *arr* ≡ *V.arr*

abbreviation *src*
where $src \equiv V.src$

abbreviation *trg*
where $trg \equiv V.trg$

abbreviation *dom*
where $dom \equiv H.dom$

abbreviation *cod*
where $cod \equiv H.cod$

We refer to the identities for the horizontal composition as *objects*.

abbreviation *obj*
where $obj \equiv H.ide$

We refer to the identities for the vertical residuation as *states*.

abbreviation *sta*
where $sta \equiv V.ide$

interpretation VV: *fibered-product-of-extensional-rts resid resid resid dom cod*
⟨*proof*⟩
interpretation H: *simulation-between-extensional-rts* $VV.resid$ *resid*
$\langle\lambda t.\ if\ VV.arr\ t\ then\ fst\ t \star snd\ t\ else\ null\rangle$
⟨*proof*⟩

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

lemma *obj-implies-sta*:
assumes *obj* a
shows *sta* a
⟨*proof*⟩

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

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

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

lemma *dom-src* [*simp*]:
shows *dom* (*src t*) = *dom t*
⟨*proof*⟩

lemma *dom-trg* [*simp*]:
shows *dom* (*trg t*) = *dom t*
⟨*proof*⟩

lemma *cod-src* [*simp*]:
shows *cod* (*src t*) = *cod t*
⟨*proof*⟩

lemma *cod-trg* [*simp*]:
shows *cod* (*trg t*) = *cod t*
⟨*proof*⟩

lemma *arr-hcomp* [*intro*]:
assumes *H.seq t u*
shows *arr* (*t* ⋆ *u*)
⟨*proof*⟩

lemma *sta-hcomp* [*intro*]:
assumes *H.seq t u* **and** *sta t* **and** *sta u*
shows *sta* (*t* ⋆ *u*)
⟨*proof*⟩

lemma *src-hcomp* [*simp*]:
assumes *H.seq t u*
shows *src* (*t* ⋆ *u*) = *src t* ⋆ *src u*
⟨*proof*⟩

lemma *trg-hcomp* [*simp*]:
assumes *H.seq t u*
shows *trg* (*t* ⋆ *u*) = *trg t* ⋆ *trg u*
⟨*proof*⟩

lemma *con-implies-hpar*:
assumes *t* ⌢ *u*
shows *H.par t u*
⟨*proof*⟩

lemma *hpar-arr-resid*:
assumes *t* ⌢ *u*
shows *H.par t* (*t* \ *u*)
⟨*proof*⟩

lemma *dom-resid* [*simp*]:
assumes *t* ⌢ *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*:
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* ≡ *sub-rts.resid resid* (λ*t*. «*t* : *a* → *b*»)

lemma *sub-rts-HOM*:
shows *sub-rts resid* (λ*t*. «*t* : *a* → *b*»)
⟨*proof*⟩

lemma *extensional-rts-HOM*:
assumes *obj a* **and** *obj b*
shows *HOM a b* ∈ *Collect extensional-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-HOM* (*HOM*$^{\rightarrow}$)
where *HOM*$^{\rightarrow}$ *a t* ≡
(λ*x*. *if residuation.arr* (*HOM a* (*dom t*)) *x then t* ⋆ *x else null*)

lemma *simulation-cov-HOM-sta*:
assumes *obj a* **and** *sta f*
shows *simulation* (*HOM a* (*dom f*)) (*HOM a* (*cod f*)) ($HOM^{\rightarrow}$ *a f*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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 $HOM^{\rightarrow}\ a\ (t \star u) = HOM^{\rightarrow}\ a\ t \circ HOM^{\rightarrow}\ a\ u$
⟨*proof*⟩

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

lemma *cov-HOM-resid*:
assumes *obj a* **and** *V.con t u*
shows *cov-HOM a* (*t* \ *u*) =
consistent-transformations.resid
(*HOM a* (*dom t*)) (*HOM a* (*cod t*)) ($HOM^{\rightarrow}$ *a* (*trg u*))
($HOM^{\rightarrow}$ *a t*) ($HOM^{\rightarrow}$ *a u*)
⟨*proof*⟩

We can dualize the above, to define, given an object c and an arrow t, a contravariant mapping $HOM^{\leftarrow}\ c\ t$ from *HOM* (*H.cod t*) *c* to *HOM* (*H.dom t*) *c*. I have not carried out a full development parallel to the covariant case, because the contravariant version is not used in an essential way in this article.

abbreviation *cnt-HOM* ($HOM^{\leftarrow}$)
where $HOM^{\leftarrow}\ c\ t \equiv$
(λx. *if residuation.arr* (*HOM* (*cod t*) *c*) *x then x* ⋆ *t else null*)

lemma *simulation-cnt-HOM-sta*:
assumes *sta f* **and** «*f* : *a* → *b*» **and** *obj c*
shows *simulation* (*HOM b c*) (*HOM a c*) ($HOM^{\leftarrow}$ *c f*)
⟨*proof*⟩

lemma *HOM-preserves-isomorphic-left*:
assumes *H.isomorphic a b* **and** *obj c*
shows *isomorphic-rts* (*HOM a c*) (*HOM b c*)
⟨*proof*⟩

end

4.1.3 Additional Notions

An RTS-category is *locally small* if each of the hom-RTS's is a small RTS.

locale *locally-small-rts-category* =
  rts-category +
assumes *small-homs*: $[\![obj\ a;\ obj\ b]\!] \Longrightarrow small\ (H.hom\ a\ b)$
begin

  lemma *HOM-is-small-extensional-rts*:
  assumes *obj a* **and** *obj b*
  shows *HOM a b* $\in$ *Collect extensional-rts* $\cap$ *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* $resid_A$ $comp_A$ +
  B: *rts-category* $resid_B$ $comp_B$ +
  functor $comp_A$ $comp_B$ *F* +
  simulation $resid_A$ $resid_B$ *F*
for $resid_A$ :: $'a\ resid$ (**infix** $\backslash_A$ *70*)
and $comp_A$ :: $'a\ comp$ (**infixr** $\star_A$ *53*)
and $resid_B$ :: $'b\ resid$ (**infix** $\backslash_B$ *70*)
and $comp_B$ :: $'b\ comp$ (**infixr** $\star_B$ *53*)
and *F* :: $'a \Rightarrow 'b$
begin

  notation $A.V.con$ (**infix** $\frown_A$ *50*)
  notation $B.V.con$ (**infix** $\frown_B$ *50*)

  lemma *is-invertible-simulation-if*:
  assumes *invertible-functor* $comp_A$ $comp_B$ *F*
  and $\bigwedge t\ u.\ F\ t \frown_B F\ u \Longrightarrow t \frown_A u$
  shows *invertible-simulation* $resid_A$ $resid_B$ *F*
  ⟨*proof*⟩

  lemma *is-invertible-if*:
  assumes *invertible-simulation* $resid_A$ $resid_B$ *F*
  shows *invertible-functor* $comp_A$ $comp_B$ *F*
  ⟨*proof*⟩

end

locale *rts-category-isomorphism* =

```
    rts-functor +
    invertible-simulation resid_A resid_B F
  begin

    sublocale invertible-functor comp_A comp_B 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 ∈ Obj; B ∈ Obj; C ∈ Obj; D ∈ Obj;
                    residuation.arr (Hom C D) t; residuation.arr (Hom B C) u;
                    residuation.arr (Hom A B) v ⟧ ⟹
                     Comp A B D (Comp B C D t u) v =
```

Comp A C D t (Comp A B C u v)

begin

datatype ('o, 'a) arr =
  Null
| *MkArr 'o 'o 'a*

fun *Dom* :: ('O, 'A) *arr* ⇒ 'O
where *Dom* (*MkArr a - -*) = *a*
  | *Dom - = undefined*

fun *Cod* :: ('O, 'A) *arr* ⇒ 'O
where *Cod* (*MkArr - b -*) = *b*
  | *Cod - = undefined*

fun *Trn* :: ('O, 'A) *arr* ⇒ 'A
where *Trn* (*MkArr - - t*) = *t*
  | *Trn - = undefined*

abbreviation *Arr* :: ('O, 'A) *arr* ⇒ *bool*
where *Arr* ≡ λ*t*. *t* ≠ *Null* ∧ *Dom t* ∈ *Obj* ∧ *Cod t* ∈ *Obj* ∧
    residuation.arr (*Hom* (*Dom t*) (*Cod t*)) (*Trn t*)

abbreviation *Ide* :: ('O, 'A) *arr* ⇒ *bool*
where *Ide* ≡ λ*t*. *t* ≠ *Null* ∧ *Dom t* ∈ *Obj* ∧ *Cod t* ∈ *Obj* ∧
    residuation.ide (*Hom* (*Dom t*) (*Cod t*)) (*Trn t*)

abbreviation *Con* :: ('O, 'A) *arr* ⇒ ('O, 'A) *arr* ⇒ *bool*
where *Con t u* ≡ *Arr t* ∧ *Arr u* ∧ *Dom t* = *Dom u* ∧ *Cod t* = *Cod u* ∧
    residuation.con (*Hom* (*Dom t*) (*Cod t*)) (*Trn t*) (*Trn u*)

fun *resid* (**infix** \ 70)
where *resid Null u* = *Null*
  | *resid t Null* = *Null*
  | *resid t u* =
    (*if Con t u*
     then MkArr (*Dom t*) (*Cod t*) (*Hom* (*Dom t*) (*Cod t*) (*Trn t*) (*Trn u*))
     else Null)

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

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

sublocale *V*: *residuation resid*
⟨*proof*⟩

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

lemma *con-char*:
shows $t \frown u \longleftrightarrow Con\ t\ u$
⟨*proof*⟩

lemma *arr-char*:
shows $V.arr\ t \longleftrightarrow Arr\ t$
⟨*proof*⟩

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 $V.arr\ t$
⟨*proof*⟩

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

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

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 $V.ide\ t$
⟨*proof*⟩

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

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

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

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*⟩

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

sublocale *rts resid*
⟨*proof*⟩

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 $a \in Obj$ **and** $b \in Obj$
and *residuation.arr* $(Hom\ a\ b)\ t$
shows $V.arr\ (MkArr\ a\ b\ t)$
⟨*proof*⟩

lemma *arr-eqI*:
assumes $t \neq Null$ **and** $u \neq Null$
and $Dom\ t = Dom\ u$ **and** $Cod\ t = Cod\ u$ **and** $Trn\ t = Trn\ u$
shows $t = u$
⟨*proof*⟩

lemma *MkArr-Trn*:
assumes $V.arr\ t$

shows *MkArr* (*Dom t*) (*Cod t*) (*Trn t*) = *t*
⟨*proof*⟩

lemma *src-char*:
shows *V.src t* = (*if V.arr t*
then MkArr (*Dom t*) (*Cod t*)
(*weakly-extensional-rts.src* (*Hom* (*Dom t*) (*Cod t*)) (*Trn t*))
else Null)
⟨*proof*⟩

definition *hcomp* (**infixr** ⋆ *53*)
where *t* ⋆ *u* ≡ *if V.arr t* ∧ *V.arr u* ∧ *Dom t* = *Cod u*
then MkArr (*Dom u*) (*Cod t*)
(*Comp* (*Dom u*) (*Cod u*) (*Cod t*) (*Trn t*) (*Trn u*))
else V.null

lemma *arr-hcomp*$_{CRC}$:
assumes *V.arr t* **and** *V.arr u* **and** *Dom t* = *Cod u*
shows *V.arr* (*t* ⋆ *u*)
⟨*proof*⟩

lemma *Dom-hcomp* [*simp*]:
assumes *V.arr t* **and** *V.arr u* **and** *Dom t* = *Cod u*
shows *Dom* (*t* ⋆ *u*) = *Dom u*
⟨*proof*⟩

lemma *Cod-hcomp* [*simp*]:
assumes *V.arr t* **and** *V.arr u* **and** *Dom t* = *Cod u*
shows *Cod* (*t* ⋆ *u*) = *Cod t*
⟨*proof*⟩

lemma *Trn-hcomp* [*simp*]:
assumes *V.arr t* **and** *V.arr u* **and** *Dom t* = *Cod u*
shows *Trn* (*t* ⋆ *u*) = *Comp* (*Dom u*) (*Cod u*) (*Cod t*) (*Trn t*) (*Trn u*)
⟨*proof*⟩

lemma *hcomp-Null* [*simp*]:
shows *t* ⋆ *Null* = *Null* **and** *Null* ⋆ *u* = *Null*
⟨*proof*⟩

sublocale *H*: *Category.partial-magma hcomp*
⟨*proof*⟩

lemma *null-coincidence*$_{CRC}$ [*simp*]:
shows *H.null* = *V.null*
⟨*proof*⟩

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

lemma *H-composable-char*:
shows $t \star u \neq V.null \longleftrightarrow V.arr\ t \wedge V.arr\ u \wedge Dom\ t = Cod\ u$
⟨*proof*⟩

lemma *objI* [*intro*]:
assumes $t \neq V.null$
and $Dom\ t \in Obj$ **and** $Cod\ t = Dom\ t$ **and** $Trn\ t = Id\ (Dom\ t)$
shows $H.ide\ t$
⟨*proof*⟩

lemma *objE* [*elim*]:
assumes $H.ide\ a$
and $[\![a \neq V.null;\ Dom\ a \in Obj;\ Cod\ a = Dom\ a;\ Trn\ a = Id\ (Dom\ a)]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

definition *mkobj*
where $mkobj\ A \equiv MkArr\ A\ A\ (Id\ A)$

lemma *mkobj-simps* [*simp*]:
shows $Dom\ (mkobj\ A) = A$ **and** $Cod\ (mkobj\ A) = A$
and $Trn\ (mkobj\ A) = Id\ A$
⟨*proof*⟩

lemma *obj-mkobj*:
assumes $A \in Obj$
shows $H.ide\ (mkobj\ A)$
⟨*proof*⟩

lemma *obj-char*:
shows $H.ide\ a \longleftrightarrow V.arr\ a \wedge mkobj\ (Dom\ a) = a$
⟨*proof*⟩

lemma *obj-is-sta*:
assumes $H.ide\ a$
shows $V.ide\ a$
⟨*proof*⟩

lemma *obj-simps*:
assumes $H.ide\ a$
shows $Cod\ a = Dom\ a$ **and** $Trn\ a = Id\ (Dom\ a)$
⟨*proof*⟩

lemma *domains-char*:
shows $H.domains\ t = \{a.\ V.arr\ t \wedge mkobj\ (Dom\ t) = a\}$
⟨*proof*⟩

lemma *codomains-char*:
shows $H.codomains\ t = \{a.\ V.arr\ t \wedge mkobj\ (Cod\ t) = a\}$

⟨*proof*⟩

lemma *H-arr-char*:
shows $H.arr\ t \longleftrightarrow t \neq Null \wedge Dom\ t \in Obj \wedge Cod\ t \in Obj \wedge$
$residuation.arr\ (Hom\ (Dom\ t)\ (Cod\ t))\ (Trn\ t)$
⟨*proof*⟩

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 $H.arr\ t$
⟨*proof*⟩

lemma *H-seq-char*:
shows $H.seq\ t\ u \longleftrightarrow V.arr\ t \wedge V.arr\ u \wedge Dom\ t = Cod\ u$
⟨*proof*⟩

lemma *H-seqI* [*intro*]:
assumes $V.arr\ t$ **and** $V.arr\ u$ **and** $Dom\ t = Cod\ u$
shows $H.seq\ t\ u$
⟨*proof*⟩

sublocale H: *category hcomp*
⟨*proof*⟩

lemma *is-category*:
shows *category hcomp*
⟨*proof*⟩

lemma $arr\text{-}coincidence_{CRC}$ [*simp*]:
shows $H.arr = V.arr$
⟨*proof*⟩

lemma *dom-char*:
shows $H.dom = (\lambda t.\ if\ H.arr\ t\ then\ mkobj\ (Dom\ t)\ else\ V.null)$
⟨*proof*⟩

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

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

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

⟨*proof*⟩

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

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

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

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

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

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

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: *fibered-product-rts resid resid resid H.dom H.cod* ⟨*proof*⟩

sublocale H_{VV}: *simulation VV.resid resid*
‹λ*t. if VV.arr t then fst t ⋆ snd t else V.null*›
⟨*proof*⟩

lemma *simulation-hcomp*:
shows *simulation VV.resid resid* (λ*t. if VV.arr t then fst t ⋆ 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*$_{CRC}$ [*simp*]:
shows *H.dom* (*V.src t*) = *H.dom t*
⟨*proof*⟩

lemma *dom-trg*$_{CRC}$ [*simp*]:
shows *H.dom* (*V.trg t*) = *H.dom t*
⟨*proof*⟩

lemma *cod-src*$_{CRC}$ [*simp*]:
shows *H.cod* (*V.src t*) = *H.cod t*
⟨*proof*⟩

lemma *cod-trg*$_{CRC}$ [*simp*]:
shows *H.cod* (*V.trg t*) = *H.cod t*
⟨*proof*⟩

lemma *src-dom*$_{CRC}$ [*simp*]:
shows *V.src* (*H.dom t*) = *H.dom t*

⟨*proof*⟩

lemma *trg-dom*$_{CRC}$ [*simp*]:
shows *V.trg* (*H.dom t*) = *H.dom t*
⟨*proof*⟩

lemma *src-cod*$_{CRC}$ [*simp*]:
shows *V.src* (*H.cod t*) = *H.cod t*
⟨*proof*⟩

lemma *trg-cod*$_{CRC}$ [*simp*]:
shows *V.trg* (*H.cod t*) = *H.cod t*
⟨*proof*⟩

sublocale *rts-category resid hcomp*
⟨*proof*⟩

proposition *is-rts-category*:
shows *rts-category resid hcomp*
⟨*proof*⟩

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

lemma *bij-mkobj*:
shows *mkobj* ∈ *Obj* → *Collect obj*
and *Dom* ∈ *Collect obj* → *Obj*
and ⋀*A*. *Dom* (*mkobj A*) = *A*
and ⋀*a*. *obj a* ⟹ *mkobj* (*Dom a*) = *a*
and *bij-betw mkobj Obj* (*Collect obj*)
and *bij-betw Dom* (*Collect obj*) *Obj*
⟨*proof*⟩

abbreviation *MkArr*$_{ext}$
where *MkArr*$_{ext}$ *A B* ≡
λ*t*. *if residuation.arr* (*Hom A B*) *t then MkArr A B t else Null*

abbreviation *Trn*$_{ext}$
where *Trn*$_{ext}$ *a b* ≡
λ*t*. *if residuation.arr* (*HOM a b*) *t then Trn t*
else ResiduatedTransitionSystem.partial-magma.null
(*Hom* (*Dom a*) (*Dom b*))

lemma *inverse-simulations-Trn-MkArr*:
assumes *A* ∈ *Obj* **and** *B* ∈ *Obj*
shows *inverse-simulations* (*Hom A B*) (*HOM* (*mkobj A*) (*mkobj B*))
(*Trn*$_{ext}$ (*mkobj A*) (*mkobj B*)) (*MkArr*$_{ext}$ *A B*)
⟨*proof*⟩

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

lemma *isomorphic-rts-Hom-HOM*:
assumes $A \in Obj$ **and** $B \in Obj$
shows *isomorphic-rts* (*Hom A B*) (*HOM* (*mkobj A*) (*mkobj B*))
⟨*proof*⟩

end

end

4.3 The RTS-Category of RTS's and Transformations

theory *RTSCatx*
imports *Main ConcreteRTSCategory*
begin

In this section we apply the *concrete-rts-category* construction to create an RTS-category, taking the set of all small extensional RTS's at a given arrow type as the objects and the exponential RTS's formed from these as the hom's, so that the arrows correspond to transformations and the arrows that are identities with respect to the residuation correspond to simulations. We prove that the resulting category, which we will refer to in informal text as $\mathbf{RTS}^\dagger$, is cartesian closed. For that to hold, we need to start with the assumption that the underlying arrow type is a universe.

locale *rtscatx* =
universe arr-type
for *arr-type* :: *'A itself*
begin

sublocale *concrete-rts-category*
‹*TYPE*('*A resid*)› ‹*TYPE*(('*A*, '*A*) *exponential-rts.arr*)›
‹*Collect extensional-rts* ∩ *Collect small-rts*›
‹λ*A B. exponential-rts.resid A B*›
‹λ*A. exponential-rts.MkArr* (*I A*) (*I A*) (*I A*)›
‹λ*A B C f g. COMP.map A B C* (*f*, *g*)›
⟨*proof*⟩

type-synonym *'a arr* =
('*a resid*, ('*a*, '*a*) *exponential-rts.arr*) *concrete-rts-category.arr*

notation *resid* (**infix** \ *70*)
notation *hcomp* (**infixr** ⋆ *53*)

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

lemma *inj-Trn*:
assumes *obj a* **and** *obj b*

shows *Trn* ∈ *H.hom a b* →
Collect (*residuation.arr* (*exponential-rts.resid* (*Dom a*) (*Dom b*)))
and *inj-on Trn* (*H.hom a b*)
⟨*proof*⟩

sublocale *locally-small-rts-category resid hcomp*
⟨*proof*⟩

abbreviation *sta-in-hom* («- : - →$_{sta}$ -»)
where *sta-in-hom f a b* ≡ *H.in-hom f a b* ∧ *sta f*

abbreviation *trn-to* («- : - ⇒ -»)
where *trn-to t f g* ≡ *arr t* ∧ *src t* = *f* ∧ *trg t* = *g*

definition *mkarr* :: *'A resid* ⇒ *'A resid* ⇒
('*A* ⇒ '*A*) ⇒ ('*A* ⇒ '*A*) ⇒ ('*A* ⇒ '*A*) ⇒
'*A arr*
where *mkarr A B F G τ* ≡
MkArr A B (*exponential-rts.MkArr F G τ*)

abbreviation *mksta*
where *mksta A B F* ≡ *mkarr A B F F F*

lemma *Dom-mkarr* [*simp*]:
shows *Dom* (*mkarr A B F G τ*) = *A*
⟨*proof*⟩

lemma *Cod-mkarr* [*simp*]:
shows *Cod* (*mkarr A B F G τ*) = *B*
⟨*proof*⟩

lemma *arr-mkarr* [*intro*]:
assumes *small-rts A* **and** *extensional-rts A*
and *small-rts B* **and** *extensional-rts B*
and *transformation A B F G τ*
shows *arr* (*mkarr A B F G τ*)
and *src* (*mkarr A B F G τ*) = *mksta A B F*
and *trg* (*mkarr A B F G τ*) = *mksta A B G*
and *dom* (*mkarr A B F G τ*) = *mkobj A*
and *cod* (*mkarr A B F G τ*) = *mkobj B*
⟨*proof*⟩

lemma *mkarr-simps* [*simp*]:
assumes *arr* (*mkarr A B F G σ*)
shows *dom* (*mkarr A B F G σ*) = *mkobj A*
and *cod* (*mkarr A B F G σ*) = *mkobj B*
and *src* (*mkarr A B F G σ*) = *mksta A B F*
and *trg* (*mkarr A B F G σ*) = *mksta A B G*
⟨*proof*⟩

lemma *sta-mksta* [*intro*]:
assumes *small-rts A* **and** *extensional-rts A*
and *small-rts B* **and** *extensional-rts B*
and *simulation A B F*
shows *sta* (*mksta A B F*)
and *dom* (*mksta A B F*) = *mkobj A* **and** *cod* (*mksta A B F*) = *mkobj B*
⟨*proof*⟩

abbreviation *Src*
where *Src* ≡ *exponential-rts.Dom* ∘ *Trn*

abbreviation *Trg*
where *Trg* ≡ *exponential-rts.Cod* ∘ *Trn*

abbreviation *Map*
where *Map* ≡ *exponential-rts.Map* ∘ *Trn*

lemma *Src-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F G σ*)
shows *Src* (*mkarr A B F G σ*) = *F*
⟨*proof*⟩

lemma *Trg-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F G σ*)
shows *Trg* (*mkarr A B F G σ*) = *G*
⟨*proof*⟩

lemma *Map-simps* [*simp*]:
assumes *arr t*
shows *Map* (*dom t*) = *I* (*Dom t*)
and *Map* (*cod t*) = *I* (*Cod t*)
and *Map* (*src t*) = *Src t*
and *Map* (*trg t*) = *Trg t*
⟨*proof*⟩

lemma *src-simp*:
assumes *arr t*
shows *src t* = *mksta* (*Dom t*) (*Cod t*) (*Src t*)
⟨*proof*⟩

lemma *trg-simp*:
assumes *arr t*
shows *trg t* = *mksta* (*Dom t*) (*Cod t*) (*Trg t*)
⟨*proof*⟩

The mapping *Map* that takes a transition to its underlying transformation, is a bijection, which cuts down to a bijection between states and simulations.

lemma *bij-mkarr*:
assumes *small-rts A* **and** *extensional-rts A*
and *small-rts B* **and** *extensional-rts B*
and *simulation A B F* **and** *simulation A B G*
shows *mkarr A B F G* ∈ *Collect* (*transformation A B F G*)
→ {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
and *Map* ∈ {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
→ *Collect* (*transformation A B F G*)
and [*simp*]: *Map* (*mkarr A B F G τ*) = *τ*
and [*simp*]: *t* ∈ {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
⟹ *mkarr A B F G* (*Map t*) = *t*
and *bij-betw* (*mkarr A B F G*) (*Collect* (*transformation A B F G*))
{*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
and *bij-betw Map* {*t*. «*t* : *mksta A B F* ⇒ *mksta A B G*»}
(*Collect* (*transformation A B F G*))
⟨*proof*⟩

lemma *bij-mksta*:
assumes *small-rts A* **and** *extensional-rts A*
and *small-rts B* **and** *extensional-rts B*
shows *mksta A B* ∈ *Collect* (*simulation A B*)
→ {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
and *Map* ∈ {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
→ *Collect* (*simulation A B*)
and [*simp*]: *Map* (*mksta A B F*) = *F*
and [*simp*]: *t* ∈ {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
⟹ *mksta A B* (*Map t*) = *t*
and *bij-betw* (*mksta A B*) (*Collect* (*simulation A B*))
{*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
and *bij-betw Map* {*t*. «*t* : *mkobj A* →$_{sta}$ *mkobj B*»}
(*Collect* (*simulation A B*))
⟨*proof*⟩

lemma *mkarr-comp*:
assumes *small-rts A* **and** *extensional-rts A*
and *small-rts B* **and** *extensional-rts B*
and *small-rts C* **and** *extensional-rts C*
and *transformation A B F G σ*
and *transformation B C H K τ*
shows *mkarr A C* (*H* ∘ *F*) (*K* ∘ *G*) (*τ* ∘ *σ*) =
mkarr B C H K τ ⋆ *mkarr A B F G σ*
⟨*proof*⟩

lemma *mkarr-resid*:
assumes *small-rts A* ∧ *extensional-rts A*
and *small-rts B* ∧ *extensional-rts B*
and *consistent-transformations A B F G H σ τ*
shows *mkarr A B F G σ* ⌢ *mkarr A B F H τ*
and *mkarr A B H* (*consistent-transformations.apex A B H σ τ*)

(*consistent-transformations.resid A B H σ τ*) =
mkarr A B F G σ \ *mkarr A B F H τ*
⟨*proof*⟩

lemma *Map-hcomp*:
assumes *H.seq t u*
shows *Map* (*t* ⋆ *u*) = *Map t* ∘ *Map u*
⟨*proof*⟩

lemma *Map-resid*:
assumes *V.con t u*
shows *consistent-transformations* (*Dom t*) (*Cod t*)
(*Src t*) (*Trg t*) (*Trg u*) (*Map t*) (*Map u*)
and *Map* (*t* \ *u*) =
consistent-transformations.resid (*Dom t*) (*Cod t*) (*Trg u*)
(*Map t*) (*Map u*)
⟨*proof*⟩

lemma *simulation-Map-sta*:
assumes *sta f*
shows *simulation* (*Dom f*) (*Cod f*) (*Map f*)
⟨*proof*⟩

lemma *transformation-Map-arr*:
assumes *arr t*
shows *transformation* (*Dom t*) (*Cod t*) (*Src t*) (*Trg t*) (*Map t*)
⟨*proof*⟩

lemma *iso-char*:
shows *H.iso t* ⟷ *arr t* ∧ *Src t* = *Map t* ∧ *Trg t* = *Map t* ∧
invertible-simulation (*Dom t*) (*Cod t*) (*Map t*)
⟨*proof*⟩

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* ⟨*proof*⟩
interpretation I_1: *identity-simulation One.resid* ⟨*proof*⟩

abbreviation *one* (**1**)

where *one* ≡ *mkobj One.resid*

lemma *obj-one*:
shows *obj* **1**
⟨*proof*⟩

definition *trm*
where *trm a* ≡ *MkArr* (*Dom a*) *One.resid*
(*exponential-rts.MkIde*
(*constant-simulation.map* (*Dom a*) *One.resid One.the-arr*))

lemma *one-universality*:
assumes *obj a*
shows «*trm a* : *a* → **1**»
and ⋀*t*. «*t* : *a* → **1**» ⟹ *t* = *trm a*
and ∃!*t*. «*t* : *a* → **1**»
⟨*proof*⟩

lemma *terminal-one*:
shows *H.terminal* **1**
⟨*proof*⟩

lemma *trm-in-hom* [*intro*, *simp*]:
assumes *obj a*
shows «*trm a* : *a* → **1**»
⟨*proof*⟩

lemma *terminal-arrow-is-sta*:
assumes «*t* : *a* → **1**»
shows *sta t*
⟨*proof*⟩

For any object a we have an RTS isomorphism $Dom\ a \cong HOM\ \mathbf{1}\ a$. Note that these are *not* at the same type.

abbreviation *UP* :: *'A arr* ⇒ *'A* ⇒ *'A arr*
where *UP a* ≡ $MkArr_{ext}$ (\₁) (*Dom a*) ∘ *exponential-by-One.Up* (*Dom a*)

abbreviation *DN* :: *'A arr* ⇒ *'A arr* ⇒ *'A*
where *DN a* ≡ *exponential-by-One.Dn* (*Dom a*) ∘ Trn_{ext} **1** *a*

lemma *inverse-simulations-DN-UP*:
assumes *obj a*
shows *inverse-simulations* (*Dom a*) (*HOM* **1** *a*) (*DN a*) (*UP a*)
and *isomorphic-rts* (*Dom a*) (*HOM* **1** *a*)
⟨*proof*⟩

lemma *terminal-char*:
shows *H.terminal x* ⟷ *obj x* ∧ (∃!*t*. *residuation.arr* (*Dom x*) *t*)
⟨*proof*⟩

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* ⟨*proof*⟩

definition *one* (**1**)
where *one* ≡ *terminal-object-in-rtscat.one*

definition *trm*
where *trm* = *terminal-object-in-rtscat.trm*

interpretation *Trm*: *terminal-object-in-rtscat* ⟨*proof*⟩
no-notation *Trm.one* (**1**)

lemma *obj-one* [*intro*, *simp*]:
shows *obj one*
⟨*proof*⟩

lemma *trm-simps′* [*simp*]:
assumes *obj a*
shows *arr* (*trm a*) **and** *dom* (*trm a*) = *a* **and** *cod* (*trm a*) = **1**
and *src* (*trm a*) = *trm a* **and** *trg* (*trm a*) = *trm a*
and *sta* (*trm a*)
⟨*proof*⟩

sublocale *category-with-terminal-object hcomp*
⟨*proof*⟩

sublocale *elementary-category-with-terminal-object hcomp one trm*
⟨*proof*⟩

lemma *is-elementary-category-with-terminal-object*:
shows *elementary-category-with-terminal-object hcomp one trm*
⟨*proof*⟩

lemma *terminal-char*:
shows *H.terminal x* ⟷ *obj x* ∧ (∃!*t*. *residuation.arr* (*Dom x*) *t*)
⟨*proof*⟩

lemma *Map-trm*:
assumes *obj a*
shows *Map* (*trm a*) =
constant-simulation.map (*Dom a*) *One.resid One.the-arr*
⟨*proof*⟩

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*)
⟨*proof*⟩

abbreviation UP_{rts} :: *'A arr* ⇒ *'A* ⇒ *'A arr*
where UP_{rts} *a* ≡ $MkArr_{ext}$ (\$_1$) (*Dom a*) ∘ *exponential-by-One.Up* (*Dom a*)

abbreviation DN_{rts} :: *'A arr* ⇒ *'A arr* ⇒ *'A*
where DN_{rts} *a* ≡ *exponential-by-One.Dn* (*Dom a*) ∘ Trn_{ext} **1** *a*

lemma *UP-DN-naturality*:
assumes *arr t*
shows DN_{rts} (*cod t*) ∘ *cov-HOM* **1** *t* = *Map t* ∘ DN_{rts} (*dom t*)
and UP_{rts} (*cod t*) ∘ *Map t* = *cov-HOM* **1** *t* ∘ UP_{rts} (*dom t*)
and *cov-HOM* **1** *t* = UP_{rts} (*cod t*) ∘ *Map t* ∘ DN_{rts} (*dom t*)
and *Map t* = DN_{rts} (*cod t*) ∘ *cov-HOM* **1** *t* ∘ UP_{rts} (*dom t*)
⟨*proof*⟩

Equality of parallel arrows «*u* : *a* → *b*» and «*v* : *a* → *b*» is determined by their compositions with global transitions «*t* : **1** → *a*».

lemma *arr-extensionality*:
assumes «*u* : *a* → *b*» **and** «*v* : *a* → *b*» **and** *src u* = *src v* **and** *trg u* = *trg v*
shows *u* = *v* ⟷ (∀ *t*. «*t* : **1** → *a*» ⟶ *u* ⋆ *t* = *v* ⋆ *t*)
⟨*proof*⟩

lemma *sta-extensionality*:
assumes «*f* : *a* →$_{sta}$ *b*» **and** «*g* : *a* →$_{sta}$ *b*»
shows *f* = *g* ⟷ (∀ *t*. «*t* : **1** → *a*» ⟶ *f* ⋆ *t* = *g* ⋆ *t*)
⟨*proof*⟩

The mapping *HOM* **1**, like *Dom*, takes each object to a corresponding RTS, but unlike *Dom* it stays at type *'A arr*, rather than decreasing the type from *'A arr* to *'A*.

lemma *HOM1-mapsto*:
shows *HOM* **1** ∈ *Collect obj* → *Collect extensional-rts* ∩ *Collect small-rts*
⟨*proof*⟩

The mapping *HOM* **1** is not necessarily injective, but it is essentially so.

lemma *HOM1-reflects-isomorphic*:
assumes *obj a* **and** *obj b* **and** *isomorphic-rts* (*HOM* **1** *a*) (*HOM* **1** *b*)
shows *H.isomorphic a b*
⟨*proof*⟩

end

4.3.2 Products

In this section we show that the category $\mathbf{RTS}^\dagger$ has products. A product of objects a and b is obtained by constructing the product $Dom\ a \times Dom\ b$ of their underlying RTS's and then showing that there exists an object $a \otimes b$ such that $Dom\ (a \otimes b)$ is isomorphic to $Dom\ a \times Dom\ b$. Since $Dom\ (a \otimes b)$ will have arrow type $'A$, but $Dom\ a \otimes Dom\ b$ has arrow type $'A \times 'A$, we need a way to reduce the arrow type of $Dom\ a \otimes Dom\ b$ from $'A \times 'A$ to $'A$. This is done by using the assumption that the type $'A$ admits pairing to obtain an injective map from $'A \times 'A$ to $'A$, and then applying the injective image construction to obtain an RTS with arrow type $'A$ that is isomorphic to $Dom\ a \otimes Dom\ b$.

locale *product-in-rtscat* =
  rtscatx arr-type
for *arr-type* :: *'A itself*
and *a*
and *b* +
assumes *obj-a*: *obj a*
and *obj-b*: *obj b*
begin

  notation *hcomp* (**infixr** ⋆ *53*)

  interpretation *A*: *extensional-rts* ‹*Dom a*›
    ⟨*proof*⟩
  interpretation *A*: *small-rts* ‹*Dom a*›
    ⟨*proof*⟩
  interpretation *B*: *extensional-rts* ‹*Dom b*›
    ⟨*proof*⟩
  interpretation *B*: *small-rts* ‹*Dom b*›
    ⟨*proof*⟩
  interpretation *AB*: *exponential-rts* ‹*Dom a*› ‹*Dom b*› ⟨*proof*⟩

  sublocale *PROD*: *product-rts* ‹*Dom a*› ‹*Dom b*› ⟨*proof*⟩
  sublocale *PROD*: *product-of-extensional-rts* ‹*Dom a*› ‹*Dom b*› ⟨*proof*⟩
  sublocale *PROD*: *product-of-small-rts* ‹*Dom a*› ‹*Dom b*› ⟨*proof*⟩

  sublocale *Prod*: *inj-image-rts pairing.some-pair PROD.resid*
    ⟨*proof*⟩
  sublocale *Prod*: *small-rts Prod.resid*
    ⟨*proof*⟩
  sublocale *Prod*: *extensional-rts Prod.resid*
    ⟨*proof*⟩

The injective image construction on RTS's gives us invertible simulations between *Prod.resid* and *PROD.resid*.

  abbreviation *Pack* :: *'A* × *'A* ⇒ *'A*
  where *Pack* ≡ *Prod.map*$_{ext}$

abbreviation *Unpack* :: *'A* ⇒ *'A* × *'A*
where *Unpack* ≡ *Prod.map'*$_{ext}$

interpretation P_1: *composite-simulation Prod.resid PROD.resid* ‹*Dom a*› *Unpack PROD.*P_1
⟨*proof*⟩
interpretation P_0: *composite-simulation Prod.resid PROD.resid* ‹*Dom b*› *Unpack PROD.*P_0
⟨*proof*⟩

abbreviation *prod* :: *'A arr*
where *prod* ≡ *mkobj Prod.resid*

lemma *obj-prod*:
shows *obj prod*
⟨*proof*⟩

lemma *Dom-prod* [*simp*]:
shows *Dom prod* = *Prod.resid*
⟨*proof*⟩

definition p_0 :: *'A arr*
where p_0 ≡ *mksta Prod.resid* (*Dom b*) P_0*.map*

definition p_1 :: *'A arr*
where p_1 ≡ *mksta Prod.resid* (*Dom a*) P_1*.map*

lemma p_0*-simps* [*simp*]:
shows *sta* p_0 **and** *dom* p_0 = *prod* **and** *cod* p_0 = *b*
and *Dom* p_0 = *Prod.resid* **and** *Cod* p_0 = *Dom b*
and *Trn* p_0 = *exponential-rts.MkIde* P_0*.map*
⟨*proof*⟩

lemma p_1*-simps* [*simp*]:
shows *sta* p_1 **and** *dom* p_1 = *prod* **and** *cod* p_1 = *a*
and *Dom* p_1 = *Prod.resid* **and** *Cod* p_1 = *Dom a*
and *Trn* p_1 = *exponential-rts.MkIde* P_1*.map*
⟨*proof*⟩

lemma p_0*-in-hom* [*intro*]:
shows «p_0 : *prod* → *b*»
⟨*proof*⟩

lemma p_1*-in-hom* [*intro*]:
shows «p_1 : *prod* → *a*»
⟨*proof*⟩

It should be noted that the length of the proof of the following result is partly due to the fact that it is proving something rather stronger than

one might expect at first blush. The category we are working with here is analogous to a 2-category in the sense that there are essentially two classes of arrows: *states*, which correspond to simulations between RTS's, and *transitions*, which correspond to transformations. The class of states is included in the class of transitions. The universality result below shows the universality of the product for the full class of arrows, so it is in that sense analogous to showing that the category has 2-products, rather than just ordinary products.

lemma *universality*:
assumes «$h : x \to a$» **and** «$k : x \to b$»
shows $\exists!m.\ p_1 \star m = h \wedge p_0 \star m = k$
⟨*proof*⟩

lemma *has-as-binary-product*:
shows *H.has-as-binary-product a b* p_1 p_0
⟨*proof*⟩

sublocale *binary-product hcomp a b* p_1 p_0
⟨*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-*p_0:
shows *Map* p_0 = *PROD.*P_0 ∘ *Unpack*
⟨*proof*⟩

lemma *Map-*p_1:
shows *Map* p_1 = *PROD.*P_1 ∘ *Unpack*
⟨*proof*⟩

lemma *Map-tuple*:
assumes «$t : x \to a$» **and** «$u : x \to b$»
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 p_0
where $p_0 \equiv$ *product-in-rtscat*.p_0

definition p_1
where $p_1 \equiv$ *product-in-rtscat*.p_1

lemma *sta-*p_0:
assumes *obj a* **and** *obj b*
shows *sta* (p_0 *a b*)
⟨*proof*⟩

lemma *sta-*p_1:
assumes *obj a* **and** *obj b*
shows *sta* (p_1 *a b*)
⟨*proof*⟩

lemma *has-binary-products*:
assumes *obj a* **and** *obj b*
shows *H.has-as-binary-product a b* (p_1 *a b*) (p_0 *a b*)
⟨*proof*⟩

interpretation *category-with-binary-products hcomp*
⟨*proof*⟩

proposition *is-category-with-binary-products*:
shows *category-with-binary-products hcomp*
⟨*proof*⟩

lemma *extends-to-elementary-category-with-binary-products*:
shows *elementary-category-with-binary-products hcomp* p_0 p_1
⟨*proof*⟩

interpretation *elementary-category-with-binary-products hcomp* p_0 p_1
⟨*proof*⟩

notation p_0 ($\mathfrak{p}_0$[-, -])
notation p_1 ($\mathfrak{p}_1$[-, -])
notation *tuple* (⟨-, -⟩)
notation *prod* (**infixr** ⊗ *51*)

lemma *prod-eq*:
assumes *obj a* **and** *obj b*
shows $a \otimes b$ = *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*⟩

definition *Pack* :: *'A arr* ⇒ *'A arr* ⇒ *'A* × *'A* ⇒ *'A*
where *Pack* ≡ *product-in-rtscat.Pack*

definition *Unpack* :: *'A arr* ⇒ *'A arr* ⇒ *'A* ⇒ *'A* × *'A*
where *Unpack* ≡ *product-in-rtscat.Unpack*

lemma *inverse-simulations-Pack-Unpack*:
assumes *obj a* **and** *obj b*
shows *inverse-simulations* (*Dom* (*a* ⊗ *b*)) (*product-rts.resid* (*Dom a*) (*Dom b*))
(*Pack a b*) (*Unpack a b*)
⟨*proof*⟩

lemma *simulation-Pack*:
assumes *obj a* **and** *obj b*
shows *simulation* (*product-rts.resid* (*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*)) (*product-rts.resid* (*Dom a*) (*Dom b*))
(*Unpack a b*)
⟨*proof*⟩

lemma *Pack-o-Unpack*:
assumes *obj a* **and** *obj b*
shows *Pack a b* ∘ *Unpack a b* = *I* (*Dom* (*a* ⊗ *b*))
⟨*proof*⟩

lemma *Unpack-o-Pack*:
assumes *obj a* **and** *obj b*
shows *Unpack a b* ∘ *Pack a b* = *I* (*product-rts.resid* (*Dom a*) (*Dom b*))
⟨*proof*⟩

lemma *Pack-Unpack* [*simp*]:
assumes *obj a* **and** *obj b*
and *residuation.arr* (*Dom* (*a* ⊗ *b*)) *t*
shows *Pack a b* (*Unpack a b t*) = *t*
⟨*proof*⟩

lemma *Unpack-Pack* [*simp*]:
assumes *obj a* **and** *obj b*
and *residuation.arr* (*product-rts.resid* (*Dom a*) (*Dom b*)) *t*
shows *Unpack a b* (*Pack a b t*) = *t*
⟨*proof*⟩

lemma *src-tuple* [*simp*]:
assumes *H.span t u*
shows *src* ⟨*t*, *u*⟩ = ⟨*src t*, *src u*⟩
⟨*proof*⟩

lemma *trg-tuple* [*simp*]:
assumes *H.span t u*
shows *trg* ⟨*t*, *u*⟩ = ⟨*trg t*, *trg u*⟩
⟨*proof*⟩

lemma *Map-*$\mathfrak{p}_0$:
assumes *obj a* **and** *obj b*
shows *Map* $\mathfrak{p}_0[a, b]$ = *product-rts.P*$_0$ (*Dom a*) (*Dom b*) ∘ *Unpack a b*
⟨*proof*⟩

lemma *Map-*$\mathfrak{p}_1$:
assumes *obj a* **and** *obj b*
shows *Map* $\mathfrak{p}_1[a, b]$ = *product-rts.P*$_1$ (*Dom a*) (*Dom b*) ∘ *Unpack a b*
⟨*proof*⟩

lemma *Map-tuple*:
assumes «*t* : *x* → *a*» **and** «*u* : *x* → *b*»
shows *Map* ⟨*t*, *u*⟩ = *Pack a b* ∘ ⟨⟨*Map t*, *Map u*⟩⟩
⟨*proof*⟩

lemma *assoc-expansion*:
assumes *obj a* **and** *obj b* **and** *obj c*
shows *assoc a b c* =
⟨$\mathfrak{p}_1[a, b]$ ⋆ $\mathfrak{p}_1[a \otimes b, c]$, ⟨$\mathfrak{p}_0[a, b]$ ⋆ $\mathfrak{p}_1[a \otimes b, c]$, $\mathfrak{p}_0[a \otimes b, c]$⟩ ⟩
⟨*proof*⟩

end

4.3.3 Exponentials

In this section we show that the category **RTS**† 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
    ⟨proof⟩

  notation hcomp  (infixr ⋆ 53)
  notation p₀     (𝔭₀[-, -])
  notation p₁     (𝔭₁[-, -])
  notation tuple  (⟨-, -⟩)
  notation prod   (infixr ⊗ 51)

  sublocale B: extensional-rts ‹Dom b›
    ⟨proof⟩
```

sublocale *B*: *small-rts* ‹*Dom b*›
⟨*proof*⟩
sublocale *C*: *extensional-rts* ‹*Dom c*›
⟨*proof*⟩
sublocale *C*: *small-rts* ‹*Dom c*›
⟨*proof*⟩

sublocale *EXP*: *exponential-rts* ‹*Dom b*› ‹*Dom c*› ⟨*proof*⟩
sublocale *EXP*: *exponential-of-small-rts* ‹*Dom b*› ‹*Dom c*› ⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩
sublocale *Exp*: *extensional-rts Exp.resid*
⟨*proof*⟩
sublocale *Exp*: *small-rts Exp.resid*
⟨*proof*⟩

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

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

abbreviation *Func* :: *'A* ⇒ (*'A*, *'A*) *EXP.arr*
where *Func* ≡ *Exp.map'*$_{ext}$

abbreviation *Unfunc* :: (*'A*, *'A*) *EXP.arr* ⇒ *'A*
where *Unfunc* ≡ *Exp.map*$_{ext}$

We define *exp* to be the object of the category **RTS**† having *Exp* as its underlying RTS.

definition *exp*
where *exp* ≡ *mkobj Exp.resid*

lemma *obj-exp*:
shows *obj exp*
⟨*proof*⟩

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*
⟨*proof*⟩

sublocale *EXPxB*: *product-rts EXP.resid* ‹*Dom b*› ⟨*proof*⟩
sublocale *ExpxB*: *product-rts Exp.resid* ‹*Dom b*› ⟨*proof*⟩

sublocale *B*: *identity-simulation* ‹*Dom b*› ⟨*proof*⟩
sublocale *B*: *simulation-as-transformation* ‹*Dom b*› ‹*Dom b*› *B.map* ⟨*proof*⟩
sublocale *B*: *transformation-to-extensional-rts*
‹*Dom b*› ‹*Dom b*› *B.map B.map B.map* ⟨*proof*⟩
sublocale *UnfuncxB*: *product-simulation*
EXP.resid ‹*Dom b*› *Exp.resid* ‹*Dom b*› *Unfunc B.map* ⟨*proof*⟩
sublocale *FuncxB*: *product-simulation*
Exp.resid ‹*Dom b*› *EXP.resid* ‹*Dom b*› *Func B.map* ⟨*proof*⟩

sublocale *inverse-simulations EXPxB.resid ExpxB.resid*
FuncxB.map UnfuncxB.map
⟨*proof*⟩

lemma *obj-expxb*:
shows *obj* (*exp* ⊗ *b*)
⟨*proof*⟩

We now have a simulation *FuncxB-o-Unpack*, which refers to the result of composing the isomorphism *Unpack exp b* from *Dom expxb* to *ExpxB*, with the isomorphism *FuncxB* from *ExpxB* to *EXPxB*. This composite essentially "unpacks" the RTS *Dom expxb*, which underlies the product object *expxb*, to expose its construction as an application of the exponential RTS construction, followed by an application of the product RTS construction.

sublocale *FuncxB-o-Unpack*: *composite-simulation*
‹*Dom* (*exp* ⊗ *b*)› *ExpxB.resid EXPxB.resid*
‹*Unpack exp b*› *FuncxB.map*
⟨*proof*⟩

We construct the evaluation map associated with *ExpxB* by composing the evaluation map *Eval.map* from *EXPxB* to *C*, derived from the exponential RTS construction, with the isomorphism *FuncxB-o-Unpack* from *Dom expxb.prod* to *EXPxB* and then obtain the corresponding arrow of the category.

sublocale *Eval*: *evaluation-map* ‹*Dom b*› ‹*Dom c*› ⟨*proof*⟩
sublocale *Eval*: *evaluation-map-between-extensional-rts* ‹*Dom b*› ‹*Dom c*› ⟨*proof*⟩
sublocale *Eval-o-FuncxB-o-Unpack*:
composite-simulation
‹*Dom* (*exp* ⊗ *b*)› *EXPxB.resid* ‹*Dom c*›

FuncxB-o-Unpack.map Eval.map
⟨*proof*⟩

lemma *EvaloFuncxB-o-Unpack-is-simulation*:
shows *simulation* (*Dom* (*exp* ⊗ *b*)) (*Dom c*) *Eval-o-FuncxB-o-Unpack.map*
⟨*proof*⟩

definition *eval*
where *eval* ≡ *mksta* (*Dom* (*exp* ⊗ *b*)) (*Dom c*) *Eval-o-FuncxB-o-Unpack.map*

lemma *eval-simps* [*simp*]:
shows *sta eval* **and** *dom eval* = *exp* ⊗ *b* **and** *cod eval* = *c*
and *Dom eval* = *Dom* (*exp* ⊗ *b*) **and** *Cod eval* = *Dom c*
and *Trn eval* = *exponential-rts.MkIde Eval-o-FuncxB-o-Unpack.map*
⟨*proof*⟩

lemma *eval-in-hom* [*intro*]:
shows «*eval* : *exp* ⊗ *b* → *c*»
⟨*proof*⟩

lemma *Map-eval*:
shows *Map eval* = *Eval.map* ∘ (*FuncxB.map* ∘ *Unpack exp b*)
⟨*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** ⊗ *51*)

definition *exp*
where *exp b c* ≡ *exponential-in-rtscat.exp b c*

lemma *obj-exp*:
assumes *obj b* **and** *obj c*
shows *obj* (*exp b c*)
⟨*proof*⟩

definition *eval*
where *eval b c* ≡ *exponential-in-rtscat.eval b c*

lemma *eval-simps* [*simp*]:
assumes *obj b* **and** *obj c*
shows *sta* (*eval b c*)

and *dom* (*eval b c*) = *exp b c* ⊗ *b*
and *cod* (*eval b c*) = *c*
⟨*proof*⟩

lemma *eval-in-hom*$_{RCR}$ [*intro*]:
assumes *obj b* **and** *obj c*
shows «*eval b c* : *exp b c* ⊗ *b* → *c*»
⟨*proof*⟩

definition *Func* :: *'a arr* ⇒ *'a arr* ⇒ *'a* ⇒ (*'a*, *'a*) *exponential-rts.arr*
where *Func* ≡ *exponential-in-rtscat.Func*

definition *Unfunc* :: *'a arr* ⇒ *'a arr* ⇒ (*'a*, *'a*) *exponential-rts.arr* ⇒ *'a*
where *Unfunc* ≡ *exponential-in-rtscat.Unfunc*

lemma *inverse-simulations-Func-Unfunc*:
assumes *obj b* **and** *obj c*
shows *inverse-simulations*
(*exponential-rts.resid* (*Dom b*) (*Dom c*)) (*Dom* (*exp b c*))
(*Func b c*) (*Unfunc b c*)
⟨*proof*⟩

lemma *simulation-Func*:
assumes *obj b* **and** *obj c*
shows *simulation* (*Dom* (*exp b c*)) (*exponential-rts.resid* (*Dom b*) (*Dom c*))
(*Func b c*)
⟨*proof*⟩

lemma *simulation-Unfunc*:
assumes *obj b* **and** *obj c*
shows *simulation* (*exponential-rts.resid* (*Dom b*) (*Dom c*)) (*Dom* (*exp b c*))
(*Unfunc b c*)
⟨*proof*⟩

lemma *Func-o-Unfunc*:
assumes *obj b* **and** *obj c*
shows *Func b c* ∘ *Unfunc b c* = *I* (*exponential-rts.resid* (*Dom b*) (*Dom c*))
⟨*proof*⟩

lemma *Unfunc-o-Func*:
assumes *obj b* **and** *obj c*
shows *Unfunc b c* ∘ *Func b c* = *I* (*Dom* (*exp b c*))
⟨*proof*⟩

lemma *Func-Unfunc* [*simp*]:
assumes *obj b* **and** *obj c*
and *residuation.arr* (*exponential-rts.resid* (*Dom b*) (*Dom c*)) *t*
shows *Func b c* (*Unfunc b c t*) = *t*

⟨*proof*⟩

lemma *Unfunc-Func* [*simp*]:
assumes *obj b* **and** *obj c*
and *residuation.arr* (*Dom* (*exp b c*)) *t*
shows *Unfunc b c* (*Func b c t*) = *t*
⟨*proof*⟩

lemma *Map-eval*:
assumes *obj b* **and** *obj c*
shows *Map* (*eval b c*) =
evaluation-map.map (*Dom b*) (*Dom c*) ∘
(*product-simulation.map*
(*Dom* (*exp b c*)) (*Dom b*) (*Func b c*) (*I* (*Dom b*)) ∘
Unpack (*exp b c*) *b*)
⟨*proof*⟩

end

locale *currying-in-rtscat* =
exponential-in-rtscat arr-type b c
for *arr-type* :: *'A itself*
and *a* :: *'A rtscatx.arr*
and *b* :: *'A rtscatx.arr*
and *c* :: *'A rtscatx.arr* +
assumes *obj-a*: *obj a*
begin

sublocale *A*: *extensional-rts* ‹*Dom a*›
⟨*proof*⟩
sublocale *A*: *small-rts* ‹*Dom a*›
⟨*proof*⟩
sublocale *B*: *extensional-rts* ‹*Dom b*›
⟨*proof*⟩
sublocale *B*: *small-rts* ‹*Dom b*›
⟨*proof*⟩

sublocale *AxB*: *product-of-extensional-rts* ‹*Dom a*› ‹*Dom b*› ⟨*proof*⟩
sublocale *A-Exp*: *exponential-rts* ‹*Dom a*› *Exp.resid* ⟨*proof*⟩

sublocale *aXb*: *extensional-rts* ‹*Dom* (*a* ⊗ *b*)›
⟨*proof*⟩
sublocale *aXb*: *small-rts* ‹*Dom* (*a* ⊗ *b*)›
⟨*proof*⟩
sublocale *expXb*: *exponential-rts Exp.resid* ‹*Dom b*› ⟨*proof*⟩
sublocale *aXb-C*: *exponential-rts* ‹*Dom* (*a* ⊗ *b*)› ‹*Dom c*› ⟨*proof*⟩

sublocale *Currying* ‹*Dom a*› ‹*Dom b*› ‹*Dom c*› ⟨*proof*⟩

definition *curry* :: *'A arr* ⇒ *'A arr*
where *curry f* = *mkarr* (*Dom a*) *Exp.resid*
(*Unfunc* ∘ *Curry3* (*aXb-C.Dom* (*Trn f*) ∘ *Pack a b*))
(*Unfunc* ∘ *Curry3* (*aXb-C.Cod* (*Trn f*) ∘ *Pack a b*))
(*Unfunc* ∘ *Curry* (*aXb-C.Dom* (*Trn f*) ∘ *Pack a b*)
(*aXb-C.Cod* (*Trn f*) ∘ *Pack a b*)
(*aXb-C.Map* (*Trn f*) ∘ *Pack a b*))

lemma *curry-in-hom* [*intro*]:
assumes «*f* : *a* ⊗ *b* → *c*»
shows «*curry f* : *a* → *exp*»
⟨*proof*⟩

lemma *curry-simps* [*simp*]:
assumes «*t* : *a* ⊗ *b* → *c*»
shows *arr* (*curry t*) **and** *dom* (*curry t*) = *a* **and** *cod* (*curry t*) = *exp*
and *Dom* (*curry t*) = *Dom a* **and** *Cod* (*curry t*) = *Exp.resid*
and *src* (*curry t*) = *curry* (*src t*) **and** *trg* (*curry t*) = *curry* (*trg t*)
and *Map* (*curry t*) =
(*Unfunc* ∘ *Curry* (*aXb-C.Dom* (*Trn t*) ∘ *Pack a b*)
(*aXb-C.Cod* (*Trn t*) ∘ *Pack a b*)
(*aXb-C.Map* (*Trn t*) ∘ *Pack a b*))
⟨*proof*⟩

lemma *sta-curry*:
assumes «*f* : *a* ⊗ *b* → *c*» **and** *sta f*
shows *sta* (*curry f*)
⟨*proof*⟩

definition *uncurry* :: *'A arr* ⇒ *'A arr*
where *uncurry g* = *mkarr* (*Dom* (*a* ⊗ *b*)) (*Dom c*)
(*Uncurry* (*Func* ∘ *exponential-rts.Dom* (*Trn g*)) ∘ *Unpack a b*)
(*Uncurry* (*Func* ∘ *exponential-rts.Cod* (*Trn g*)) ∘ *Unpack a b*)
(*Uncurry* (*Func* ∘ *exponential-rts.Map* (*Trn g*)) ∘ *Unpack a b*)

lemma *uncurry-in-hom* [*intro*]:
assumes «*g* : *a* → *exp*»
shows «*uncurry g* : *a* ⊗ *b* → *c*»
⟨*proof*⟩

lemma *uncurry-simps* [*simp*]:
assumes «*u* : *a* → *exp*»
shows *arr* (*uncurry u*)
and *dom* (*uncurry u*) = *a* ⊗ *b* **and** *cod* (*uncurry u*) = *c*
and *Dom* (*uncurry u*) = *Dom* (*a* ⊗ *b*) **and** *Cod* (*uncurry u*) = *Dom c*
and *Map* (*uncurry u*) =
Uncurry (*Func* ∘ *exponential-rts.Map* (*Trn u*)) ∘ *Unpack a b*
and *src* (*uncurry u*) = *uncurry* (*src u*)

and *trg* (*uncurry u*) = *uncurry* (*trg u*)
⟨*proof*⟩

lemma *sta-uncurry*:
assumes «*g* : *a* → *exp*» **and** *sta g*
shows *sta* (*uncurry g*)
⟨*proof*⟩

lemma *uncurry-curry*:
assumes *obj a* **and** *obj b*
and «*t* : *a* ⊗ *b* → *c*»
shows *uncurry* (*curry t*) = *t*
⟨*proof*⟩

lemma *curry-uncurry*:
assumes «*u* : *a* → *exp*»
shows *curry* (*uncurry u*) = *u*
⟨*proof*⟩

We are not yet quite where we want to go, because to establish the naturality of the curry/uncurry bijection we have to show how uncurry relates to evaluation.

lemma *uncurry-expansion*:
assumes «*u* : *a* → *exp*»
shows *uncurry u* = *eval* ⋆ ⟨*u* ⋆ 𝔭₁[*a*, *b*], 𝔭₀[*a*, *b*]⟩
⟨*proof*⟩

end

Once again, we transfer the things we want to *rtscatx*.

context *rtscatx*
begin

interpretation *elementary-category-with-binary-products hcomp* p_0 p_1
⟨*proof*⟩

notation *hcomp* (**infixr** ⋆ *53*)
notation p_0 (𝔭₀[-, -])
notation p_1 (𝔭₁[-, -])
notation *tuple* (⟨-, -⟩)
notation *prod* (**infixr** ⊗ *51*)

definition *curry* :: *'A arr* ⇒ *'A arr* ⇒ *'A arr* ⇒ *'A arr* ⇒ *'A arr*
where *curry* ≡ *currying-in-rtscat.curry*

definition *uncurry* :: *'A arr* ⇒ *'A arr* ⇒ *'A arr* ⇒ *'A arr* ⇒ *'A arr*
where *uncurry* ≡ *currying-in-rtscat.uncurry*

lemma *curry-in-hom* [*intro*, *simp*]:

assumes *obj a* **and** *obj b*
and «$f : a \otimes b \to c$»
shows «*curry a b c f* : $a \to$ *exp b c*»
⟨*proof*⟩

lemma *curry-simps* [*simp*]:
assumes *obj a* **and** *obj b*
and «$f : a \otimes b \to c$»
shows *arr* (*curry a b c f*)
and *dom* (*curry a b c f*) = *a* **and** *cod* (*curry a b c f*) = *exp b c*
and *src* (*curry a b c f*) = *curry a b c* (*src f*)
and *trg* (*curry a b c f*) = *curry a b c* (*trg f*)
⟨*proof*⟩

lemma *sta-curry*:
assumes *obj a* **and** *obj b*
and «$f : a \otimes b \to c$» **and** *sta f*
shows *sta* (*curry a b c f*)
⟨*proof*⟩

lemma *uncurry-in-hom* [*intro*, *simp*]:
assumes *obj b* **and** *obj c*
and «$g : a \to$ *exp b c*»
shows «*uncurry a b c g* : $a \otimes b \to c$»
⟨*proof*⟩

lemma *uncurry-simps* [*simp*]:
assumes *obj b* **and** *obj c*
and «$g : a \to$ *exp b c*»
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*)
⟨*proof*⟩

lemma *sta-uncurry*:
assumes *obj b* **and** *obj c*
and «$g : a \to$ *exp b c*» **and** *sta g*
shows *sta* (*uncurry a b c g*)
⟨*proof*⟩

lemma *uncurry-curry*:
assumes *obj a* **and** *obj b*
and «$t : a \otimes b \to c$»
shows *uncurry a b c* (*curry a b c t*) = *t*
⟨*proof*⟩

lemma *curry-uncurry*:
assumes *obj b* **and** *obj c*

and «$u : a \to exp\ b\ c$»
shows $curry\ a\ b\ c\ (uncurry\ a\ b\ c\ u) = u$
⟨*proof*⟩

lemma *uncurry-expansion*:
assumes $obj\ b$ **and** $obj\ c$
and «$u : a \to exp\ b\ c$»
shows $uncurry\ a\ b\ c\ u = eval\ b\ c \star (u \otimes b)$
⟨*proof*⟩

lemma *Map-curry*:
assumes $obj\ a$ **and** $obj\ b$ **and** $obj\ c$
shows $Map\ (curry\ a\ b\ c\ f) =$
$Unfunc\ b\ c \circ$
$Currying.Curry\ (Dom\ a)\ (Dom\ b)\ (Dom\ c)$
$(Src\ f \circ Pack\ a\ b)\ (Trg\ f \circ Pack\ a\ b)\ (Map\ f \circ Pack\ a\ b)$
⟨*proof*⟩

lemma *Map-uncurry*:
assumes $obj\ a$ **and** $obj\ b$ **and** $obj\ c$
shows $Map\ (uncurry\ a\ b\ c\ g) =$
$Currying.Uncurry\ (Dom\ a)\ (Dom\ b)\ (Dom\ c)$
$(Func\ b\ c \circ exponential\text{-}rts.Map\ (Trn\ g)) \circ Unpack\ a\ b$
⟨*proof*⟩

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* p_0 p_1
⟨*proof*⟩

notation *prod* (**infixr** $\otimes$ *51*)

interpretation *elementary-cartesian-closed-category*
hcomp p_0 p_1 *one trm exp eval curry*
⟨*proof*⟩

lemma *is-elementary-cartesian-closed-category*:
shows *elementary-cartesian-closed-category*
hcomp p_0 p_1 *one trm exp eval curry*
⟨*proof*⟩

theorem *is-cartesian-closed-category*:
shows *cartesian-closed-category hcomp*

⟨*proof*⟩

end

4.3.5 Repleteness

context *rtscatx*
begin

We have shown that the RTS-category $\mathbf{RTS}^\dagger$ has objects that are in bijective correspondence with small extensional RTS's, states (identities for the vertical residuation) that are in bijective correspondence with simulations, and arrows that are in bijective correspondence with transformations. These results allow us to pass back and forth between external constructions on RTS's and internal structure of the RTS-category, as was demonstrated in the proof of cartesian closure. However, these results make use of extra structure beyond that of an RTS-category; namely the mapping *Dom* that takes an object to its underlying RTS. We would like to have a characterization of $\mathbf{RTS}^\dagger$ in terms that make sense for an abstract RTS-category without additional structure. It seems that it should be possible to do this, because as we have shown, for any object a the RTS *Dom a* is isomorphic to *Hom* **1** a. So we ought to be able to dispense with the extrinsic mapping *Dom* and work instead with the intrinsic mapping *Hom* **1**. However, there is an issue here to do with types. The mapping *Dom* takes an object a to a small extensional RTS *Dom a* having arrow type $'A$. On the other hand, the RTS *Hom* **1** a has arrow type $'A$ *arr*. So one thing that needs to be done in order to carry out this program is to express the "object repleteness" of $\mathbf{RTS}^\dagger$ in terms of small extensional RTS's with arrow type $'A$ *arr*, as opposed to small extensional RTS's with arrow type $'A$. However, the type $'A$ *arr* is larger than the type $'A$, and consequently it could admit a larger class of small extensional RTS's than type $'A$ does. It is possible, though, to define a mapping from $'A$ *arr* to $'A$ whose restriction to the set of arrows (and null) of *rtscatx* is injective. This will allow us to take any small extensional RTS A with arrow type $'A$ *arr*, as long as its arrows and null are drawn from the set of arrows and null of *rtscatx* as a whole, and obtain an isomorphic image of it with arrow type $'A$.

We first define the required mapping from $'A$ *arr* to $'A$.

```
fun inj-arr :: 'A arr ⇒ 'A
where inj-arr (MkArr A B F) =
      lifting.some-lift
        (Some (pairing.some-pair
                (some-inj-resid A,
                 pairing.some-pair
                   (some-inj-resid B, inj-exp F))))
  | inj-arr Null = lifting.some-lift None
```

The mapping *inj-arr* has the required injectiveness property.

lemma *inj-inj-arr*:
fixes A :: $'A$ *arr resid*
assumes *small-rts A* **and** *extensional-rts A*
and *Collect* (*residuation.arr A*) ∪
{*ResiduatedTransitionSystem.partial-magma.null A*} ⊆
Collect arr ∪ {*Null*}
shows *inj-on inj-arr*
(*Collect* (*residuation.arr A*) ∪
{*ResiduatedTransitionSystem.partial-magma.null A*})
⟨*proof*⟩

The following result says that, for any small extensional RTS A whose arrows inhabit type $'A$ *arr resid* and are drawn from among the arrows and null of *rtscatx*, there is an object a of *rtscatx* such that the RTS *HOM* **1** a is isomorphic to A. It is expressed in terms that are intrinsic to **RTS**† as an abstract RTS-category, as opposed to the fact *bij-mkobj*, which uses the extrinsically given mapping *Dom*. The result is proved by taking an isomorphic image of the given RTS A under the injective mapping *inj-arr* :: $'A$ *arr* ⇒ $'A$, then applying *bij-mkobj* to obtain the corresponding object a, and finally using the isomorphism *Dom* a ≅ *HOM* **1** a to conclude that *HOM* **1** a ≅ A.

lemma *obj-replete*:
fixes A :: $'A$ *arr resid*
assumes *small-rts A* ∧ *extensional-rts A*
and *Collect* (*residuation.arr A*) ∪
{*ResiduatedTransitionSystem.partial-magma.null A*}
⊆ *Collect arr* ∪ {*null*}
shows ∃ *a*. *obj a* ∧ *isomorphic-rts A* (*HOM* **1** *a*)
⟨*proof*⟩

We now turn our attention to showing that, for any given objects a and b, the states from a to b correspond bijectively (via the "covariant hom" mapping *cov-HOM*) to simulations from *HOM* **1** a to *HOM* **1** b and the arrows from a to b correspond bijectively to the transformations between such simulations.

lemma *HOM1-faithful-for-sta*:
assumes «f : a →$_{sta}$ b» **and** «g : a →$_{sta}$ b»
and *cov-HOM* **1** f = *cov-HOM* **1** g
shows f = g
⟨*proof*⟩

lemma *HOM1-faithful-for-arr*:
assumes *arr t* **and** *arr u* **and** *src t* = *src u* **and** *trg t* = *trg u*
and *cov-HOM* **1** t = *cov-HOM* **1** u
shows t = u
⟨*proof*⟩

lemma *HOM1-full-for-sta*:
assumes *obj a* **and** *obj b* **and** *simulation* (*HOM* **1** *a*) (*HOM* **1** *b*) *F*
shows $\exists f.$ «$f : a \to b$» $\wedge$ *sta f* $\wedge$ *cov-HOM* **1** *f* = *F*
⟨*proof*⟩

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 $\exists t.$ *arr t* $\wedge$ *src t* = *f* $\wedge$ *trg t* = *g* $\wedge$ *cov-HOM* **1** *t* = *T*
⟨*proof*⟩

lemma *bij-HOM1-sta*:
assumes *obj a* **and** *obj b*
shows *bij-betw* (*cov-HOM* **1**) {*f*. «$f : a \to_{sta} b$»}
(*Collect* (*simulation* (*HOM* **1** *a*) (*HOM* **1** *b*)))
⟨*proof*⟩

lemma *bij-HOM1-arr*:
assumes «$f : a \to_{sta} b$» **and** «$g : a \to_{sta} b$»
shows *bij-betw* (*cov-HOM* **1**) {*t*. «$t : f \Rightarrow g$»}
(*Collect* (*transformation* (*HOM* **1** *a*) (*HOM* **1** *b*)
(*cov-HOM* **1** *f*) (*cov-HOM* **1** *g*)))
⟨*proof*⟩

My original objective for the results in this section was to obtain a characterization up to equivalence of the RTS-category $\mathbf{RTS}^\dagger$ in terms of intrinsic notions that make sense for any RTS-category, and to carry out the proof of cartesian closure using *HOM* **1** in place of *Dom*. This can probably be done, and I did push the idea through the construction of products, but for exponentials there were some technicalities that started to get messy and become distractions from the main things that I was trying to do. So I decided to leave this program for future work.

end

end

4.4 The Category of RTS's and Simulations

In this section, we show that the subcategory of $\mathbf{RTS}^\dagger$, comprised of the arrows that are identities with respect to the residuation, is also cartesian closed. In informal text, we will refer to this category as **RTS**. In a later section, we will show that the entire structure of the RTS-category $\mathbf{RTS}^\dagger$ is already determined by the ordinary subcategory **RTS**.

theory *RTSCat*
imports *Main RTSCatx EnrichedCategoryBasics.CartesianClosedMonoidalCategory*

begin

locale *rtscat* =
universe arr-type
for *arr-type* :: *'A itself*
begin

sublocale *One*: *one-arr-rts arr-type* ⟨*proof*⟩

interpretation *RTSx*: *rtscatx arr-type* ⟨*proof*⟩
interpretation *RTSx*: *elementary-category-with-binary-products*
RTSx.hcomp RTSx.p_0 *RTSx.*p_1
⟨*proof*⟩
interpretation RTS_S: *subcategory RTSx.hcomp RTSx.sta*
⟨*proof*⟩

type-synonym *'a arr* = *'a rtscatx.arr*

definition *comp* :: *'A arr comp* (**infixr** · *53*)
where *comp* ≡ *subcategory.comp RTSx.hcomp RTSx.sta*

sublocale *category comp*
⟨*proof*⟩

notation *in-hom* («- : - → -»)

lemma *ide-iff-RTS-obj*:
shows *ide a* ⟷ *RTSx.obj a*
⟨*proof*⟩

lemma *arr-iff-RTS-sta*:
shows *arr f* ⟷ *RTSx.sta f*
⟨*proof*⟩

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* ≡ *extensional-rts A* ∧ *small-rts A*

definition *mkide* :: *'A resid* ⇒ *'A arr*
where *mkide* ≡ *RTSx.mkobj*

definition *mkarr* :: *'A resid* ⇒ *'A resid* ⇒ (*'A* ⇒ *'A*) ⇒ *'A arr*
where *mkarr* ≡ *RTSx.mksta*

definition *Rts* :: *'A arr* ⇒ *'A resid*
where *Rts* ≡ *RTSx.Dom*

abbreviation *Dom* :: *'A arr* ⇒ *'A resid*
where *Dom t* ≡ *Rts* (*dom t*)

abbreviation *Cod* :: *'A arr* ⇒ *'A resid*
where *Cod t* ≡ *Rts* (*cod t*)

definition *Map* :: *'A arr* ⇒ *'A* ⇒ *'A*
where *Map* ≡ *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* ∈ *Collect Obj* → *Collect ide*
and *Rts* ∈ *Collect ide* → *Collect Obj*
and *Rts* (*mkide A*) = *A*
and *ide a* ⟹ *mkide* (*Rts a*) = *a*
and *bij-betw mkide* (*Collect Obj*) (*Collect ide*)

and *bij-betw Rts* (*Collect ide*) (*Collect Obj*)
⟨*proof*⟩

lemma *arrD*:
assumes *arr f*
shows *Obj* (*Rts* (*dom f*)) **and** *Obj* (*Rts* (*cod f*))
and *simulation* (*Rts* (*dom f*)) (*Rts* (*cod f*)) (*Map f*)
⟨*proof*⟩

lemma *arr-mkarr* [*intro*, *simp*]:
assumes *Obj A* **and** *Obj B* **and** *simulation A B F*
shows *arr* (*mkarr A B F*)
⟨*proof*⟩

lemma $arrI_{RTSC}$:
assumes *f* ∈ *mkarr A B ‘ Collect* (*simulation A B*)
and *Obj A* **and** *Obj B*
shows *arr f*
⟨*proof*⟩

lemma *Dom-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F*)
shows *Dom* (*mkarr A B F*) = *A*
⟨*proof*⟩

lemma *Cod-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F*)
shows *Cod* (*mkarr A B F*) = *B*
⟨*proof*⟩

lemma *Map-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F*)
shows *Map* (*mkarr A B F*) = *F*
⟨*proof*⟩

lemma *mkarr-Map* [*simp*]:
assumes *Obj A* **and** *Obj B* **and** *t* ∈ {*t*. «*t* : *mkide A* → *mkide B*»}
shows *mkarr A B* (*Map t*) = *t*
⟨*proof*⟩

lemma *dom-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F*)
shows *dom* (*mkarr A B F*) = *mkide A*
⟨*proof*⟩

lemma *cod-mkarr* [*simp*]:
assumes *arr* (*mkarr A B F*)
shows *cod* (*mkarr A B F*) = *mkide B*
⟨*proof*⟩

lemma *mkarr-in-hom* [*intro*]:
assumes *simulation A B F* **and** *Rts a = A* **and** *Rts b = B*
and *ide a* **and** *ide b*
shows «*mkarr A B F* : *a* → *b*»
⟨*proof*⟩

lemma *bij-mkarr*:
assumes *Obj A* **and** *Obj B*
shows *mkarr A B* ∈ *Collect* (*simulation A B*)
→ {*t*. «*t* : *mkide A* → *mkide B*»}
and *Map* ∈ {*t*. «*t* : *mkide A* → *mkide B*»} → *Collect* (*simulation A B*)
and *Map* (*mkarr A B F*) = *F*
and *t* ∈ {*t*. «*t* : *mkide A* → *mkide B*»} ⟹ *mkarr A B* (*Map t*) = *t*
and *bij-betw* (*mkarr A B*) (*Collect* (*simulation A B*))
{*t*. «*t* : *mkide A* → *mkide B*»}
and *bij-betw Map* {*t*. «*t* : *mkide A* → *mkide B*»}
(*Collect* (*simulation A B*))
⟨*proof*⟩

lemma *arr-eqI*:
assumes *par f g* **and** *Map f = Map g*
shows *f = g*
⟨*proof*⟩

lemma *Map-ide*:
assumes *ide a*
shows *Map a* = *I* (*Rts a*)
⟨*proof*⟩

lemma *Map-comp*:
assumes *seq g f*
shows *Map* (*g* · *f*) = *Map g* ∘ *Map f*
⟨*proof*⟩

lemma *comp-mkarr*:
assumes *arr* (*mkarr A B F*) **and** *arr* (*mkarr B C G*)
shows *mkarr B C G* · *mkarr A B F* = *mkarr A C* (*G* ∘ *F*)
⟨*proof*⟩

lemma *iso-char*:
shows *iso t* ⟷ *arr t* ∧ *invertible-simulation* (*Dom t*) (*Cod t*) (*Map t*)
⟨*proof*⟩

lemma *isomorphic-char*:
shows *isomorphic a b* ⟷
ide a ∧ *ide b* ∧ (∃ *F*. *invertible-simulation* (*Rts a*) (*Rts b*) *F*)
⟨*proof*⟩

4.4.1 Terminal Object

definition *one* (**1**)
where *one* ≡ *RTSx.one*
no-notation *RTSx.one* (**1**)

definition *trm*
where *trm* ≡ *RTSx.trm*

lemma *Rts-one* [*simp*]:
shows *Rts* **1** = *One.resid*
⟨*proof*⟩

lemma *mkide-One* [*simp*]:
shows *mkide One.resid* = **1**
⟨*proof*⟩

sublocale *elementary-category-with-terminal-object comp one trm*
⟨*proof*⟩

lemma *Map-trm*:
assumes *ide a*
shows *Map* (*trm a*) = *constant-simulation.map* (*Rts a*) *One.resid One.the-arr*
⟨*proof*⟩

lemma *terminal-char*:
shows *terminal x* ⟷ *ide x* ∧ (∃!*t*. *residuation.arr* (*Rts x*) *t*)
⟨*proof*⟩

4.4.2 Products

definition p_0 :: *'A arr* ⇒ *'A arr* ⇒ *'A arr*
where p_0 ≡ *RTSx.*p_0

definition p_1 :: *'A arr* ⇒ *'A arr* ⇒ *'A arr*
where p_1 ≡ *RTSx.*p_1

no-notation *RTSx.*p_0 ($\mathfrak{p}_0$[-, -])
no-notation *RTSx.*p_1 ($\mathfrak{p}_1$[-, -])
notation p_0 ($\mathfrak{p}_0$[-, -])
notation p_1 ($\mathfrak{p}_1$[-, -])

sublocale *elementary-cartesian-category comp* p_0 p_1 *one trm*
⟨*proof*⟩

notation *prod* (**infixr** ⊗ *51*)
notation *tuple* (⟨-, -⟩)
notation *dup* (d[-])
notation *assoc* (a[-, -, -])
notation *assoc'* (a^{-1}[-, -, -])

notation *lunit* (l[-])
notation *lunit'* (l^{-1}[-])
notation *runit* (r[-])
notation *runit'* (r^{-1}[-])

lemma *tuple-char*:
shows *tuple* = ($\lambda f\ g.$ *if span f g then RTSx.tuple f g else null*)
⟨*proof*⟩

lemma *prod-char*:
shows ($\otimes$) = ($\lambda f\ g.$ *if arr f* $\wedge$ *arr g then RTSx.prod f g else null*)
⟨*proof*⟩

definition *Pack* :: *'A arr* $\Rightarrow$ *'A arr* $\Rightarrow$ *'A* $\times$ *'A* $\Rightarrow$ *'A*
where *Pack* $\equiv$ *RTSx.Pack*

definition *Unpack* :: *'A arr* $\Rightarrow$ *'A arr* $\Rightarrow$ *'A* $\Rightarrow$ *'A* $\times$ *'A*
where *Unpack* $\equiv$ *RTSx.Unpack*

lemma *inverse-simulations-Pack-Unpack*:
assumes *ide a* **and** *ide b*
shows *inverse-simulations* (*Rts* (*a* $\otimes$ *b*)) (*product-rts.resid* (*Rts a*) (*Rts b*))
(*Pack a b*) (*Unpack a b*)
⟨*proof*⟩

lemma *simulation-Pack*:
assumes *ide a* **and** *ide b*
shows *simulation*
(*product-rts.resid* (*Rts a*) (*Rts b*)) (*Rts* (*a* $\otimes$ *b*)) (*Pack a b*)
⟨*proof*⟩

lemma *simulation-Unpack*:
assumes *ide a* **and** *ide b*
shows *simulation*
(*Rts* (*a* $\otimes$ *b*)) (*product-rts.resid* (*Rts a*) (*Rts b*)) (*Unpack a b*)
⟨*proof*⟩

lemma *Pack-o-Unpack*:
assumes *ide a* **and** *ide b*
shows *Pack a b* $\circ$ *Unpack a b* = *I* (*Rts* (*a* $\otimes$ *b*))
⟨*proof*⟩

lemma *Unpack-o-Pack*:
assumes *ide a* **and** *ide b*
shows *Unpack a b* $\circ$ *Pack a b* = *I* (*product-rts.resid* (*Rts a*) (*Rts b*))
⟨*proof*⟩

lemma *Pack-Unpack* [*simp*]:
assumes *ide a* **and** *ide b*

and *residuation.arr* (*Rts* (*a* ⊗ *b*)) *t*
shows *Pack a b* (*Unpack a b t*) = *t*
⟨*proof*⟩

lemma *Unpack-Pack* [*simp*]:
assumes *ide a* **and** *ide b*
and *residuation.arr* (*product-rts.resid* (*Rts a*) (*Rts b*)) *t*
shows *Unpack a b* (*Pack a b t*) = *t*
⟨*proof*⟩

lemma *Map-p*$_0$:
assumes *ide a* **and** *ide b*
shows *Map* $\mathfrak{p}_0[a, b]$ = *product-rts.P*$_0$ (*Rts a*) (*Rts b*) ∘ *Unpack a b*
⟨*proof*⟩

lemma *Map-p*$_1$:
assumes *ide a* **and** *ide b*
shows *Map* $\mathfrak{p}_1[a, b]$ = *product-rts.P*$_1$ (*Rts a*) (*Rts b*) ∘ *Unpack a b*
⟨*proof*⟩

lemma *Map-tuple*:
assumes «*f* : *x* → *a*» **and** «*g* : *x* → *b*»
shows *Map* ⟨*f*, *g*⟩ = *Pack a b* ∘ ⟨⟨*Map f*, *Map g*⟩⟩
⟨*proof*⟩

corollary *Map-dup*:
assumes *ide a*
shows *Map* d[*a*] = *Pack a a* ∘ ⟨⟨*Map a*, *Map a*⟩⟩
⟨*proof*⟩

proposition *Map-lunit*:
assumes *ide a*
shows *Map* l[*a*] = *product-rts.P*$_0$ (*Rts* **1**) (*Rts a*) ∘ *Unpack* **1** *a*
⟨*proof*⟩

proposition *Map-runit*:
assumes *ide a*
shows *Map* r[*a*] = *product-rts.P*$_1$ (*Rts a*) (*Rts* **1**) ∘ *Unpack a* **1**
⟨*proof*⟩

lemma *assoc-expansion*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows a[*a*, *b*, *c*] =
⟨$\mathfrak{p}_1[a, b]$ · $\mathfrak{p}_1[a \otimes b, c]$, ⟨$\mathfrak{p}_0[a, b]$ · $\mathfrak{p}_1[a \otimes b, c]$, $\mathfrak{p}_0[a \otimes b, c]$⟩ ⟩
⟨*proof*⟩

lemma *Unpack-naturality*:
assumes *arr f* **and** *arr g*
shows *Unpack* (*cod f*) (*cod g*) ∘ *Map* (*f* ⊗ *g*) =

product-simulation.map (*Rts* (*dom f*)) (*Rts* (*dom g*)) (*Map f*) (*Map g*) ∘
Unpack (*dom f*) (*dom g*)
⟨*proof*⟩

lemma *Map-prod*:
assumes *arr f* **and** *arr g*
shows *Map* (*f* ⊗ *g*) =
Pack (*cod f*) (*cod g*) ∘
product-simulation.map (*Rts* (*dom f*)) (*Rts* (*dom g*)) (*Map f*) (*Map g*) ∘
Unpack (*dom f*) (*dom g*)
⟨*proof*⟩

4.4.3 Exponentials

definition *exp* :: *'A arr* ⇒ *'A arr* ⇒ *'A arr*
where *exp* ≡ *RTSx.exp*

definition *eval* :: *'A arr* ⇒ *'A arr* ⇒ *'A arr*
where *eval* ≡ *RTSx.eval*

definition *curry* :: *'A arr* ⇒ *'A arr* ⇒ *'A arr* ⇒ *'A arr* ⇒ *'A arr*
where *curry* ≡ *RTSx.curry*

definition *uncurry* :: *'A arr* ⇒ *'A arr* ⇒ *'A arr* ⇒ *'A arr* ⇒ *'A arr*
where *uncurry* ≡ *RTSx.uncurry*

definition *Func* :: *'A arr* ⇒ *'A arr* ⇒ *'A* ⇒ (*'A*, *'A*) *exponential-rts.arr*
where *Func* ≡ *RTSx.Func*

definition *Unfunc* :: *'A arr* ⇒ *'A arr* ⇒ (*'A*, *'A*) *exponential-rts.arr* ⇒ *'A*
where *Unfunc* ≡ *RTSx.Unfunc*

lemma *inverse-simulations-Func-Unfunc*:
assumes *ide b* **and** *ide c*
shows *inverse-simulations*
(*exponential-rts.resid* (*Rts b*) (*Rts c*)) (*Rts* (*exp b c*))
(*Func b c*) (*Unfunc b c*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *Func-o-Unfunc*:
assumes *ide b* **and** *ide c*
shows *Func b c* ∘ *Unfunc b c* = *I* (*exponential-rts.resid* (*Rts b*) (*Rts c*))
⟨*proof*⟩

lemma *Unfunc-o-Func*:
assumes *ide b* **and** *ide c*
shows *Unfunc b c* ∘ *Func b c* = *I* (*Rts* (*exp b c*))
⟨*proof*⟩

lemma *Func-Unfunc* [*simp*]:
assumes *ide b* **and** *ide c*
and *residuation.arr* (*exponential-rts.resid* (*Rts b*) (*Rts c*)) *t*
shows *Func b c* (*Unfunc b c t*) = *t*
⟨*proof*⟩

lemma *Unfunc-Func* [*simp*]:
assumes *ide b* **and** *ide c*
and *residuation.arr* (*Rts* (*exp b c*)) *t*
shows *Unfunc b c* (*Func b c t*) = *t*
⟨*proof*⟩

lemma *Map-eval*:
assumes *ide b* **and** *ide c*
shows *Map* (*eval b c*) =
evaluation-map.map (*Rts b*) (*Rts c*) ∘
product-simulation.map (*Rts* (*exp b c*)) (*Rts b*) (*Func b c*) (*I* (*Rts b*)) ∘
Unpack (*exp b c*) *b*
⟨*proof*⟩

lemma *Map-curry*:
assumes *ide a* **and** *ide b* **and** «*f* : *a* ⊗ *b* → *c*»
shows *Map* (*curry a b c f*) =
Unfunc b c ∘
Currying.Curry3 (*Rts a*) (*Rts b*) (*Rts c*) (*Map f* ∘ *Pack a b*)
⟨*proof*⟩

lemma *Map-uncurry*:
assumes *ide b* **and** *ide c* **and** «*g* : *a* → *exp b c*»
shows *Map* (*uncurry a b c g*) =
Currying.Uncurry (*Rts a*) (*Rts b*) (*Rts c*) (*Func b c* ∘ *Map g*) ∘
Unpack a b
⟨*proof*⟩

4.4.4 Cartesian Closure

sublocale *elementary-cartesian-closed-category*

comp p_0 p_1 *one trm exp eval curry*
⟨*proof*⟩

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* ‹*Rts a*›
⟨*proof*⟩
interpretation *B*: *extensional-rts* ‹*Rts b*›
⟨*proof*⟩
interpretation *C*: *extensional-rts* ‹*Rts c*›
⟨*proof*⟩

interpretation *A*: *identity-simulation* ‹*Rts a*› ⟨*proof*⟩
interpretation *B*: *identity-simulation* ‹*Rts b*› ⟨*proof*⟩
interpretation *C*: *identity-simulation* ‹*Rts c*› ⟨*proof*⟩

interpretation *AxB*: *product-rts* ‹*Rts a*› ‹*Rts b*› ⟨*proof*⟩
interpretation *AxB-xC*: *product-rts AxB.resid* ‹*Rts c*› ⟨*proof*⟩
interpretation *BxC*: *product-rts* ‹*Rts b*› ‹*Rts c*› ⟨*proof*⟩
interpretation *Ax-BxC*: *product-rts* ‹*Rts a*› *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* ‹*Rts a*› ‹*Rts b*› ‹*Rts c*› ⟨*proof*⟩

interpretation *axb*: *extensional-rts* ‹*Rts* (*a* ⊗ *b*)›
⟨*proof*⟩
interpretation *bxc*: *extensional-rts* ‹*Rts* (*b* ⊗ *c*)›
⟨*proof*⟩
interpretation *axb-xc*: *extensional-rts* ‹*Rts* ((*a* ⊗ *b*) ⊗ *c*)›
⟨*proof*⟩
interpretation *ax-bxc*: *extensional-rts* ‹*Rts* (*a* ⊗ (*b* ⊗ *c*))›
⟨*proof*⟩

interpretation *PU-ab*: *inverse-simulations*
‹*Rts* (*a* ⊗ *b*)› *AxB.resid* ‹*Pack a b*› ‹*Unpack a b*›
⟨*proof*⟩
interpretation *PU-bc*: *inverse-simulations*
‹*Rts* (*b* ⊗ *c*)› *BxC.resid* ‹*Pack b c*› ‹*Unpack b c*›
⟨*proof*⟩

interpretation *Axbc*: *product-rts* ‹*Rts a*› ‹*Rts* (*b* ⊗ *c*)› ⟨*proof*⟩
interpretation *Axbc*: *identity-simulation Axbc.resid* ⟨*proof*⟩
interpretation *abxC*: *product-rts* ‹*Rts* (*a* ⊗ *b*)› ‹*Rts c*› ⟨*proof*⟩
interpretation *abxC*: *identity-simulation abxC.resid* ⟨*proof*⟩

interpretation *AxPack-bc*:
product-simulation ‹*Rts a*› *BxC.resid* ‹*Rts a*› ‹*Rts* (*b* ⊗ *c*)›
‹*I* (*Rts a*)› ‹*Pack b c*› ⟨*proof*⟩
interpretation *AxUnpack-bc*:
product-simulation ‹*Rts a*› ‹*Rts* (*b* ⊗ *c*)› ‹*Rts a*› *BxC.resid*
‹*I* (*Rts a*)› ‹*Unpack b c*› ⟨*proof*⟩
interpretation *Unpack-abxC*:
product-simulation ‹*Rts* (*a* ⊗ *b*)› ‹*Rts c*› *AxB.resid* ‹*Rts c*›
‹*Unpack a b*› ‹*I* (*Rts c*)› ⟨*proof*⟩

sublocale *PU-Axbc*: *inverse-simulations Axbc.resid Ax-BxC.resid*
AxPack-bc.map AxUnpack-bc.map
⟨*proof*⟩

The following shows that the simulation *Map* a[*a*, *b*, *c*] underlying an internal associator a[*a*, *b*, *c*] is transformed into a corresponding external associator *ASSOC.map* via invertible simulations that "unpack" product objects in **RTS** into corresponding product RTS's.

lemma *Unpack-o-Map-assoc*:
shows (*AxUnpack-bc.map* ∘ *Unpack a* (*b* ⊗ *c*)) ∘ *Map* a[*a*, *b*, *c*] =
ASSOC.map ∘ (*Unpack-abxC.map* ∘ *Unpack* (*a* ⊗ *b*) *c*)
⟨*proof*⟩

As a corollary, we obtain an explicit formula for *Map* a[*a*, *b*, *c*] in terms of the external associator *ASSOC.map* and packing and unpacking isomor-

phisms.

corollary *Map-assoc*:
shows *Map* a[*a*, *b*, *c*] =
(*Pack a* (*b* ⊗ *c*) ∘ *AxPack-bc.map*) ∘
ASSOC.map ∘
(*Unpack-abxC.map* ∘ *Unpack* (*a* ⊗ *b*) *c*)
⟨*proof*⟩

end

context *rtscat*
begin

proposition *Map-assoc*:
assumes *ide a* **and** *ide b* **and** *ide c*
shows *Map* a[*a*, *b*, *c*] =
Pack a (*b* ⊗ *c*) ∘
product-simulation.map (*Rts a*) (*product-rts.resid* (*Rts b*) (*Rts c*))
(*I* (*Rts a*)) (*Pack b c*) ∘
ASSOC.map (*Rts a*) (*Rts b*) (*Rts c*) ∘
(*product-simulation.map*
(*Rts* (*a* ⊗ *b*)) (*Rts c*) (*Unpack a b*) (*I* (*Rts c*)) ∘
Unpack (*a* ⊗ *b*) *c*)
⟨*proof*⟩

end

4.4.6 Compositors

Here we expose the relationship between the compositors internal to **RTS** (inherited from *closed-monoidal-category*) and those external to it (given by horizontal composition of simulations).

context *rtscat*
begin

sublocale *CMC*: *cartesian-monoidal-category comp Prod* α ι
⟨*proof*⟩

sublocale *ECMC*: *elementary-closed-monoidal-category comp Prod* α ι
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* ‹*Rts a*›
⟨*proof*⟩
interpretation *B*: *extensional-rts* ‹*Rts b*›
⟨*proof*⟩
interpretation *C*: *extensional-rts* ‹*Rts c*›
⟨*proof*⟩
interpretation *AxB*: *product-rts* ‹*Rts a*› ‹*Rts b*› ⟨*proof*⟩
interpretation *BxC*: *product-rts* ‹*Rts b*› ‹*Rts c*› ⟨*proof*⟩
interpretation *AB*: *exponential-rts* ‹*Rts a*› ‹*Rts b*› ⟨*proof*⟩
interpretation *BC*: *exponential-rts* ‹*Rts b*› ‹*Rts c*› ⟨*proof*⟩
interpretation *AC*: *exponential-rts* ‹*Rts a*› ‹*Rts c*› ⟨*proof*⟩
interpretation *ABxA*: *product-rts AB.resid* ‹*Rts a*› ⟨*proof*⟩
interpretation *BCxAB*: *product-rts BC.resid AB.resid* ⟨*proof*⟩
interpretation *BCxAB*: *extensional-rts BCxAB.resid*
⟨*proof*⟩
interpretation *BCxAB-x-A*: *product-rts BCxAB.resid* ‹*Rts a*› ⟨*proof*⟩

interpretation *ab*: *extensional-rts* ‹*EXP a b*›
⟨*proof*⟩
interpretation *bc*: *extensional-rts* ‹*EXP b c*›
⟨*proof*⟩
interpretation *bcxab*: *product-of-extensional-rts* ‹*EXP b c*› ‹*EXP a b*› ⟨*proof*⟩
interpretation *abxA*: *product-rts* ‹*EXP a b*› ‹*Rts a*› ⟨*proof*⟩
interpretation *bcxB*: *product-rts* ‹*EXP b c*› ‹*Rts b*› ⟨*proof*⟩
interpretation *bc-x-abxA*: *product-rts* ‹*EXP b c*› *abxA.resid* ⟨*proof*⟩
interpretation *bcxab-x-A*: *product-rts bcxab.resid* ‹*Rts a*› ⟨*proof*⟩

interpretation *ASSOC*: *ASSOC BC.resid AB.resid* ‹*Rts a*› ⟨*proof*⟩
interpretation *COMP*: *COMP* ‹*Rts a*› ‹*Rts b*› ‹*Rts c*› ⟨*proof*⟩

interpretation *Assoc-abc*: *Association arr-type a b c*
⟨*proof*⟩
interpretation *Assoc-bc-ab-a*: *Association arr-type* ‹*exp b c*› ‹*exp a b*› *a*
⟨*proof*⟩

interpretation *Func-Unfunc-ab*: *inverse-simulations AB.resid* ‹*EXP a b*›
‹*Func a b*› ‹*Unfunc a b*›
⟨*proof*⟩
interpretation *Func-Unfunc-bc*: *inverse-simulations BC.resid* ‹*EXP b c*›

$\langle Func\ b\ c\rangle\ \langle Unfunc\ b\ c\rangle$
$\langle proof\rangle$
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* ‹*TYPE*(*V*)›
⟨*proof*⟩

interpretation *RTS*: *rtscat* ‹*TYPE*(*V*)›
⟨*proof*⟩

end

Chapter 5

RTS-Enriched Categories

5.1 RTS-Enriched Categories

The category **RTS** is cartesian closed, hence monoidal closed. This implies that each hom-set of **RTS** itself carries the structure of an RTS, so that **RTS** becomes a category "enriched in itself". In this section we show that RTS-categories are essentially the same thing as categories enriched in **RTS**, and that the RTS-category $\mathbf{RTS}^\dagger$ is equivalent to the RTS-category determined by **RTS** regarded as a category enriched in itself. Thus, the complete structure of the RTS-category $\mathbf{RTS}^\dagger$ is already determined by its ordinary subcategory **RTS**.

theory *RTSEnrichedCategory*
imports *RTSCatx RTSCat EnrichedCategoryBasics.CartesianClosedMonoidalCategory*
EnrichedCategoryBasics.EnrichedCategory
begin

context *rtscat*
begin

sublocale *CMC*: *cartesian-monoidal-category comp Prod* α ι
⟨*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* ⊗ *g*
⟨*proof*⟩

The situation with tupling and "duplicators" is similar.

lemma *tuple-agreement*:
assumes *span f g*
shows *CMC.tuple f g* = ⟨*f*, *g*⟩
⟨*proof*⟩

lemma *dup-agreement*:
assumes *arr f*
shows *CMC.dup f* = *dup f*
⟨*proof*⟩

sublocale *elementary-cartesian-closed-monoidal-category*
comp Prod α ι *exp eval curry*
⟨*proof*⟩

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* =
⟨𝔭₁[*a*, *b*] · 𝔭₁[*a* ⊗ *b*, *c*], ⟨𝔭₀[*a*, *b*] · 𝔭₁[*a* ⊗ *b*, *c*], 𝔭₀[*a* ⊗ *b*, *c*]⟩ ⟩
⟨*proof*⟩

lemma *extends-to-enriched-category*:
shows *enriched-category comp Prod* α ι
(*Collect ide*) *exp ECMC.Id ECMC.Comp*
⟨*proof*⟩

end

locale *rts-enriched-category* =
universe arr-type +
RTS: *rtscat arr-type* +
*enriched-category RTS.comp RTS.Prod RTS.*α *RTS.*ι *Obj Hom Id Comp*

for *arr-type* :: *'A itself*
and *Obj* :: *'O set*
and *Hom* :: *'O* ⇒ *'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* ‹$HOM_{EC}\ a\ b$›
⟨*proof*⟩

sublocale *small-rts* ‹$HOM_{EC}\ a\ b$›
⟨*proof*⟩

end

locale *rts-enriched-functor* =
RTS: *rtscat arr-type* +
A: *rts-enriched-category arr-type* Obj_A Hom_A Id_A $Comp_A$ +
B: *rts-enriched-category arr-type* Obj_B Hom_B Id_B $Comp_B$ +
enriched-functor RTS.comp RTS.Prod RTS.α RTS.ι
for *arr-type* :: *'A itself*
begin

lemma *is-local-simulation*:
assumes $a \in Obj_A$ **and** $b \in Obj_A$
shows *simulation* $(A.HOM_{EC}\ a\ b)\ (B.HOM_{EC}\ (F_o\ a)\ (F_o\ b))$
$(RTS.Map\ (F_a\ a\ b))$
⟨*proof*⟩

end

locale *fully-faithful-rts-enriched-functor* =
rts-enriched-functor +
fully-faithful-enriched-functor RTS.comp RTS.Prod RTS.α RTS.ι

5.2 RTS-Enriched Categories induce RTS-Categories

Here we show that every RTS-enriched category determines a corresponding RTS-category. This is done by combining the RTS's underlying the homs of the RTS-enriched category, forming a global RTS that provides the vertical structure of the RTS-category. The composition operation of the RTS-enriched category is used to obtain the horizontal structure.

```
locale rts-category-of-enriched-category =
  universe arr-type +
  RTS: rtscat arr-type +
  rts-enriched-category arr-type Obj Hom Id Comp
for arr-type :: 'A itself
and Obj :: 'O set
and Hom :: 'O ⇒ 'O ⇒ 'A rtscatx.arr
and Id :: 'O ⇒ 'A rtscatx.arr
and Comp :: 'O ⇒ 'O ⇒ 'O ⇒ 'A rtscatx.arr
begin

  notation RTS.in-hom    («- : - → -»)
  notation RTS.prod      (infixr ⊗ 51)
  notation RTS.one       (1)
  notation RTS.assoc     (a[-, -, -])
  notation RTS.lunit     (l[-])
  notation RTS.runit     (r[-])
```

Here we define the "global RTS", obtained as the disjoint union of all the hom-RTS's. Note that types 'O and 'A are fixed in the current context: type 'O is the type of "objects" of the given RTS-enriched category, and type 'A is the type of the universe that underlies the base category *RTS*.

```
  datatype ('o, 'a) arr =
    Null
  | MkArr 'o 'o 'a

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

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

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

  abbreviation Arr :: ('O, 'A) arr ⇒ bool
  where Arr ≡ λt. t ≠ Null ∧ Dom t ∈ Obj ∧ Cod t ∈ Obj ∧
                  residuation.arr (HOM_EC (Dom t) (Cod t)) (Trn t)
```

abbreviation *Ide* :: (*'O*, *'A*) *arr* ⇒ *bool*
where *Ide* ≡ λ*t*. *t* ≠ *Null* ∧ *Dom t* ∈ *Obj* ∧ *Cod t* ∈ *Obj* ∧
residuation.ide (*HOM*$_{EC}$ (*Dom t*) (*Cod t*)) (*Trn t*)

definition *Con* :: (*'O*, *'A*) *arr* ⇒ (*'O*, *'A*) *arr* ⇒ *bool*
where *Con t u* ≡ *Arr t* ∧ *Arr u* ∧ *Dom t* = *Dom u* ∧ *Cod t* = *Cod u* ∧
residuation.con (*HOM*$_{EC}$ (*Dom t*) (*Cod t*)) (*Trn t*) (*Trn u*)

The global residuation is obtained by combining the local residuations of each of the hom-RTS's.

fun *resid* (**infix** \ *70*)
where *resid Null u* = *Null*
| *resid t Null* = *Null*
| *resid t u* = (*if Con t u*
then MkArr (*Dom t*) (*Cod t*)
(*HOM*$_{EC}$ (*Dom t*) (*Cod t*) (*Trn t*) (*Trn u*))
else Null)

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

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

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*
⟨*proof*⟩

lemma *ConE* [*elim*]:
assumes *Con t u*
and ⟦*Arr t*; *Arr u*; *Dom t* = *Dom u*; *Cod t* = *Cod u*;
residuation.con (*HOM*$_{EC}$ (*Dom t*) (*Cod t*)) (*Trn t*) (*Trn u*)⟧ ⟹ *T*
shows *T*
⟨*proof*⟩

lemma *Con-sym*:
assumes *Con t u*
shows *Con u t*
⟨*proof*⟩

lemma *resid-ne-Null-imp-Con*:
assumes *t* \ *u* ≠ *Null*
shows *Con t u*
⟨*proof*⟩

sublocale V: *residuation resid*
⟨*proof*⟩

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

lemma *con-char*:
shows $t \frown u \longleftrightarrow Con\ t\ u$
⟨*proof*⟩

lemma *arr-char*:
shows $V.arr\ t \longleftrightarrow Arr\ t$
⟨*proof*⟩

lemma *ide-char*:
shows $V.ide\ t \longleftrightarrow Ide\ t$
⟨*proof*⟩

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

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*⟩

lemma *Trn-resid* [*simp*]:
assumes $t \frown u$
shows $Trn\ (t \setminus u) = HOM_{EC}\ (Dom\ t)\ (Cod\ t)\ (Trn\ t)\ (Trn\ u)$
⟨*proof*⟩

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 $V.trg\ t = (if\ V.arr\ t$
$\quad then\ MkArr\ (Dom\ t)\ (Cod\ t)$
$\quad\quad (residuation.trg\ (HOM_{EC}\ (Dom\ t)\ (Cod\ t))\ (Trn\ t))$
$\quad else\ Null)$
⟨*proof*⟩

sublocale *rts resid*
⟨*proof*⟩

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* ∈ *Obj* **and** *b* ∈ *Obj*
and *residuation.arr* (*HOM*$_{EC}$ *a b*) *t*
shows *V.arr* (*MkArr a b t*)
⟨*proof*⟩

lemma *arr-eqI*:
assumes *t* ≠ *V.null* **and** *u* ≠ *V.null*
and *Dom t* = *Dom u* **and** *Cod t* = *Cod u* **and** *Trn t* = *Trn u*
shows *t* = *u*
⟨*proof*⟩

lemma *MkArr-Trn*:
assumes *V.arr t*
shows *t* = *MkArr* (*Dom t*) (*Cod t*) (*Trn t*)
⟨*proof*⟩

lemma *src-char*:
shows *V.src t* = (*if V.arr t*
then MkArr (*Dom t*) (*Cod t*)
(*weakly-extensional-rts.src*
(*HOM*$_{EC}$ (*Dom t*) (*Cod t*)) (*Trn t*))
else Null)
⟨*proof*⟩

Here we use the composition operation of the original RTS-enriched category to define horizontal composition of transitions of the global RTS. Note that a pair of transitions (which comprise a transition of a product RTS) must be "packed" into a single transition of the RTS underlying a product object, before the composition operation can be applied.

definition *hcomp* (**infixr** ⋆ *53*)
where *t* ⋆ *u* ≡
if V.arr t ∧ *V.arr u* ∧ *Dom t* = *Cod u*
then MkArr (*Dom u*) (*Cod t*)
(*RTS.Map* (*Comp* (*Dom u*) (*Cod u*) (*Cod t*))
(*RTS.Pack* (*Hom* (*Dom t*) (*Cod t*))
(*Hom* (*Dom u*) (*Cod u*))
(*Trn t*, *Trn u*)))

else V.null

lemma *arr-hcomp*:
assumes *V.arr t* **and** *V.arr u* **and** *Dom t = Cod u*
shows *V.arr (t ⋆ u)*
⟨*proof*⟩

lemma *Dom-hcomp* [*simp*]:
assumes *V.arr t* **and** *V.arr u* **and** *Dom t = Cod u*
shows *Dom (t ⋆ u) = Dom u*
⟨*proof*⟩

lemma *Cod-hcomp* [*simp*]:
assumes *V.arr t* **and** *V.arr u* **and** *Dom t = Cod u*
shows *Cod (t ⋆ u) = Cod t*
⟨*proof*⟩

lemma *Trn-hcomp* [*simp*]:
assumes *V.arr t* **and** *V.arr u* **and** *Dom t = Cod u*
shows *Trn (t ⋆ u) =*
RTS.Map (Comp (Dom u) (Cod u) (Cod t))
(RTS.Pack (Hom (Cod u) (Cod t)) (Hom (Dom u) (Cod u))
(Trn t, Trn u))
⟨*proof*⟩

lemma *hcomp-Null* [*simp*]:
shows *t ⋆ Null = Null* **and** *Null ⋆ 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 ⋆ u ≠ V.null ⟷ V.arr t ∧ V.arr u ∧ Dom t = Cod u*
⟨*proof*⟩

definition *horizontal-unit*
where *horizontal-unit a ≡*
V.arr a ∧ Dom a = Cod a ∧
(∀ t. (V.arr t ∧ Dom t = Cod a ⟶ t ⋆ a = t) ∧
(V.arr t ∧ Dom a = Cod t ⟶ a ⋆ 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 «$Id\ A : One \to Hom\ A\ A$» 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 $H.dom =$
$(\lambda t.\ if\ H.arr\ t$
$then\ MkArr\ (Dom\ t)\ (Dom\ t)$
$(RTS.Map\ (Id\ (Dom\ t))\ RTS.One.the\text{-}arr)$

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* =
(λ*t*. *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*: *fibered-product-rts resid resid resid H.dom H.cod* ⟨*proof*⟩

sublocale *H*: *simulation RR.resid resid*
‹λ*t*. *if RR.arr t then fst t* ⋆ *snd t else V.null*›

⟨*proof*⟩

lemma *simulation-hcomp*:
shows *simulation RR.resid resid*
(λ*t*. *if RR.arr t then fst t ⋆ 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 *null-coincidence* [*simp*]:
shows *H.null* = *V.null*
⟨*proof*⟩

lemma *arr-coincidence* [*simp*]:
shows *H.arr* = *V.arr*
⟨*proof*⟩

lemma *dom-src* [*simp*]:
shows *H.dom* (*V.src t*) = *H.dom t*
⟨*proof*⟩

lemma *src-dom* [*simp*]:
shows *V.src* (*H.dom t*) = *H.dom t*
⟨*proof*⟩

lemma *small-homs*:
shows *small* (*H.hom a b*)
⟨*proof*⟩

Note that the arrow type of the RTS-category given by the following is (*'O*, *'A*) *arr*, where 'A is the type of the universe underlying the category RTS and 'O is the type of objects of the context RTS-enriched category.

If we start with an RTS-enriched category having object type 'O, then we construct an RTS-category having arrow type $('O, 'A)\ arr$, and then we try to go back to an RTS-enriched category, the hom-RTS's will have arrow type $('O, 'A)\ arr$, not $'A$ as required for them to determine objects of *RTS*. So to show that the passage between RTS-categories and RTS-enriched categories is an equivalence, we will need to be able to reduce the type of the hom-RTS's from $('O, 'A)\ arr$ back to $'A$.

sublocale *rts-category resid hcomp*
⟨*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 $('O, 'A)\ arr$, 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* ⇒ *'O* ⇒ *'A rtscatx.arr*
and *Id* :: *'O* ⇒ *'A rtscatx.arr*
and *Comp* :: *'O* ⇒ *'O* ⇒ *'O* ⇒ *'A rtscatx.arr* +
assumes *small-Obj*: *small Obj*
and *inj-Obj-to-arr*: ∃φ :: *'O* ⇒ *'A*. *inj-on* φ *Obj*
begin

We will use R to refer to the RTS constructed from the given enriched category.

abbreviation *R* :: $('O, 'A)\ arr\ resid$
where *R* ≡ *resid*

The locale assumptions are sufficient to allow us to uniquely encode each element of $Collect\ arr \cup \{null\}$ as single element of $'A$.

lemma *ex-arrow-injection*:
shows $\exists\, i :: ('O, 'A)\ arr \Rightarrow 'A.\ inj\text{-}on\ i\ (Collect\ arr \cup \{null\})$
⟨*proof*⟩

lemma *bij-betw-Obj-horiz-ide*:
shows *bij-betw mkobj Obj* (*Collect H.ide*)
⟨*proof*⟩

lemma *ex-isomorphic-image-rts*:
shows $\exists\, R'\ (UP :: 'A \Rightarrow ('O, 'A)\ arr)\ (DN :: ('O, 'A)\ arr \Rightarrow 'A).$
small-rts $R' \wedge$ *extensional-rts* $R' \wedge$ *inverse-simulations* $R\ R'\ UP\ DN$
⟨*proof*⟩

We now choose some RTS with the properties asserted by the previous lemma, along with the invertible simulations that relate it to R.

definition $R' :: 'A\ resid$
where $R' \equiv SOME\ R'.\ \exists\, UP\ DN.$ *small-rts* $R' \wedge$ *extensional-rts* $R' \wedge$
inverse-simulations resid $R'\ UP\ DN$

definition $UP :: 'A \Rightarrow ('O, 'A)\ arr$
where $UP \equiv SOME\ UP.\ \exists\, DN.$ *small-rts* $R' \wedge$ *extensional-rts* $R' \wedge$
inverse-simulations resid $R'\ UP\ DN$

definition $DN :: ('O, 'A)\ arr \Rightarrow 'A$
where $DN \equiv SOME\ DN.$ *small-rts* $R' \wedge$ *extensional-rts* $R' \wedge$
inverse-simulations resid $R'\ UP\ DN$

lemma R'*-prop*:
shows $\exists\, UP\ DN.$ *small-rts* $R' \wedge$ *extensional-rts* $R' \wedge$
inverse-simulations $R\ R'\ UP\ DN$
⟨*proof*⟩

sublocale R': *extensional-rts* R'
⟨*proof*⟩
sublocale R': *small-rts* R'
⟨*proof*⟩

lemma *extensional-rts-*R':
shows *extensional-rts* R'
⟨*proof*⟩

lemma *small-rts-*R':
shows *small-rts* R'
⟨*proof*⟩

sublocale *UP-DN*: *inverse-simulations* $R\ R'\ UP\ DN$
⟨*proof*⟩

lemma *inverse-simulations-UP-DN*:
shows *inverse-simulations resid R′ UP DN*
⟨*proof*⟩

lemma *R′-src-char*:
shows *R′.src* = *DN* ∘ *src* ∘ *UP*
⟨*proof*⟩

lemma *R′-trg-char*:
shows *R′.trg* = *DN* ∘ *trg* ∘ *UP*
⟨*proof*⟩

We transport the horizontal composition ($\star$) to R' via the isomorphisms *UP* and *DN*.

abbreviation *hcomp′* :: *'A resid* (**infixr** ⋆′ *53*)
where *t* ⋆′ *u* ≡ *DN* (*UP t* ⋆ *UP u*)

interpretation *H′*: *Category.partial-magma hcomp′*
⟨*proof*⟩

lemma *H′-null-char*:
shows *H′.null* = *DN null*
⟨*proof*⟩

interpretation *H′*: *partial-composition* ‹λ*t u*. *DN* (*hcomp* (*UP t*) (*UP u*))›
⟨*proof*⟩

lemma *H′-ide-char*:
shows *H′.ide t* ⟷ *H.ide* (*UP t*)
⟨*proof*⟩

lemma *H′-domains-char*:
shows *H′.domains t* = *DN* ‘ *H.domains* (*UP t*)
⟨*proof*⟩

lemma *H′-codomains-char*:
shows *H′.codomains t* = *DN* ‘ *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* ⟷ *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* ∘ *H.dom* ∘ *UP*
⟨*proof*⟩

lemma *H′-cod-char*:
shows *H′.cod* = *DN* ∘ *H.cod* ∘ *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* ‘ *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* ⟷ *V.con* (*UP t*) (*UP u*)
⟨*proof*⟩

sublocale *R′R′*: *fibered-product-rts R′ R′ R′ H′.dom H′.cod* ⟨*proof*⟩

sublocale *H′*: *simulation R′R′.resid R′*
‹λ*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* Obj_A Hom_A Id_A $Comp_A$ +
  B: *rts-enriched-category arr-type* Obj_B Hom_B Id_B $Comp_B$ +
  EF: *rts-enriched-functor*
      Obj_A Hom_A Id_A $Comp_A$ Obj_B Hom_B Id_B $Comp_B$ F_o F_a
for $Obj_A :: 'a\ set$
and $Hom_A :: 'a \Rightarrow 'a \Rightarrow 'A\ rtscatx.arr$
and $Id_A :: 'a \Rightarrow 'A\ rtscatx.arr$
and $Comp_A :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow 'A\ rtscatx.arr$
and $Obj_B :: 'b\ set$
and $Hom_B :: 'b \Rightarrow 'b \Rightarrow 'A\ rtscatx.arr$
and $Id_B :: 'b \Rightarrow 'A\ rtscatx.arr$
and $Comp_B :: 'b \Rightarrow 'b \Rightarrow 'b \Rightarrow 'A\ rtscatx.arr$
and $F_o :: 'a \Rightarrow 'b$
and $F_a :: 'a \Rightarrow 'a \Rightarrow 'A\ rtscatx.arr$
begin

  interpretation *A*: *rts-category-of-enriched-category*
                  arr-type Obj_A Hom_A Id_A $Comp_A$
    ⟨*proof*⟩
  interpretation *B*: *rts-category-of-enriched-category*
                  arr-type Obj_B Hom_B Id_B $Comp_B$
    ⟨*proof*⟩

  definition *F*
  where $F\ t \equiv$ *if residuation.arr A.resid t*
              then B.MkArr $(F_o\ (A.Dom\ t))\ (F_o\ (A.Cod\ t))$
                      $(RTS.Map\ (F_a\ (A.Dom\ t)\ (A.Cod\ t))\ (A.Trn\ t))$
              else ResiduatedTransitionSystem.partial-magma.null B.resid

  lemma *preserves-arr*:
  assumes $A.H.arr\ f$
  shows $B.H.arr\ (F\ f)$
  ⟨*proof*⟩

  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* (**infix** \ *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* («- : - → -»)
  no-notation *RTS.prod* (**infixr** ⊗ *51*)

  notation *RTS.in-hom* («- : - → -»)
  notation *RTS.CMC.tensor* (**infixr** ⊗ *51*)
  notation *RTS.CMC.unity* (**1**)
  notation *RTS.CMC.assoc* (a[-, -, -])
  notation *RTS.CMC.lunit* (l[-])
  notation *RTS.CMC.runit* (r[-])

  abbreviation *Obj*
  where *Obj* ≡ *Collect H.ide*

  definition *Hom*
  where *Hom a b* ≡
      if a ∈ *Obj* ∧ *b* ∈ *Obj then RTS.mkide* (*HOM a b*) *else RTS.null*

  definition *Id*
  where *Id a* ≡
      RTS.mkarr RTS.One.resid (*RTS.Rts* (*Hom a a*))
       (λ*t*. *if RTS.One.arr t*
           then a
           else ResiduatedTransitionSystem.partial-magma.null (*HOM a a*))

  definition *Comp*
  where *Comp a b c* ≡
      RTS.mkarr
       (*RTS.Rts* (*Hom b c* ⊗ *Hom a b*))

$(RTS.Rts\ (Hom\ a\ c))$
$(\lambda t.\ (\lambda x.\ fst\ x \star snd\ x)\ (RTS.Unpack\ (Hom\ b\ c)\ (Hom\ a\ b)\ t))$

lemma *ide-Hom* [*intro*, *simp*]:
assumes $a \in Obj$ **and** $b \in Obj$
shows $RTS.ide\ (Hom\ a\ b)$
⟨*proof*⟩

lemma
assumes $a \in Obj$ **and** $b \in Obj$
shows *HOM-null-char*: $ResiduatedTransitionSystem.partial\text{-}magma.null$
$(RTS.Rts\ (Hom\ a\ b)) =$
$null$
and *HOM-arr-char*:
$residuation.arr\ (RTS.Rts\ (Hom\ a\ b))\ t \longleftrightarrow H.in\text{-}hom\ t\ a\ b$
⟨*proof*⟩

lemma *Id-in-hom* [*intro*]:
assumes $a \in Obj$
shows «$Id\ a : \mathbf{1} \to Hom\ a\ a$»
⟨*proof*⟩

lemma *Id-simps* [*simp*]:
assumes $a \in Obj$
shows $RTS.arr\ (Id\ a)$
and $RTS.dom\ (Id\ a) = \mathbf{1}$
and $RTS.cod\ (Id\ a) = Hom\ a\ a$
⟨*proof*⟩

lemma *Comp-in-hom* [*intro*, *simp*]:
assumes $a \in Obj$ **and** $b \in Obj$ **and** $c \in Obj$
shows «$Comp\ a\ b\ c : Hom\ b\ c \otimes Hom\ a\ b \to Hom\ a\ c$»
⟨*proof*⟩

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 $RTS.cod\ (Comp\ a\ b\ c) = Hom\ a\ c$
⟨*proof*⟩

lemma *Map-Comp-Pack*:
assumes $a \in Obj$ **and** $b \in Obj$ **and** $c \in Obj$
and $residuation.arr$
$(product\text{-}rts.resid\ (RTS.Rts\ (Hom\ b\ c))\ (RTS.Rts\ (Hom\ a\ b)))\ x$
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** $\backslash_A$ *70*)
and $comp_A$ :: $'A$ *comp* (**infixr** $\star_A$ *53*)
and $resid_B$ :: $'A$ *resid* (**infix** $\backslash_B$ *70*)
and $comp_B$ :: $'A$ *comp* (**infixr** $\star_B$ *53*)
and *F* :: $'A \Rightarrow 'A$
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* $resid_A\ (\lambda t.\ A.H.in\text{-}hom\ t\ a\ b) = A.HOM_{EC}\ a\ b$
and *sub-rts.resid* $resid_B\ (\lambda t.\ B.H.in\text{-}hom\ t\ (F_o\ a)\ (F_o\ b)) =$
$B.HOM_{EC}\ (F_o\ a)\ (F_o\ b)$
⟨*proof*⟩

sublocale *rts-enriched-functor*
‹*Collect A.H.ide*› *A.Hom A.Id A.Comp*
‹*Collect B.H.ide*› *B.Hom B.Id B.Comp*
$F_o\ F_a$
⟨*proof*⟩

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 ⟨*proof*⟩

no-notation *RTS.prod* (**infixr** $\otimes$ *51*)

interpretation *Trn*: *simulation RC.resid resid*
‹$\lambda t.$ *if RC.arr t then RC.Trn t else null*›
⟨*proof*⟩

interpretation *MkArr*: *simulation resid RC.resid*
‹λ*t*. *if arr t then RC.MkArr* (*H.dom t*) (*H.cod t*) *t*
else RC.null›
⟨*proof*⟩

interpretation *Trn-MkArr*: *inverse-simulations resid RC.resid*
‹λ*t*. *if RC.arr t then RC.Trn t else null*›
‹λ*t*. *if arr t then RC.MkArr* (*H.dom t*) (*H.cod t*) *t*
else RC.null›
⟨*proof*⟩

lemma *inverse-simulations-Trn-MkArr*:
shows *inverse-simulations resid RC.resid*
(λ*t*. *if RC.arr t then RC.Trn t else null*)
(λ*t*. *if arr t then RC.MkArr* (*H.dom t*) (*H.cod t*) *t else RC.null*)
⟨*proof*⟩

interpretation *Trn*: *functor RC.hcomp hcomp*
‹λ*t*. *if RC.arr t then RC.Trn t else null*›
⟨*proof*⟩

interpretation *MkArr*: *functor hcomp RC.hcomp*
‹λ*t*. *if arr t then RC.MkArr* (*H.dom t*) (*H.cod t*) *t*
else RC.null›
⟨*proof*⟩

interpretation *Trn-MkArr*: *inverse-functors hcomp RC.hcomp*
‹λ*t*. *if RC.arr t then RC.Trn t else null*›
‹λ*t*. *if arr t*
then RC.MkArr (*H.dom t*) (*H.cod t*) *t*
else RC.null›
⟨*proof*⟩

lemma *inverse-functors-Trn-MkArr*:
shows *inverse-functors hcomp RC.hcomp*
(λ*t*. *if RC.arr t then RC.Trn t else null*)
(λ*t*. *if arr t then RC.MkArr* (*H.dom t*) (*H.cod t*) *t else RC.null*)
⟨*proof*⟩

proposition *induces-rts-category-isomorphism*:
shows *rts-category-isomorphism resid hcomp RC.resid RC.hcomp*
(λ*t*. *if arr t then RC.MkArr* (*H.dom t*) (*H.cod t*) *t else RC.null*)
⟨*proof*⟩

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'*
⟨*proof*⟩

interpretation R'': *enriched-category-of-rts-category arr-type* R' *hcomp'*
⟨*proof*⟩

Our objective is now to construct a fully faithful RTS-enriched functor (F_o, F_a), from the originally given RTS-enriched category (*Obj*, *Hom*, *Id*, *Comp*) to the newly constructed RTS-category ($R''.Obj$ $R''.Hom$ $R''.Id$ $R''.Comp$). Note that this makes sense, because, due to the type reduction from R' to R'', we have arranged for the base category of ($R''.Obj$ $R''.Hom$ $R''.Id$ $R''.Comp$) to be the same category **RTS** as that of the originally given (*Obj*, *Hom*, *Id*, *Comp*). The object map F_o will take $a \in Obj :: 'O\ set$ to $DN\ (MkArr\ a\ a\ (RTS.Map\ (Id\ a)\ one)) \in R''.Obj :: 'A\ set$. The arrow map F_a will take each pair (a, b) of elements of *Obj* to an invertible arrow «$F_a\ a\ b : Hom\ a\ b \to R''.Hom\ (F_o\ a)\ (F_o\ b)$» of **RTS**. This arrow corresponds to the invertible simulation from $HOM_{EC}\ a\ b$ to $R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b)$ that takes $t \in Hom\ a\ b$ to $DN\ (MkArr\ a\ b\ t) \in R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b)$.

abbreviation $F_o :: 'O \Rightarrow 'A$
where $F_o \equiv \lambda a.\ DN\ (MkArr\ a\ a\ (RTS.Map\ (Id\ a)\ RTS.One.the\text{-}arr))$

abbreviation $F_a :: 'O \Rightarrow 'O \Rightarrow 'A\ rtscat.arr$
where $F_a \equiv \lambda a\ b.$ *if* $a \in Obj \wedge b \in Obj$
then $RTS.mkarr\ (HOM_{EC}\ a\ b)\ (R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b))$
$(\lambda t.$ *if* $residuation.arr\ (HOM_{EC}\ a\ b)\ t$
then $DN\ (MkArr\ a\ b\ t)$
else $ResiduatedTransitionSystem.partial\text{-}magma.null$
$(R''.HOM_{EC}\ (F_o\ a)\ (F_o\ b)))$

else RTS.null

lemma *ide-F_o*:
assumes $a \in Obj$
shows *DN (MkArr a a (RTS.Map (Id a) RTS.One.the-arr)) ∈ Collect H'.ide*
⟨*proof*⟩

lemma *bij-F_o*:
shows *bij-betw F_o Obj R''.Obj*
⟨*proof*⟩

lemma *F_a-in-hom* [*intro*, *simp*]:
assumes $a \in Obj$ **and** $b \in Obj$
shows «*F_a a b : Hom a b → R''.Hom (F_o a) (F_o b)*»
⟨*proof*⟩

lemma *F_a-simps* [*simp*]:
assumes $a \in Obj$ **and** $b \in Obj$
shows *RTS.arr (F_a a b)*
and *RTS.dom (F_a a b) = Hom a b*
and *RTS.cod (F_a a b) = R''.Hom (F_o a) (F_o b)*
⟨*proof*⟩

lemma *Map-F_a-simp* [*simp*]:
assumes $a \in Obj$ **and** $b \in Obj$ **and** *residuation.arr (HOM_{EC} a b) t*
shows *RTS.Map (F_a a b) t = DN (MkArr a b t)*
⟨*proof*⟩

interpretation Φ: *rts-enriched-functor*
Obj Hom Id Comp R''.Obj R''.Hom R''.Id R''.Comp
F_o F_a
⟨*proof*⟩

lemma *induces-rts-enriched-functor*:
shows *rts-enriched-functor*
Obj Hom Id Comp R''.Obj R''.Hom R''.Id R''.Comp F_o F_a
⟨*proof*⟩

proposition *induces-fully-faithful-rts-enriched-functor*:
shows *fully-faithful-rts-enriched-functor*
Obj Hom Id Comp R''.Obj R''.Hom R''.Id R''.Comp F_o F_a
⟨*proof*⟩

end

5.5 RTS† Determined by its Underlying Category

In this section we show that the category **RTS**† is fully determined by its subcategory **RTS** comprising the arrows that are identities for the residuation. Specifically, we show that there is an invertible RTS-functor from **RTS**† to the RTS-category obtained from the category **RTS** regarded as a category enriched in itself.

context *rtscat*
begin

The following produces a stand-alone instance of the category **RTS**†, independent of the current context. Arrows of **RTS**† have type $'A$ *rtscatx.arr* and they have the form *MkArr A B F*, where *A* and *B* have type $'A$ *resid* and *F* has the type $('A, 'A)$ *exponential-rts.arr* of an arrow of the exponential RTS $[A, B]$).

interpretation *RTSx*: *rtscatx arr-type* ⟨*proof*⟩

In the current locale context, *comp* is the composition for the ordinary category **RTS**. As a cartesian closed category, this category determines a category enriched in itself.

interpretation *enriched-category comp Prod* α ι
‹*Collect ide*› *exp ECMC.Id ECMC.Comp*
⟨*proof*⟩

This self-enriched category determines an RTS-category, using the general construction defined in *rts-category-of-enriched-category*. We will refer to this RTS-category as **RC**. Arrows of **RC** have type $('A$ *rtscatx.arr*, $'A)$ *RC.arr* and they have the form *RC.MkArr a b t*, where *a* and *b* are objects of **RTS** and *t* is an arrow of the hom-RTS HOM_{EC} *a b*, which has arrow type $'A$.

interpretation *RC*: *rts-category-of-enriched-category*
arr-type ‹*Collect ide*› *exp ECMC.Id ECMC.Comp*
⟨*proof*⟩

We now define the mapping Φ which we will show to be an RTS-category isomorphism from **RC** to **RTS**. In order to map an arrow *MkArr a b t* of **RC** to an arrow of **RTS**, it is necessary to use the invertible simulation *RTS.Func a b* to lift the arrow $t :: 'A$ of HOM_{EC} *a b* to an arrow *RTS.Func a b t* :: $('A, 'A)$ *exponential-rts.arr* of the exponential RTS [*RTSx.Rts a*, *RTSx.Rts b*].

definition Φ :: $('A$ *rtscatx.arr*, $'A)$ *RC.arr* $\Rightarrow$ $'A$ *rtscatx.arr*
where Φ *t* $\equiv$ *if RC.arr t*
then RTSx.MkArr
(*RTSx.Dom* (*RC.Dom t*)) (*RTSx.Dom* (*RC.Cod t*))
(*Func* (*RC.Dom t*) (*RC.Cod t*) (*RC.Trn t*))
else RTSx.null

lemma Φ-*simps* [*simp*]:
assumes *RC.arr t*
shows *RTSx.arr* (Φ *t*)
and *RTSx.dom* (Φ *t*) = *RTSx.mkobj* (*RTSx.Dom* (*RC.Dom t*))
and *RTSx.cod* (Φ *t*) = *RTSx.mkobj* (*RTSx.Dom* (*RC.Cod t*))
⟨*proof*⟩

lemma Φ-*in-hom* [*intro*]:
assumes *RC.arr t*
shows *RTSx.H.in-hom* (Φ *t*)
(*RTSx.mkobj*
(*RTSx.Dom* (*RC.Dom t*))) (*RTSx.mkobj* (*RTSx.Dom* (*RC.Cod t*)))
⟨*proof*⟩

interpretation Φ: *simulation RC.resid RTSx.resid* Φ
⟨*proof*⟩

The following fact is key to showing that Φ is functorial.

lemma *Func-Trn-obj*:
assumes *RC.obj a*
shows *Func* (*RC.Dom a*) (*RC.Cod a*) (*RC.Trn a*) =
exponential-rts.MkIde (*I* (*Rts* (*RC.Dom a*)))
⟨*proof*⟩

lemma *obj*-Φ-*obj*:
assumes *RC.obj a*
shows *RTSx.obj* (Φ *a*)
⟨*proof*⟩

interpretation Φ: *functor RC.hcomp RTSx.hcomp* Φ
⟨*proof*⟩

interpretation Φ: *rts-functor RC.resid RC.hcomp*
RTSx.resid RTSx.hcomp Φ
⟨*proof*⟩

interpretation Φ: *fully-faithful-functor RC.hcomp RTSx.hcomp* Φ
⟨*proof*⟩

interpretation Φ: *full-embedding-functor RC.hcomp RTSx.hcomp* Φ
⟨*proof*⟩

interpretation Φ: *invertible-functor RC.hcomp RTSx.hcomp* Φ
⟨*proof*⟩

interpretation Φ: *invertible-simulation RC.resid RTSx.resid* Φ
⟨*proof*⟩

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.