Residuated Transition Systems

Eugene W. Stark

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

March 17, 2025

Abstract

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

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

Contents

Chapter 1

Introduction

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

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

In [10], concurrent transition systems were defined in terms of graphs, consisting of states, transitions, and a pair of functions that assign to each transition a *source* (or domain) state and a *target* (or codomain) state. In addition, the presence of transitions that are *identities* for the residuation was assumed. Identity transitions had the same source and target state, and they could be thought of as representing empty computational steps. The key axiom for concurrent transition systems is the "cube axiom", which is a parallel moves property stating that the same result is achieved when transporting a transition by residuation along the two paths from the base to the apex of a "commuting diamond". Using the residuation operation and the associated cube axiom, it becomes possible to define notions of "join" and "composition" of transitions. The residuation also induces a notion of congruence of transitions; namely, transitions t and u are congruent whenever they are coinitial and both $t\backslash u$ and $u\backslash t$ are identities. In [10], the basic definition of concurrent transition system included an axiom, called "extensionality", which states that the congruence relation is trivial (*i.e.* coincides with equality). An advantage of the extensionality axiom is that, in its presence, joins and composites of transitions are uniquely defined when they exist. It was shown in [10] that a concurrent transition system could always be quotiented by congruence to achieve extensionality.

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

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

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

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

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

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

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

Chapter 2

Residuated Transition Systems

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

2.1 Basic Definitions and Properties

2.1.1 Partial Magmas

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

locale *partial-magma* =
fixes $OP :: 'a \Rightarrow 'a \Rightarrow 'a$
assumes *ex-un-null*: $\exists !n.\ \forall t.\ OP\ n\ t = n \wedge OP\ t\ n = n$
begin

definition $null :: 'a$
where $null = (THE\ n.\ \forall t.\ OP\ n\ t = n \wedge OP\ t\ n = n)$

lemma *null-eqI*:
assumes $\bigwedge t.\ OP\ n\ t = n \wedge OP\ t\ n = n$
shows $n = null$
⟨*proof*⟩

lemma *null-is-zero* [*simp*]:
shows $OP\ null\ t = null$ **and** $OP\ t\ null = null$
⟨*proof*⟩

end

2.1.2 Residuation

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

type-synonym $'a\ resid = 'a \Rightarrow 'a \Rightarrow 'a$

locale *residuation* = *partial-magma resid*
for *resid* :: $'a$ *resid* (**infix** ‹\› *70*) +
assumes *con-sym-ax*: $t \setminus u \neq null \Longrightarrow u \setminus t \neq null$
and *con-imp-arr-resid*: $t \setminus u \neq null \Longrightarrow (t \setminus u) \setminus (t \setminus u) \neq null$
and *cube-ax*: $(v \setminus t) \setminus (u \setminus t) \neq null \Longrightarrow (v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$
begin

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

lemma *cube*:
shows $(v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$
⟨*proof*⟩

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

definition *con* (**infix** ‹⌢› *50*)
where $t \frown u \equiv t \setminus u \neq null$

lemma *conI* [*intro*]:
assumes $t \setminus u \neq null$
shows $t \frown u$
⟨*proof*⟩

lemma *conE* [*elim*]:
assumes $t \frown u$
and $t \setminus u \neq null \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *con-sym*:
assumes $t \frown u$
shows $u \frown t$
⟨*proof*⟩

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

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

lemma *arrI* [*intro*]:
assumes $t \frown t$

shows *arr t*
⟨*proof*⟩

lemma *arrE* [*elim*]:
assumes *arr t*
and $t \frown t \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *not-arr-null* [*simp*]:
shows ¬ *arr null*
⟨*proof*⟩

lemma *con-implies-arr*:
assumes $t \frown u$
shows *arr t* **and** *arr u*
⟨*proof*⟩

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

lemma *arr-resid-iff-con*:
shows *arr* $(t \setminus u) \longleftrightarrow t \frown u$
⟨*proof*⟩

lemma *con-arr-self* [*simp*]:
assumes *arr f*
shows $f \frown f$
⟨*proof*⟩

lemma *not-con-null* [*simp*]:
shows *con null t* = *False* **and** *con t null* = *False*
⟨*proof*⟩

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

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

lemma *resid-arr-self*:
shows $t \setminus t$ = *trg t*
⟨*proof*⟩

lemma *arr-trg-iff-arr*:
shows *arr* (*trg t*) ⟷ *arr t*
⟨*proof*⟩

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

definition *ide*

where *ide* $a \equiv a \frown a \wedge a \setminus a = a$

lemma *ideI* [*intro*]:
assumes $a \frown a$ **and** $a \setminus a = a$
shows *ide* a
⟨*proof*⟩

lemma *ideE* [*elim*]:
assumes *ide* a
and $[\![a \frown a;\ a \setminus a = a]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *ide-implies-arr* [*simp*]:
assumes *ide* a
shows *arr* a
⟨*proof*⟩

lemma *not-ide-null* [*simp*]:
shows *ide null* = *False*
⟨*proof*⟩

end

2.1.3 Residuated Transition System

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

locale *rts* = *residuation* +
assumes *ide-trg* [*simp*]: *arr* $t \Longrightarrow$ *ide* (*trg* t)
and *resid-arr-ide*: $[\![\textit{ide}\ a;\ t \frown a]\!] \Longrightarrow t \setminus a = t$
and *resid-ide-arr* [*simp*]: $[\![\textit{ide}\ a;\ a \frown t]\!] \Longrightarrow \textit{ide}\ (a \setminus t)$
and *con-imp-coinitial-ax*: $t \frown u \Longrightarrow \exists\, a.\ \textit{ide}\ a \wedge a \frown t \wedge a \frown u$
and *con-target*: $[\![\textit{ide}\ (t \setminus u);\ u \frown v]\!] \Longrightarrow t \setminus u \frown v \setminus u$
begin

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

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

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

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

lemma *in-sourcesI* [*intro*, *simp*]:
assumes *ide a* **and** $t \frown a$
shows $a \in sources\ t$
⟨*proof*⟩

lemma *in-sourcesE* [*elim*]:
assumes $a \in sources\ t$
and $[\![ide\ a;\ t \frown a]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *in-targetsI* [*intro*, *simp*]:
assumes *ide b* **and** $trg\ t \frown b$
shows $b \in targets\ t$
⟨*proof*⟩

lemma *in-targetsE* [*elim*]:
assumes $b \in targets\ t$
and $[\![ide\ b;\ trg\ t \frown b]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *trg-in-targets*:
assumes *arr t*
shows $trg\ t \in targets\ t$
⟨*proof*⟩

lemma *source-is-ide*:
assumes $a \in sources\ t$
shows *ide a*
⟨*proof*⟩

lemma *target-is-ide*:
assumes $a \in targets\ t$
shows *ide a*
⟨*proof*⟩

Consistent arrows have a common source.

lemma *con-imp-common-source*:
assumes $t \frown u$
shows $sources\ t \cap sources\ u \neq \{\}$

$\langle proof \rangle$

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

lemma *arr-iff-has-source*:
shows $arr\ t \longleftrightarrow sources\ t \neq \{\}$
$\langle proof \rangle$

lemma *arr-iff-has-target*:
shows $arr\ t \longleftrightarrow targets\ t \neq \{\}$
$\langle proof \rangle$

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

lemma *resid-source-in-targets*:
assumes $a \in sources\ t$
shows $a \setminus t \in targets\ t$
$\langle proof \rangle$

Residuation along an identity reflects identities.

lemma *ide-backward-stable*:
assumes $ide\ a$ **and** $ide\ (t \setminus a)$
shows $ide\ t$
$\langle proof \rangle$

lemma *resid-reflects-con*:
assumes $t \frown v$ **and** $u \frown v$ **and** $t \setminus v \frown u \setminus v$
shows $t \frown u$
$\langle proof \rangle$

lemma *con-transitive-on-ide*:
assumes $ide\ a$ **and** $ide\ b$ **and** $ide\ c$
shows $[\![a \frown b;\ b \frown c]\!] \Longrightarrow a \frown c$
$\langle proof \rangle$

lemma *sources-are-con*:
assumes $a \in sources\ t$ **and** $a' \in sources\ t$
shows $a \frown a'$
$\langle proof \rangle$

lemma *sources-con-closed*:
assumes $a \in sources\ t$ **and** $ide\ a'$ **and** $a \frown a'$
shows $a' \in sources\ t$
$\langle proof \rangle$

lemma *sources-eqI*:
assumes $sources\ t \cap sources\ t' \neq \{\}$
shows $sources\ t = sources\ t'$

$\langle proof \rangle$

lemma *targets-are-con*:
assumes $b \in targets\ t$ **and** $b' \in targets\ t$
shows $b \frown b'$
$\langle proof \rangle$

lemma *targets-con-closed*:
assumes $b \in targets\ t$ **and** $ide\ b'$ **and** $b \frown b'$
shows $b' \in targets\ t$
$\langle proof \rangle$

lemma *targets-eqI*:
assumes $targets\ t \cap targets\ t' \neq \{\}$
shows $targets\ t = targets\ t'$
$\langle proof \rangle$

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

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

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

lemma *coinitialI* [*intro*]:
assumes $arr\ t$ **and** $sources\ t = sources\ u$
shows $coinitial\ t\ u$
$\langle proof \rangle$

lemma *coinitialE* [*elim*]:
assumes $coinitial\ t\ u$
and $[\![arr\ t; arr\ u; sources\ t = sources\ u]\!] \Longrightarrow T$
shows T
$\langle proof \rangle$

lemma *con-imp-coinitial*:
assumes $t \frown u$
shows $coinitial\ t\ u$
$\langle proof \rangle$

lemma *coinitial-iff*:
shows $coinitial\ t\ t' \longleftrightarrow arr\ t \wedge arr\ t' \wedge sources\ t = sources\ t'$
$\langle proof \rangle$

lemma *coterminal-iff*:
shows $coterminal\ t\ t' \longleftrightarrow arr\ t \wedge arr\ t' \wedge targets\ t = targets\ t'$
$\langle proof \rangle$

lemma *coterminal-iff-con-trg*:
shows *coterminal t u* $\longleftrightarrow$ *trg t* $\frown$ *trg u*
⟨*proof*⟩

lemma *coterminalI* [*intro*]:
assumes *arr t* **and** *targets t* = *targets u*
shows *coterminal t u*
⟨*proof*⟩

lemma *coterminalE* [*elim*]:
assumes *coterminal t u*
and ⟦*arr t*; *arr u*; *targets t* = *targets u*⟧ $\Longrightarrow$ *T*
shows *T*
⟨*proof*⟩

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

lemma *targets-resid-sym*:
assumes $t \frown u$
shows *targets* $(t \setminus u)$ = *targets* $(u \setminus t)$
⟨*proof*⟩

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

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

lemma *seqI* [*intro*]:
shows ⟦*arr t*; *targets t* = *sources u*⟧ $\Longrightarrow$ *seq t u*
and ⟦*arr u*; *targets t* = *sources u*⟧ $\Longrightarrow$ *seq t u*
⟨*proof*⟩

lemma *seqE* [*elim*]:
assumes *seq t u*
and ⟦*arr t*; *arr u*; *targets t* = *sources u*⟧ $\Longrightarrow$ *T*
shows *T*
⟨*proof*⟩

Congruence of Transitions

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

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

lemma *prfxE*:
assumes $t \lesssim u$

and *ide* $(t \setminus u) \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *prfx-implies-con*:
assumes $t \lesssim u$
shows $t \frown u$
⟨*proof*⟩

lemma *prfx-reflexive*:
assumes *arr* t
shows $t \lesssim t$
⟨*proof*⟩

lemma *prfx-transitive* [*trans*]:
assumes $t \lesssim u$ **and** $u \lesssim v$
shows $t \lesssim v$
⟨*proof*⟩

lemma *source-is-prfx*:
assumes $a \in$ *sources* t
shows $a \lesssim t$
⟨*proof*⟩

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

abbreviation *cong* (**infix** ‹∼› *50*)
where $t \sim u \equiv t \lesssim u \wedge u \lesssim t$

lemma *congE*:
assumes $t \sim u$
and $[\![t \frown u;$ *ide* $(t \setminus u);$ *ide* $(u \setminus t)]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *cong-reflexive*:
assumes *arr* t
shows $t \sim t$
⟨*proof*⟩

lemma *cong-symmetric*:
assumes $t \sim u$
shows $u \sim t$
⟨*proof*⟩

lemma *cong-transitive* [*trans*]:
assumes $t \sim u$ **and** $u \sim v$
shows $t \sim v$
⟨*proof*⟩

lemma *cong-subst-left*:
assumes $t \sim t'$ **and** $t \frown u$
shows $t' \frown u$ **and** $t \setminus u \sim t' \setminus u$
  $\langle proof \rangle$

lemma *cong-subst-right*:
assumes $u \sim u'$ **and** $t \frown u$
shows $t \frown u'$ **and** $t \setminus u \sim t \setminus u'$
$\langle proof \rangle$

lemma *cong-implies-coinitial*:
assumes $u \sim u'$
shows *coinitial* u u'
  $\langle proof \rangle$

lemma *cong-implies-coterminal*:
assumes $u \sim u'$
shows *coterminal* u u'
  $\langle proof \rangle$

lemma *ide-imp-con-iff-cong*:
assumes *ide* t **and** *ide* u
shows $t \frown u \longleftrightarrow t \sim u$
  $\langle proof \rangle$

lemma *sources-are-cong*:
assumes $a \in$ *sources* t **and** $a' \in$ *sources* t
shows $a \sim a'$
  $\langle proof \rangle$

lemma *sources-cong-closed*:
assumes $a \in$ *sources* t **and** $a \sim a'$
shows $a' \in$ *sources* t
  $\langle proof \rangle$

lemma *targets-are-cong*:
assumes $b \in$ *targets* t **and** $b' \in$ *targets* t
shows $b \sim b'$
  $\langle proof \rangle$

lemma *targets-cong-closed*:
assumes $b \in$ *targets* t **and** $b \sim b'$
shows $b' \in$ *targets* t
  $\langle proof \rangle$

lemma *targets-char*:
shows *targets* $t = \{b.\ arr\ t \wedge t \setminus t \sim b\}$
  $\langle proof \rangle$

lemma *coinitial-ide-are-cong*:
assumes *ide a* **and** *ide a'* **and** *coinitial a a'*
shows $a \sim a'$
  ⟨*proof*⟩

lemma *cong-respects-seq*:
assumes *seq t u* **and** *cong t t'* **and** *cong u u'*
shows *seq t' u'*
  ⟨*proof*⟩

Chosen Sources

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

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

lemma *src-in-sources*:
assumes *arr t*
shows *src t* $\in$ *sources t*
  ⟨*proof*⟩

lemma *src-congI*:
assumes *ide a* **and** $a \frown t$
shows *src t* $\sim a$
  ⟨*proof*⟩

lemma *arr-src-iff-arr*:
shows *arr* (*src t*) $\longleftrightarrow$ *arr t*
  ⟨*proof*⟩

lemma *arr-src-if-arr* [*simp*]:
assumes *arr t*
shows *arr* (*src t*)
  ⟨*proof*⟩

lemma *sources-char*$_{CS}$:
shows *sources t* = {*a. arr t* $\wedge$ *src t* $\sim$ *a*}
  ⟨*proof*⟩

lemma *targets-char'*:
shows *targets t* = {*b. arr t* $\wedge$ *trg t* $\sim$ *b*}
  ⟨*proof*⟩

lemma *con-imp-cong-src*:
assumes $t \frown u$
shows *src t* $\sim$ *src u*

$\langle proof \rangle$

lemma *ide-src* [*simp*]:
assumes *arr t*
shows *ide* (*src t*)
$\langle proof \rangle$

lemma *src-resid*:
assumes $t \frown u$
shows *src* $(t \setminus u) \sim$ *trg u*
$\langle proof \rangle$

lemma *apex-arr-prfx'*:
assumes *prfx t u*
shows *trg* $(u \setminus t) \sim$ *trg u*
and *trg* $(t \setminus u) \sim$ *trg u*
$\langle proof \rangle$

lemma $seqI_{CS}$ [*intro*, *simp*]:
shows $[\![$*arr t*; *trg t* $\sim$ *src u*$]\!] \Longrightarrow$ *seq t u*
and $[\![$*arr u*; *trg t* $\sim$ *src u*$]\!] \Longrightarrow$ *seq t u*
$\langle proof \rangle$

lemma $seqE_{CS}$ [*elim*]:
assumes *seq t u*
and $[\![$*arr u*; *arr t*; *trg t* $\sim$ *src u*$]\!] \Longrightarrow T$
shows T
$\langle proof \rangle$

lemma *coinitial-iff'*:
shows *coinitial t u* $\longleftrightarrow$ *arr t* $\wedge$ *arr u* $\wedge$ *src t* $\sim$ *src u*
$\langle proof \rangle$

lemma *coterminal-iff'*:
shows *coterminal t u* $\longleftrightarrow$ *arr t* $\wedge$ *arr u* $\wedge$ *trg t* $\sim$ *trg u*
$\langle proof \rangle$

lemma *coinitialI'* [*intro*]:
assumes *arr t* **and** *src t* $\sim$ *src u*
shows *coinitial t u*
$\langle proof \rangle$

lemma *coinitialE'* [*elim*]:
assumes *coinitial t u*
and $[\![$*arr t*; *arr u*; *src t* $\sim$ *src u*$]\!] \Longrightarrow T$
shows T
$\langle proof \rangle$

lemma *coterminalI'* [*intro*]:

assumes *arr t* **and** *trg t* $\sim$ *trg u*
shows *coterminal t u*
  ⟨*proof*⟩

lemma *coterminalE′* [*elim*]:
assumes *coterminal t u*
and ⟦*arr t*; *arr u*; *trg t* $\sim$ *trg u*⟧ $\Longrightarrow$ *T*
shows *T*
  ⟨*proof*⟩

lemma *src-cong-ide*:
assumes *ide a*
shows *src a* $\sim$ *a*
  ⟨*proof*⟩

lemma *trg-ide* [*simp*]:
assumes *ide a*
shows *trg a* = *a*
  ⟨*proof*⟩

lemma *ide-iff-src-cong-self*:
assumes *arr a*
shows *ide a* $\longleftrightarrow$ *src a* $\sim$ *a*
  ⟨*proof*⟩

lemma *ide-iff-trg-cong-self*:
assumes *arr a*
shows *ide a* $\longleftrightarrow$ *trg a* $\sim$ *a*
  ⟨*proof*⟩

lemma *src-src-cong-src*:
assumes *arr t*
shows *src* (*src t*) $\sim$ *src t*
  ⟨*proof*⟩

lemma *trg-trg-cong-trg*:
assumes *arr t*
shows *trg* (*trg t*) $\sim$ *trg t*
  ⟨*proof*⟩

lemma *src-trg-cong-trg*:
assumes *arr t*
shows *src* (*trg t*) $\sim$ *trg t*
  ⟨*proof*⟩

lemma *trg-src-cong-src*:
assumes *arr t*
shows *trg* (*src t*) $\sim$ *src t*
  ⟨*proof*⟩

lemma *resid-ide-cong*:
assumes *ide a* **and** *coinitial a t*
shows $t \setminus a \sim t$ **and** $a \setminus t \sim trg\ t$
⟨*proof*⟩

lemma *con-arr-src* [*simp*]:
assumes *arr f*
shows $f \frown src\ f$ **and** $src\ f \frown f$
⟨*proof*⟩

lemma *resid-src-arr-cong*:
assumes *arr f*
shows $src\ f \setminus f \sim trg\ f$
⟨*proof*⟩

lemma *resid-arr-src-cong*:
assumes *arr f*
shows $f \setminus src\ f \sim f$
⟨*proof*⟩

end

2.1.4 Weakly Extensional RTS

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

locale *weakly-extensional-rts* = *rts* +
assumes *weak-extensionality*: $[\![t \sim u;\ ide\ t;\ ide\ u]\!] \Longrightarrow t = u$
begin

lemma *con-ide-are-eq*:
assumes *ide a* **and** *ide a′* **and** $a \frown a'$
shows $a = a'$
⟨*proof*⟩

lemma *coinitial-ide-are-eq*:
assumes *ide a* **and** *ide a′* **and** *coinitial a a′*
shows $a = a'$
⟨*proof*⟩

lemma *arr-has-un-source*:
assumes *arr t*
shows $\exists!a.\ a \in sources\ t$
⟨*proof*⟩

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

shows $\exists!b.\ b \in \mathit{targets}\ t$
  $\langle \mathit{proof} \rangle$

lemma *src-eqI*:
assumes *ide a* **and** $a \frown t$
shows $\mathit{src}\ t = a$
  $\langle \mathit{proof} \rangle$

lemma *sources-char*$_{WE}$:
shows $\mathit{sources}\ t = \{a.\ \mathit{arr}\ t \wedge \mathit{src}\ t = a\}$
  $\langle \mathit{proof} \rangle$

lemma *targets-char*$_{WE}$:
shows $\mathit{targets}\ t = \{b.\ \mathit{arr}\ t \wedge \mathit{trg}\ t = b\}$
  $\langle \mathit{proof} \rangle$

lemma *con-imp-eq-src*:
assumes $t \frown u$
shows $\mathit{src}\ t = \mathit{src}\ u$
  $\langle \mathit{proof} \rangle$

lemma *src-resid*$_{WE}$ [*simp*]:
assumes $t \frown u$
shows $\mathit{src}\ (t \setminus u) = \mathit{trg}\ u$
  $\langle \mathit{proof} \rangle$

lemma *apex-sym*:
shows $\mathit{trg}\ (t \setminus u) = \mathit{trg}\ (u \setminus t)$
  $\langle \mathit{proof} \rangle$

lemma *apex-arr-prfx*$_{WE}$:
assumes *prfx t u*
shows $\mathit{trg}\ (u \setminus t) = \mathit{trg}\ u$
and $\mathit{trg}\ (t \setminus u) = \mathit{trg}\ u$
  $\langle \mathit{proof} \rangle$

lemma *seqI*$_{WE}$ [*intro*, *simp*]:
shows $[\![\mathit{arr}\ t;\ \mathit{trg}\ t = \mathit{src}\ u]\!] \Longrightarrow \mathit{seq}\ t\ u$
and $[\![\mathit{arr}\ u;\ \mathit{trg}\ t = \mathit{src}\ u]\!] \Longrightarrow \mathit{seq}\ t\ u$
  $\langle \mathit{proof} \rangle$

lemma *seqE*$_{WE}$ [*elim*]:
assumes *seq t u*
and $[\![\mathit{arr}\ u;\ \mathit{arr}\ t;\ \mathit{trg}\ t = \mathit{src}\ u]\!] \Longrightarrow T$
shows T
  $\langle \mathit{proof} \rangle$

lemma *coinitial-iff*$_{WE}$:
shows $\mathit{coinitial}\ t\ u \longleftrightarrow \mathit{arr}\ t \wedge \mathit{arr}\ u \wedge \mathit{src}\ t = \mathit{src}\ u$

$\langle proof \rangle$

lemma *coterminal-iff*$_{WE}$:
shows *coterminal t u* $\longleftrightarrow$ *arr t* $\wedge$ *arr u* $\wedge$ *trg t* = *trg u*
$\langle proof \rangle$

lemma *coinitialI*$_{WE}$ [*intro*]:
assumes *arr t* **and** *src t* = *src u*
shows *coinitial t u*
$\langle proof \rangle$

lemma *coinitialE*$_{WE}$ [*elim*]:
assumes *coinitial t u*
and $[\![$*arr t*; *arr u*; *src t* = *src u*$]\!] \Longrightarrow T$
shows T
$\langle proof \rangle$

lemma *coterminalI*$_{WE}$ [*intro*]:
assumes *arr t* **and** *trg t* = *trg u*
shows *coterminal t u*
$\langle proof \rangle$

lemma *coterminalE*$_{WE}$ [*elim*]:
assumes *coterminal t u*
and $[\![$*arr t*; *arr u*; *trg t* = *trg u*$]\!] \Longrightarrow T$
shows T
$\langle proof \rangle$

lemma *src-ide* [*simp*]:
assumes *ide a*
shows *src a* = *a*
$\langle proof \rangle$

lemma *ide-iff-src-self*:
assumes *arr a*
shows *ide a* $\longleftrightarrow$ *src a* = *a*
$\langle proof \rangle$

lemma *ide-iff-trg-self*:
assumes *arr a*
shows *ide a* $\longleftrightarrow$ *trg a* = *a*
$\langle proof \rangle$

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

lemma *trg-trg* [*simp*]:
shows *trg* (*trg t*) = *trg t*

⟨*proof*⟩

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

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

lemma *resid-ide*:
assumes *ide* a **and** *coinitial* a t
shows $t \setminus a = t$ **and** $a \setminus t =$ *trg* t
⟨*proof*⟩

lemma *resid-src-arr* [*simp*]:
assumes *arr* f
shows *src* $f \setminus f =$ *trg* f
⟨*proof*⟩

lemma *resid-arr-src* [*simp*]:
assumes *arr* f
shows $f \setminus$ *src* $f = f$
⟨*proof*⟩

end

2.1.5 Extensional RTS

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

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

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

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

end

2.1.6 Composites of Transitions

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

context *rts*
begin

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

lemma *composite-ofI* [*intro*]:
assumes $u \lesssim v$ **and** $v \setminus u \sim t$
shows *composite-of* $u\ t\ v$
⟨*proof*⟩

lemma *composite-ofE* [*elim*]:
assumes *composite-of* $u\ t\ v$
and $[\![u \lesssim v;\ v \setminus u \sim t]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

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

lemma *composite-of-unq-upto-cong*:
assumes *composite-of* $u\ t\ v$ **and** *composite-of* $u\ t\ v'$
shows $v \sim v'$
⟨*proof*⟩

lemma *composite-of-ide-arr*:
assumes *ide* a
shows *composite-of* $a\ t\ t \longleftrightarrow t \frown a$
⟨*proof*⟩

lemma *composite-of-arr-ide*:
assumes *ide* b
shows *composite-of* $t\ b\ t \longleftrightarrow t \setminus t \frown b$
⟨*proof*⟩

lemma *composite-of-source-arr*:
assumes *arr* t **and** $a \in$ *sources* t
shows *composite-of* $a\ t\ t$
⟨*proof*⟩

lemma *composite-of-arr-target*:
assumes *arr* t **and** $b \in$ *targets* t
shows *composite-of* $t\ b\ t$
⟨*proof*⟩

lemma *composite-of-ide-self*:
assumes *ide* a

shows *composite-of* a a a
⟨*proof*⟩

lemma *con-prfx-composite-of*:
assumes *composite-of* t u w
shows $t \frown w$ **and** $w \frown v \Longrightarrow t \frown v$
⟨*proof*⟩

lemma *sources-composite-of*:
assumes *composite-of* u t v
shows *sources* v = *sources* u
⟨*proof*⟩

lemma *targets-composite-of*:
assumes *composite-of* u t v
shows *targets* v = *targets* t
⟨*proof*⟩

lemma *resid-composite-of*:
assumes *composite-of* t u w **and** $w \frown v$
shows $v \setminus t \frown w \setminus t$
and $v \setminus t \frown u$
and $v \setminus w \sim (v \setminus t) \setminus u$
and *composite-of* $(t \setminus v)$ $(u \setminus (v \setminus t))$ $(w \setminus v)$
⟨*proof*⟩

lemma *con-composite-of-iff*:
assumes *composite-of* t u v
shows $w \frown v \longleftrightarrow w \setminus t \frown u$
⟨*proof*⟩

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

lemma *composableD* [*dest*]:
assumes *composable* t u
shows *arr* t **and** *arr* u **and** *targets* t = *sources* u
⟨*proof*⟩

lemma *composable-imp-seq*:
assumes *composable* t u
shows *seq* t u
⟨*proof*⟩

lemma *composable-permute*:
shows *composable* t $(u \setminus t) \longleftrightarrow$ *composable* u $(t \setminus u)$
⟨*proof*⟩

lemma *diamond-commutes-upto-cong*:

assumes *composite-of* t $(u \setminus t)$ v **and** *composite-of* u $(t \setminus u)$ v'
shows $v \sim v'$
⟨*proof*⟩

lemma *bounded-imp-con*:
assumes *composite-of* t u v **and** *composite-of* t' u' v
shows *con* t t'
⟨*proof*⟩

lemma *composite-of-cancel-left*:
assumes *composite-of* t u v **and** *composite-of* t u' v
shows $u \sim u'$
⟨*proof*⟩

end

RTS with Composites

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

lemma *composable-iff-seq*:
shows *composable* g $f \longleftrightarrow$ *seq* g f
⟨*proof*⟩

lemma *composableI* [*intro*]:
assumes *seq* g f
shows *composable* g f
⟨*proof*⟩

lemma *composableE* [*elim*]:
assumes *composable* g f **and** *seq* g $f \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *obtains-composite-of*:
assumes *seq* g f
obtains h **where** *composite-of* g f h
⟨*proof*⟩

end

2.1.7 Joins of Transitions

context *rts*
begin

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

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

lemma *join-ofI* [*intro*]:
assumes $composite\text{-}of\ t\ (u \setminus t)\ v$ **and** $composite\text{-}of\ u\ (t \setminus u)\ v$
shows $join\text{-}of\ t\ u\ v$
⟨*proof*⟩

lemma *join-ofE* [*elim*]:
assumes $join\text{-}of\ t\ u\ v$
and $[\![composite\text{-}of\ t\ (u \setminus t)\ v;\ composite\text{-}of\ u\ (t \setminus u)\ v]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

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

lemma *joinable-implies-con*:
assumes $joinable\ t\ u$
shows $t \frown u$
⟨*proof*⟩

lemma *joinable-implies-coinitial*:
assumes $joinable\ t\ u$
shows $coinitial\ t\ u$
⟨*proof*⟩

lemma *joinable-iff-composable*:
shows $joinable\ t\ u \longleftrightarrow composable\ t\ (u \setminus t)$
⟨*proof*⟩

lemma *join-of-un-upto-cong*:
assumes $join\text{-}of\ t\ u\ v$ **and** $join\text{-}of\ t\ u\ v'$
shows $v \sim v'$
⟨*proof*⟩

lemma *join-of-symmetric*:
assumes $join\text{-}of\ t\ u\ v$
shows $join\text{-}of\ u\ t\ v$
⟨*proof*⟩

lemma *join-of-arr-self*:
assumes $arr\ t$
shows $join\text{-}of\ t\ t\ t$
⟨*proof*⟩

lemma *join-of-arr-src*:
assumes $arr\ t$ **and** $a \in sources\ t$

shows *join-of* $a\ t\ t$ **and** *join-of* $t\ a\ t$
⟨*proof*⟩

lemma *sources-join-of*:
assumes *join-of* $t\ u\ v$
shows *sources* t = *sources* v **and** *sources* u = *sources* v
⟨*proof*⟩

lemma *targets-join-of*:
assumes *join-of* $t\ u\ v$
shows *targets* $(t \setminus u)$ = *targets* v **and** *targets* $(u \setminus t)$ = *targets* v
⟨*proof*⟩

lemma *join-of-resid*:
assumes *join-of* $t\ u\ w$ **and** *con* $v\ w$
shows *join-of* $(t \setminus v)\ (u \setminus v)\ (w \setminus v)$
⟨*proof*⟩

lemma *con-with-join-of-iff*:
assumes *join-of* $t\ u\ w$
shows $u \frown v \land v \setminus u \frown t \setminus u \Longrightarrow w \frown v$
and $w \frown v \Longrightarrow t \frown v \land v \setminus t \frown u \setminus t$
⟨*proof*⟩

lemma *join-of-respects-cong-left*:
assumes *join-of* $t\ u\ v$ **and** *cong* $t\ t'$
shows *join-of* $t'\ u\ v$
⟨*proof*⟩

lemma *join-of-respects-cong-right*:
assumes *join-of* $t\ u\ v$ **and** *cong* $u\ u'$
shows *join-of* $t\ u'\ v$
⟨*proof*⟩

end

RTS with Joins

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

2.1.8 Joins and Composites in a Weakly Extensional RTS

context *weakly-extensional-rts*
begin

lemma *src-composite-of*:
assumes *composite-of* $u\ t\ v$
shows *src* v = *src* u
⟨*proof*⟩

lemma *trg-composite-of*:
assumes *composite-of u t v*
shows *trg v* = *trg t*
⟨*proof*⟩

lemma *src-join-of*:
assumes *join-of t u v*
shows *src t* = *src v* **and** *src u* = *src v*
⟨*proof*⟩

lemma *trg-join-of*:
assumes *join-of t u v*
shows *trg* (*t* \ *u*) = *trg v* **and** *trg* (*u* \ *t*) = *trg v*
⟨*proof*⟩

end

2.1.9 Joins and Composites in an Extensional RTS

context *extensional-rts*
begin

lemma *composite-of-unique*:
assumes *composite-of t u v* **and** *composite-of t u v′*
shows *v* = *v′*
⟨*proof*⟩

lemma *divisors-of-ide*:
assumes *composite-of t u v* **and** *ide v*
shows *ide t* **and** *ide u*
⟨*proof*⟩

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

definition *comp* (**infixr** ‹·› *55*)
where *t* · *u* ≡ *if composable t u then THE v. composite-of t u v else null*

lemma *comp-is-composite-of*:
shows *composable t u* ⟹ *composite-of t u* (*t* · *u*)
and *composite-of t u v* ⟹ *t* · *u* = *v*
⟨*proof*⟩

lemma *comp-null* [*simp*]:
shows *null* · *t* = *null* **and** *t* · *null* = *null*
⟨*proof*⟩

lemma *composable-iff-arr-comp*:
shows $composable\ t\ u \longleftrightarrow arr\ (t \cdot u)$
  ⟨*proof*⟩

lemma *composable-iff-comp-not-null*:
shows $composable\ t\ u \longleftrightarrow t \cdot u \neq null$
  ⟨*proof*⟩

lemma *comp-src-arr* [*simp*]:
assumes $arr\ t$ **and** $src\ t = a$
shows $a \cdot t = t$
  ⟨*proof*⟩

lemma *comp-arr-trg* [*simp*]:
assumes $arr\ t$ **and** $trg\ t = b$
shows $t \cdot b = t$
  ⟨*proof*⟩

lemma *comp-ide-self*:
assumes $ide\ a$
shows $a \cdot a = a$
  ⟨*proof*⟩

lemma *arr-comp* [*intro*, *simp*]:
assumes $composable\ t\ u$
shows $arr\ (t \cdot u)$
  ⟨*proof*⟩

lemma *trg-comp* [*simp*]:
assumes $composable\ t\ u$
shows $trg\ (t \cdot u) = trg\ u$
  ⟨*proof*⟩

lemma *src-comp* [*simp*]:
assumes $composable\ t\ u$
shows $src\ (t \cdot u) = src\ t$
  ⟨*proof*⟩

lemma *con-comp-iff*:
shows $w \frown t \cdot u \longleftrightarrow composable\ t\ u \wedge w \setminus t \frown u$
  ⟨*proof*⟩

lemma *con-compI* [*intro*]:
assumes $composable\ t\ u$ **and** $w \setminus t \frown u$
shows $w \frown t \cdot u$ **and** $t \cdot u \frown w$
  ⟨*proof*⟩

lemma *resid-comp*:

assumes $t \cdot u \frown w$
shows $w \setminus (t \cdot u) = (w \setminus t) \setminus u$
and $(t \cdot u) \setminus w = (t \setminus w) \cdot (u \setminus (w \setminus t))$
$\langle proof \rangle$

lemma *prfx-decomp*:
assumes $t \lesssim u$
shows $t \cdot (u \setminus t) = u$
$\langle proof \rangle$

lemma *prfx-comp*:
assumes *arr* u **and** $t \cdot v = u$
shows $t \lesssim u$
$\langle proof \rangle$

lemma *comp-eqI*:
assumes $t \lesssim v$ **and** $u = v \setminus t$
shows $t \cdot u = v$
$\langle proof \rangle$

lemma *comp-assoc*:
assumes *composable* $(t \cdot u)\ v$
shows $t \cdot (u \cdot v) = (t \cdot u) \cdot v$
$\langle proof \rangle$

We note the following assymmetry: *composable* $(t \cdot u)\ v \Longrightarrow$ *composable* $u\ v$ is true, but *composable* $t\ (u \cdot v) \Longrightarrow$ *composable* $t\ u$ is not.

lemma *comp-cancel-left*:
assumes *arr* $(t \cdot u)$ **and** $t \cdot u = t \cdot v$
shows $u = v$
$\langle proof \rangle$

lemma *comp-resid-prfx* [*simp*]:
assumes *arr* $(t \cdot u)$
shows $(t \cdot u) \setminus t = u$
$\langle proof \rangle$

lemma *bounded-imp-con*$_E$:
assumes $t \cdot u \sim t' \cdot u'$
shows $t \frown t'$
$\langle proof \rangle$

lemma *join-of-unique*:
assumes *join-of* $t\ u\ v$ **and** *join-of* $t\ u\ v'$
shows $v = v'$
$\langle proof \rangle$

definition *join* (**infix** ‹⊔› *52*)
where $t \sqcup u \equiv$ *if joinable t u then THE v. join-of t u v else null*

lemma *join-is-join-of*:
assumes *joinable* t u
shows *join-of* t u $(t \sqcup u)$
$\langle proof \rangle$

lemma *joinable-iff-arr-join*:
shows *joinable* t $u \longleftrightarrow$ *arr* $(t \sqcup u)$
$\langle proof \rangle$

lemma *joinable-iff-join-not-null*:
shows *joinable* t $u \longleftrightarrow t \sqcup u \neq$ *null*
$\langle proof \rangle$

lemma *join-sym*:
shows $t \sqcup u = u \sqcup t$
$\langle proof \rangle$

lemma *src-join*:
assumes *joinable* t u
shows *src* $(t \sqcup u) =$ *src* t
$\langle proof \rangle$

lemma *trg-join*:
assumes *joinable* t u
shows *trg* $(t \sqcup u) =$ *trg* $(t \setminus u)$
$\langle proof \rangle$

lemma *resid-join*$_E$ [*simp*]:
assumes *joinable* t u **and** $v \frown t \sqcup u$
shows $v \setminus (t \sqcup u) = (v \setminus u) \setminus (t \setminus u)$
and $v \setminus (t \sqcup u) = (v \setminus t) \setminus (u \setminus t)$
and $(t \sqcup u) \setminus v = (t \setminus v) \sqcup (u \setminus v)$
$\langle proof \rangle$

lemma *join-eqI*:
assumes $t \lesssim v$ **and** $u \lesssim v$ **and** $v \setminus u = t \setminus u$ **and** $v \setminus t = u \setminus t$
shows $t \sqcup u = v$
$\langle proof \rangle$

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

lemma *join-src*:
assumes *arr* t
shows *src* $t \sqcup t = t$

⟨*proof*⟩

lemma *join-arr-self*:
assumes *arr* t
shows $t \sqcup t = t$
⟨*proof*⟩

lemma *arr-prfx-join-self*:
assumes *joinable* t u
shows $t \lesssim t \sqcup u$
⟨*proof*⟩

lemma *con-prfx*:
shows $[\![t \frown u;\ v \lesssim u]\!] \Longrightarrow t \frown v$
and $[\![t \frown u;\ v \lesssim t]\!] \Longrightarrow v \frown u$
⟨*proof*⟩

lemma *join-prfx*:
assumes $t \lesssim u$
shows $t \sqcup u = u$ **and** $u \sqcup t = u$
⟨*proof*⟩

lemma *con-with-join-if* [*intro*, *simp*]:
assumes *joinable* t u **and** $u \frown v$ **and** $v \setminus u \frown t \setminus u$
shows $t \sqcup u \frown v$
and $v \frown t \sqcup u$
⟨*proof*⟩

lemma *join-assoc*$_E$:
assumes *arr* $((t \sqcup u) \sqcup v)$ **and** *arr* $(t \sqcup (u \sqcup v))$
shows $(t \sqcup u) \sqcup v = t \sqcup (u \sqcup v)$
⟨*proof*⟩

lemma *join-prfx-monotone*:
assumes $t \lesssim u$ **and** $u \sqcup v \frown t \sqcup v$
shows $t \sqcup v \lesssim u \sqcup v$
⟨*proof*⟩

lemma *join-eqI'*:
assumes $t \lesssim v$ **and** $u \lesssim v$ **and** $v \setminus u = t \setminus u$ **and** $v \setminus t = u \setminus t$
shows $v = t \sqcup u$
⟨*proof*⟩

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

lemma *join-expansion*:
assumes *joinable* t u
shows $t \sqcup u = t \cdot (u \setminus t)$ **and** *seq* t $(u \setminus t)$

⟨*proof*⟩

lemma *join3-expansion*:
assumes *joinable* $(t \sqcup u)\ v$
shows $(t \sqcup u) \sqcup v = (t \cdot (u \setminus t)) \cdot ((v \setminus t) \setminus (u \setminus t))$
⟨*proof*⟩

lemma *join-comp*:
assumes *joinable* $(t \cdot u)\ v$
shows $(t \cdot u) \sqcup v = t \cdot (v \setminus t) \cdot (u \setminus (v \setminus t))$
⟨*proof*⟩

end

Extensional RTS with Joins

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

lemma *joinable-iff-con* [*iff*]:
shows *joinable* $t\ u \longleftrightarrow t \frown u$
⟨*proof*⟩

lemma *joinableE* [*elim*]:
assumes *joinable* $t\ u$ **and** $t \frown u \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *src-join*$_{EJ}$ [*simp*]:
assumes $t \frown u$
shows *src* $(t \sqcup u)$ = *src* t
⟨*proof*⟩

lemma *trg-join*$_{EJ}$:
assumes $t \frown u$
shows *trg* $(t \sqcup u)$ = *trg* $(t \setminus u)$
⟨*proof*⟩

lemma *resid-join*$_{EJ}$ [*simp*]:
assumes $t \frown u$ **and** $v \frown t \sqcup u$
shows $v \setminus (t \sqcup u) = (v \setminus t) \setminus (u \setminus t)$
and $(t \sqcup u) \setminus v = (t \setminus v) \sqcup (u \setminus v)$
⟨*proof*⟩

lemma *join-assoc*:
shows $t \sqcup (u \sqcup v) = (t \sqcup u) \sqcup v$
⟨*proof*⟩

lemma *join-is-lub*:
assumes $t \lesssim v$ **and** $u \lesssim v$
shows $t \sqcup u \lesssim v$
⟨*proof*⟩

end

Extensional RTS with Composites

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

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

lemma *seq-implies-arr-comp*:
assumes *seq t u*
shows *arr* $(t \cdot u)$
⟨*proof*⟩

lemma *arr-comp*$_{EC}$ [*intro*, *simp*]:
assumes *arr t* **and** *trg t* = *src u*
shows *arr* $(t \cdot u)$
⟨*proof*⟩

lemma *arr-compE*$_{EC}$ [*elim*]:
assumes *arr* $(t \cdot u)$
and ⟦*arr t*; *arr u*; *trg t* = *src u*⟧ $\Longrightarrow T$
shows T
⟨*proof*⟩

lemma *trg-comp*$_{EC}$ [*simp*]:
assumes *seq t u*
shows *trg* $(t \cdot u)$ = *trg u*
⟨*proof*⟩

lemma *src-comp*$_{EC}$ [*simp*]:
assumes *seq t u*
shows *src* $(t \cdot u)$ = *src t*
⟨*proof*⟩

lemma *con-comp-iff*$_{EC}$ [*simp*]:
shows $w \frown t \cdot u \longleftrightarrow$ *seq t u* $\wedge\ u \frown w \setminus t$
and $t \cdot u \frown w \longleftrightarrow$ *seq t u* $\wedge\ u \frown w \setminus t$
⟨*proof*⟩

lemma *comp-assoc*$_{EC}$:
shows $t \cdot (u \cdot v) = (t \cdot u) \cdot v$
⟨*proof*⟩

lemma *diamond-commutes*:
shows $t \cdot (u \setminus t) = u \cdot (t \setminus u)$
⟨*proof*⟩

lemma *mediating-transition*:
assumes $t \cdot v = u \cdot w$
shows $v \setminus (u \setminus t) = w \setminus (t \setminus u)$
⟨*proof*⟩

lemma *induced-arrow*:
assumes *seq* t u **and** $t \cdot u = t' \cdot u'$
shows $(t' \setminus t) \cdot (u \setminus (t' \setminus t)) = u$
and $(t \setminus t') \cdot (u \setminus (t' \setminus t)) = u'$
and $(t' \setminus t) \cdot v = u \Longrightarrow v = u \setminus (t' \setminus t)$
⟨*proof*⟩

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

sublocale *extensional-rts-with-joins*
⟨*proof*⟩

lemma *comp-join*$_{EC}$:
assumes *composable* t u **and** *joinable* u u'
shows *composable* t $(u \sqcup u')$
and $t \cdot (u \sqcup u') = t \cdot u \sqcup t \cdot u'$
⟨*proof*⟩

lemma *resid-common-prefix*:
assumes $t \cdot u \frown t \cdot v$
shows $(t \cdot u) \setminus (t \cdot v) = u \setminus v$
⟨*proof*⟩

end

2.1.10 Confluence

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

locale *confluent-rts* = *rts* +
assumes *confluence*: *coinitial* t $u \Longrightarrow$ *con* t u

2.2 Simulations

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

locale *simulation* =
  A: *rts A* +
  B: *rts B*
for $A :: 'a\ resid$ (**infix** ‹$\backslash_A$› *70*)
and $B :: 'b\ resid$ (**infix** ‹$\backslash_B$› *70*)
and $F :: 'a \Rightarrow 'b$ +
assumes *extensionality*: $\neg\ A.arr\ t \Longrightarrow F\ t = B.null$
and *preserves-con* [*simp*]: $A.con\ t\ u \Longrightarrow B.con\ (F\ t)\ (F\ u)$
and *preserves-resid* [*simp*]: $A.con\ t\ u \Longrightarrow F\ (t \backslash_A u) = F\ t \backslash_B F\ u$
begin

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

  notation *B.con* (**infix** ‹$\frown_B$› *50*)
  notation *B.prfx* (**infix** ‹$\lesssim_B$› *50*)
  notation *B.cong* (**infix** ‹$\sim_B$› *50*)

  lemma *preserves-reflects-arr* [*iff*]:
  shows $B.arr\ (F\ t) \longleftrightarrow A.arr\ t$
    ⟨*proof*⟩

  lemma *preserves-ide* [*simp*]:
  assumes $A.ide\ a$
  shows $B.ide\ (F\ a)$
    ⟨*proof*⟩

  lemma *preserves-sources*:
  shows $F\ `\ A.sources\ t \subseteq B.sources\ (F\ t)$
    ⟨*proof*⟩

  lemma *preserves-targets*:
  shows $F\ `\ A.targets\ t \subseteq B.targets\ (F\ t)$
    ⟨*proof*⟩

  lemma *preserves-trg* [*simp*]:
  assumes $A.arr\ t$
  shows $B.trg\ (F\ t) = F\ (A.trg\ t)$
    ⟨*proof*⟩

  lemma *preserves-seq*:
  shows $A.seq\ t\ u \Longrightarrow B.seq\ (F\ t)\ (F\ u)$
    ⟨*proof*⟩

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

lemma *preserves-joins*:
assumes *A.join-of t u v*
shows *B.join-of* $(F\ t)$ $(F\ u)$ $(F\ v)$
⟨*proof*⟩

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

lemma *preserves-cong*:
assumes $t \sim_A u$
shows $F\ t \sim_B F\ u$
⟨*proof*⟩

end

2.2.1 Identity Simulation

locale *identity-simulation* =
rts
begin

abbreviation *map*
where *map* $\equiv \lambda t.$ *if arr t then t else null*

sublocale *simulation resid resid map*
⟨*proof*⟩

end

2.2.2 Composite of Simulations

lemma *simulation-comp* [*intro*]:
assumes *simulation A B F* **and** *simulation B C G*
shows *simulation A C* $(G\ o\ F)$
⟨*proof*⟩

locale *composite-simulation* =
F: *simulation A B F* +
G: *simulation B C G*
for A :: *'a resid*
and B :: *'b resid*
and C :: *'c resid*
and F :: $'a \Rightarrow 'b$
and G :: $'b \Rightarrow 'c$
begin

abbreviation *map*

where *map* $\equiv$ *G o F*

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

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

end

2.2.3 Simulations into a Weakly Extensional RTS

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

lemma *preserves-src* [*simp*]:
shows $a \in A.sources\ t \Longrightarrow B.src\ (F\ t) = F\ a$
⟨*proof*⟩

lemma *preserves-trg* [*simp*]:
shows $b \in A.targets\ t \Longrightarrow B.trg\ (F\ t) = F\ b$
⟨*proof*⟩

end

2.2.4 Simulations into an Extensional RTS

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

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

lemma *preserves-comp*:
assumes *A.composite-of t u v*
shows $F\ v = F\ t \cdot_B F\ u$
⟨*proof*⟩

lemma *preserves-join*:
assumes *A.join-of t u v*
shows $F\ v = F\ t \sqcup_B F\ u$
⟨*proof*⟩

end

2.2.5 Simulations between Weakly Extensional RTS's

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

  lemma *preserves-src* [*simp*]:
  shows *B.src* (*F t*) = *F* (*A.src t*)
    ⟨*proof*⟩

  lemma *preserves-trg* [*simp*]:
  shows *B.trg* (*F t*) = *F* (*A.trg t*)
    ⟨*proof*⟩

end

2.2.6 Simulations between Extensional RTS's

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

  sublocale *simulation-between-weakly-extensional-rts* ⟨*proof*⟩

  notation *A.comp* (**infixr** $\cdot_A$ *55*)
  notation *A.join* (**infixr** $\sqcup_A$ *52*)

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

  lemma *preserves-join*:
  assumes *A.joinable t u*
  shows $F\ (t \sqcup_A u) = F\ t \sqcup_B F\ u$
    ⟨*proof*⟩

end

2.2.7 Transformations

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

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

locale *transformation* =
  A: *rts* A +
  B: *weakly-extensional-rts* B +
  F: *simulation* $A\ B\ F$ +
  G: *simulation* $A\ B\ G$
for A :: $'a$ *resid* (**infix** ‹$\backslash_A$› *70*)
and B :: $'b$ *resid* (**infix** ‹$\backslash_B$› *70*)
and F :: $'a \Rightarrow 'b$
and G :: $'a \Rightarrow 'b$
and τ :: $'a \Rightarrow 'b$ +
assumes *extensionality*: $\neg\ A.arr\ f \Longrightarrow \tau\ f = B.null$
and *respects-cong-ide*: $[\![A.ide\ a;\ A.cong\ a\ a']\!] \Longrightarrow \tau\ a = \tau\ a'$
and *preserves-src*: $A.ide\ f \Longrightarrow B.src\ (\tau\ f) = F\ f$
and *preserves-trg*: $A.ide\ f \Longrightarrow B.trg\ (\tau\ f) = G\ f$
and *naturality1-ax*: $a \in A.sources\ f \Longrightarrow \tau\ a \backslash_B\ F\ f = \tau\ (a \backslash_A\ f)$
and *naturality2-ax*: $a \in A.sources\ f \Longrightarrow F\ f \backslash_B\ \tau\ a = G\ f$
and *naturality3*: $a \in A.sources\ f \Longrightarrow B.join\text{-}of\ (\tau\ a)\ (F\ f)\ (\tau\ f)$
begin

  notation $A.con$ (**infix** ‹$\frown_A$› *50*)
  notation $A.prfx$ (**infix** ‹$\lesssim_A$› *50*)

  notation $B.con$ (**infix** ‹$\frown_B$› *50*)
  notation $B.prfx$ (**infix** ‹$\lesssim_B$› *50*)

  lemma *naturality1*:
  shows $\tau\ (A.src\ f) \backslash_B\ F\ f = \tau\ (A.trg\ f)$
    ⟨*proof*⟩

  lemma *naturality2*:
  shows $F\ f \backslash_B\ \tau\ (A.src\ f) = G\ f$
    ⟨*proof*⟩

  lemma *respects-cong*:
  assumes $A.cong\ u\ u'$
  shows $B.cong\ (\tau\ u)\ (\tau\ u')$
  ⟨*proof*⟩

end

2.3 Normal Sub-RTS's and Congruence

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

2.3.1 Normal Sub-RTS's

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

  lemma *prfx-closed*:
  assumes $u \in \mathfrak{N}$ **and** *R.prfx t u*
  shows $t \in \mathfrak{N}$
    ⟨*proof*⟩

  lemma *composite-closed*:
  assumes $t \in \mathfrak{N}$ **and** $u \in \mathfrak{N}$ **and** *R.composite-of t u v*
  shows $v \in \mathfrak{N}$
    ⟨*proof*⟩

  lemma *factor-closed*:
  assumes *R.composite-of t u v* **and** $v \in \mathfrak{N}$

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

lemma *resid-along-elem-preserves-con*:
assumes $t \frown t'$ **and** *R.coinitial* t u **and** $u \in \mathfrak{N}$
shows $t \setminus u \frown t' \setminus u$
⟨*proof*⟩

end

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

locale *normal-in-extensional-rts-with-composites* =
R: *extensional-rts* +
R: *rts-with-composites* +
normal-sub-rts
begin

lemma *factor-closed*$_{EC}$:
assumes $t \cdot u \in \mathfrak{N}$
shows $t \in \mathfrak{N}$ **and** $u \in \mathfrak{N}$
⟨*proof*⟩

lemma *comp-in-normal-iff*:
shows $t \cdot u \in \mathfrak{N} \longleftrightarrow t \in \mathfrak{N} \wedge u \in \mathfrak{N} \wedge$ *R.seq* t u
⟨*proof*⟩

end

2.3.2 Semi-Congruence

context *normal-sub-rts*
begin

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

abbreviation *Cong*$_0$ (**infix** ‹$\approx_0$› *50*)
where $t \approx_0 t' \equiv t \setminus t' \in \mathfrak{N} \wedge t' \setminus t \in \mathfrak{N}$

lemma *Cong*$_0$*-reflexive*:
assumes *R.arr* t
shows $t \approx_0 t$
⟨*proof*⟩

lemma *Cong*$_0$*-symmetric*:
assumes $t \approx_0 t'$
shows $t' \approx_0 t$
⟨*proof*⟩

lemma *$Cong_0$-transitive* [*trans*]:
assumes $t \approx_0 t'$ **and** $t' \approx_0 t''$
shows $t \approx_0 t''$
$\langle proof \rangle$

lemma *$Cong_0$-imp-con*:
assumes $t \approx_0 t'$
shows $R.con\ t\ t'$
$\langle proof \rangle$

lemma *$Cong_0$-imp-coinitial*:
assumes $t \approx_0 t'$
shows $R.sources\ t = R.sources\ t'$
$\langle proof \rangle$

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

lemma *Resid-along-normal-preserves-$Cong_0$*:
assumes $t \approx_0 t'$ **and** $u \in \mathfrak{N}$ **and** $R.sources\ t = R.sources\ u$
shows $t \setminus u \approx_0 t' \setminus u$
$\langle proof \rangle$

lemma *Resid-along-normal-reflects-$Cong_0$*:
assumes $t \setminus u \approx_0 t' \setminus u$ **and** $u \in \mathfrak{N}$
shows $t \approx_0 t'$
$\langle proof \rangle$

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

lemma *$Cong_0$-subst-left*:
assumes $t \approx_0 t'$ **and** $t \frown u$
shows $t' \frown u$ **and** $t \setminus u \approx_0 t' \setminus u$
$\langle proof \rangle$

Semi-congruence is not exactly substitutive for residuation on the right. Instead, the following weaker property is satisfied. Obtaining exact substitutivity on the right is the motivation for defining a coarser notion of congruence below.

lemma *$Cong_0$-subst-right*:
assumes $u \approx_0 u'$ **and** $t \frown u$
shows $t \frown u'$ **and** $(t \setminus u) \setminus (u' \setminus u) \approx_0 (t \setminus u') \setminus (u \setminus u')$
$\langle proof \rangle$

lemma *$Cong_0$-subst-Con*:
assumes $t \approx_0 t'$ **and** $u \approx_0 u'$
shows $t \frown u \longleftrightarrow t' \frown u'$
$\langle proof \rangle$

lemma *$Cong_0$-cancel-left*:
assumes $R.composite\text{-}of\ t\ u\ v$ **and** $R.composite\text{-}of\ t\ u'\ v'$ **and** $v \approx_0 v'$
shows $u \approx_0 u'$

$\langle proof \rangle$

lemma *Cong$_0$-iff*:
shows $t \approx_0 t' \longleftrightarrow$
$(\exists\, u\, u'\, v\, v'.\ u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge v \approx_0 v' \wedge$
$R.composite\text{-}of\ t\ u\ v \wedge R.composite\text{-}of\ t'\ u'\ v')$
$\langle proof \rangle$

lemma *diamond-commutes-upto-Cong$_0$*:
assumes $t \frown u$ **and** $R.composite\text{-}of\ t\ (u \setminus t)\ v$ **and** $R.composite\text{-}of\ u\ (t \setminus u)\ v'$
shows $v \approx_0 v'$
$\langle proof \rangle$

2.3.3 Congruence

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

definition *Cong* (**infix** ‹≈› *50*)
where $Cong\ t\ t' \equiv \exists\, u\, u'.\ u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge t \setminus u \approx_0 t' \setminus u'$

lemma *CongI* [*intro*]:
assumes $u \in \mathfrak{N}$ **and** $u' \in \mathfrak{N}$ **and** $t \setminus u \approx_0 t' \setminus u'$
shows $Cong\ t\ t'$
$\langle proof \rangle$

lemma *CongE* [*elim*]:
assumes $t \approx t'$
obtains $u\ u'$
where $u \in \mathfrak{N}$ **and** $u' \in \mathfrak{N}$ **and** $t \setminus u \approx_0 t' \setminus u'$
$\langle proof \rangle$

lemma *Cong-imp-arr*:
assumes $t \approx t'$
shows $R.arr\ t$ **and** $R.arr\ t'$
$\langle proof \rangle$

lemma *Cong-reflexive*:
assumes $R.arr\ t$
shows $t \approx t$
$\langle proof \rangle$

lemma *Cong-symmetric*:
assumes $t \approx t'$
shows $t' \approx t$
$\langle proof \rangle$

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

lemma *Cong-transitive* [*trans*]:
assumes $t \approx t''$ **and** $t'' \approx t'$
shows $t \approx t'$

$\langle proof \rangle$

lemma *Cong-closure-props*:
shows $t \approx u \Longrightarrow u \approx t$
and $[\![t \approx u;\ u \approx v]\!] \Longrightarrow t \approx v$
and $t \approx_0 u \Longrightarrow t \approx u$
and $[\![u \in \mathfrak{N};\ R.sources\ t = R.sources\ u]\!] \Longrightarrow t \approx t \setminus u$
$\langle proof \rangle$

lemma *Cong*$_0$*-implies-Cong*:
assumes $t \approx_0 t'$
shows $t \approx t'$
$\langle proof \rangle$

lemma *in-sources-respects-Cong*:
assumes $t \approx t'$ **and** $a \in R.sources\ t$ **and** $a' \in R.sources\ t'$
shows $a \approx a'$
$\langle proof \rangle$

lemma *in-targets-respects-Cong*:
assumes $t \approx t'$ **and** $b \in R.targets\ t$ **and** $b' \in R.targets\ t'$
shows $b \approx b'$
$\langle proof \rangle$

lemma *sources-are-Cong*:
assumes $a \in R.sources\ t$ **and** $a' \in R.sources\ t$
shows $a \approx a'$
$\langle proof \rangle$

lemma *targets-are-Cong*:
assumes $b \in R.targets\ t$ **and** $b' \in R.targets\ t$
shows $b \approx b'$
$\langle proof \rangle$

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

lemma *Resid-along-normal-preserves-reflects-con*:
assumes $u \in \mathfrak{N}$ **and** $R.sources\ t = R.sources\ u$
shows $t \setminus u \frown t' \setminus u \longleftrightarrow t \frown t'$
$\langle proof \rangle$

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

inductive *Cong'*
where $\bigwedge t\ u.\ Cong'\ t\ u \Longrightarrow Cong'\ u\ t$
| $\bigwedge t\ u\ v.\ [\![Cong'\ t\ u;\ Cong'\ u\ v]\!] \Longrightarrow Cong'\ t\ v$
| $\bigwedge t\ u.\ t \approx_0 u \Longrightarrow Cong'\ t\ u$
| $\bigwedge t\ u.\ [\![R.arr\ t;\ u \in \mathfrak{N};\ R.sources\ t = R.sources\ u]\!] \Longrightarrow Cong'\ t\ (t \setminus u)$

lemma *Cong'-if*:
shows $[\![u \in \mathfrak{N};\ u' \in \mathfrak{N};\ t \setminus u \approx_0 t' \setminus u']\!] \Longrightarrow \mathit{Cong'}\ t\ t'$
$\langle \mathit{proof} \rangle$

lemma *Cong-char*:
shows $\mathit{Cong}\ t\ t' \longleftrightarrow \mathit{Cong'}\ t\ t'$
$\langle \mathit{proof} \rangle$

lemma *normal-is-Cong-closed*:
assumes $t \in \mathfrak{N}$ **and** $t \approx t'$
shows $t' \in \mathfrak{N}$
$\langle \mathit{proof} \rangle$

2.3.4 Congruence Classes

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

definition *Cong-class* ($\langle\{\!|\text{-}|\!\}\rangle$)
where $\mathit{Cong\text{-}class}\ t \equiv \{t'.\ t \approx t'\}$

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

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

lemma *Cong-class-is-nonempty*:
assumes $\mathit{is\text{-}Cong\text{-}class}\ \mathcal{T}$
shows $\mathcal{T} \neq \{\}$
$\langle \mathit{proof} \rangle$

lemma *rep-in-Cong-class*:
assumes $\mathit{is\text{-}Cong\text{-}class}\ \mathcal{T}$
shows $\mathit{Cong\text{-}class\text{-}rep}\ \mathcal{T} \in \mathcal{T}$
$\langle \mathit{proof} \rangle$

lemma *arr-in-Cong-class*:
assumes $R.\mathit{arr}\ t$
shows $t \in \{\!|t|\!\}$
$\langle \mathit{proof} \rangle$

lemma *is-Cong-classI*:
assumes $R.\mathit{arr}\ t$
shows $\mathit{is\text{-}Cong\text{-}class}\ \{\!|t|\!\}$
$\langle \mathit{proof} \rangle$

lemma *is-Cong-classI'* [*intro*]:
assumes $\mathcal{T} \neq \{\}$
and $\bigwedge t\ t'.\ [\![t \in \mathcal{T};\ t' \in \mathcal{T}]\!] \Longrightarrow t \approx t'$

and $\bigwedge t\ t'.\ [\![t \in \mathcal{T};\ t' \approx t]\!] \Longrightarrow t' \in \mathcal{T}$
shows *is-Cong-class* $\mathcal{T}$
⟨*proof*⟩

lemma *Cong-class-memb-is-arr*:
assumes *is-Cong-class* $\mathcal{T}$ **and** $t \in \mathcal{T}$
shows *R.arr t*
⟨*proof*⟩

lemma *Cong-class-membs-are-Cong*:
assumes *is-Cong-class* $\mathcal{T}$ **and** $t \in \mathcal{T}$ **and** $t' \in \mathcal{T}$
shows *Cong* $t\ t'$
⟨*proof*⟩

lemma *Cong-class-eqI*:
assumes $t \approx t'$
shows $\{\!|t|\!\} = \{\!|t'|\!\}$
⟨*proof*⟩

lemma *Cong-class-eqI'*:
assumes *is-Cong-class* $\mathcal{T}$ **and** *is-Cong-class* $\mathcal{U}$ **and** $\mathcal{T} \cap \mathcal{U} \neq \{\}$
shows $\mathcal{T} = \mathcal{U}$
⟨*proof*⟩

lemma *is-Cong-classE* [*elim*]:
assumes *is-Cong-class* $\mathcal{T}$
and $[\![\mathcal{T} \neq \{\};\ \bigwedge t\ t'.\ [\![t \in \mathcal{T};\ t' \in \mathcal{T}]\!] \Longrightarrow t \approx t';\ \bigwedge t\ t'.\ [\![t \in \mathcal{T};\ t' \approx t]\!] \Longrightarrow t' \in \mathcal{T}]\!] \Longrightarrow T$
shows T
⟨*proof*⟩

lemma *Cong-class-rep* [*simp*]:
assumes *is-Cong-class* $\mathcal{T}$
shows $\{\!|$*Cong-class-rep* $\mathcal{T}|\!\} = \mathcal{T}$
⟨*proof*⟩

lemma *Cong-class-memb-Cong-rep*:
assumes *is-Cong-class* $\mathcal{T}$ **and** $t \in \mathcal{T}$
shows *Cong* t (*Cong-class-rep* $\mathcal{T}$)
⟨*proof*⟩

lemma *composite-of-normal-arr*:
shows $[\![$ *R.arr* t; $u \in \mathfrak{N}$; *R.composite-of* $u\ t\ t'$ $]\!] \Longrightarrow t' \approx t$
⟨*proof*⟩

lemma *composite-of-arr-normal*:
shows $[\![$ *arr* t; $u \in \mathfrak{N}$; *R.composite-of* $t\ u\ t'$ $]\!] \Longrightarrow t' \approx_0 t$
⟨*proof*⟩

end

2.3.5 Coherent Normal Sub-RTS's

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

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

context *normal-sub-rts*
begin

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

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

end

context *coherent-normal-sub-rts*
begin

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

lemma *coherent'*:
assumes $v \in \mathfrak{N}$ **and** $v' \in \mathfrak{N}$ **and** $w \in \mathfrak{N}$ **and** $w' \in \mathfrak{N}$
and *R.sources* v = *R.sources* w **and** *R.sources* v' = *R.sources* w'
and *R.targets* w = *R.targets* w' **and** $t \setminus v \approx_0 t' \setminus v'$
shows $t \setminus w \approx_0 t' \setminus w'$
$\langle$*proof*$\rangle$

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

lemma *Cong-subst*:
assumes $t \approx t'$ **and** $u \approx u'$ **and** $t \frown u$ **and** *R.sources* t' = *R.sources* u'
shows $t' \frown u'$ **and** $t \setminus u \approx t' \setminus u'$
$\langle$*proof*$\rangle$

lemma *Cong-subst-con*:
assumes $R.sources\ t = R.sources\ u$ **and** $R.sources\ t' = R.sources\ u'$
and $t \approx t'$ **and** $u \approx u'$
shows $t \frown u \longleftrightarrow t' \frown u'$
⟨*proof*⟩

lemma *$Cong_0$-composite-of-arr-normal*:
assumes $R.composite\text{-}of\ t\ u\ t'$ **and** $u \in \mathfrak{N}$
shows $t' \approx_0 t$
⟨*proof*⟩

lemma *Cong-composite-of-normal-arr*:
assumes $R.composite\text{-}of\ u\ t\ t'$ **and** $u \in \mathfrak{N}$
shows $t' \approx t$
⟨*proof*⟩

end

context *normal-sub-rts*
begin

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

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

end

2.3.6 Quotient by Coherent Normal Sub-RTS

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

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

definition *Resid* (**infix** ‹{\\}› *70*)
where $\mathcal{T}\ \{\!\!\{\backslash\}\!\!\}\ \mathcal{U} \equiv$
if $N.is\text{-}Cong\text{-}class\ \mathcal{T} \wedge N.is\text{-}Cong\text{-}class\ \mathcal{U} \wedge (\exists t\ u.\ t \in \mathcal{T} \wedge u \in \mathcal{U} \wedge t \frown u)$
then $N.Cong\text{-}class$

$(fst\ (SOME\ tu.\ fst\ tu \in \mathcal{T} \wedge snd\ tu \in \mathcal{U} \wedge fst\ tu \frown snd\ tu) \setminus$
$snd\ (SOME\ tu.\ fst\ tu \in \mathcal{T} \wedge snd\ tu \in \mathcal{U} \wedge fst\ tu \frown snd\ tu))$
else $\{\}$

sublocale *partial-magma Resid*
⟨*proof*⟩

lemma *is-partial-magma*:
shows *partial-magma Resid*
⟨*proof*⟩

lemma *null-char*:
shows $null = \{\}$
⟨*proof*⟩

lemma *Resid-by-members*:
assumes *N.is-Cong-class* $\mathcal{T}$ **and** *N.is-Cong-class* $\mathcal{U}$ **and** $t \in \mathcal{T}$ **and** $u \in \mathcal{U}$ **and** $t \frown u$
shows $\mathcal{T}\ \{\!\!\{\setminus\}\!\!\}\ \mathcal{U} = \{\!\!\{t \setminus u\}\!\!\}$
⟨*proof*⟩

abbreviation *Con* (**infix** ‹$\{\!\!\{\frown\}\!\!\}$› *50*)
where $\mathcal{T}\ \{\!\!\{\frown\}\!\!\}\ \mathcal{U} \equiv \mathcal{T}\ \{\!\!\{\setminus\}\!\!\}\ \mathcal{U} \neq \{\}$

lemma *Con-char*:
shows $\mathcal{T}\ \{\!\!\{\frown\}\!\!\}\ \mathcal{U} \longleftrightarrow$
N.is-Cong-class $\mathcal{T} \wedge$ *N.is-Cong-class* $\mathcal{U} \wedge (\exists\, t\ u.\ t \in \mathcal{T} \wedge u \in \mathcal{U} \wedge t \frown u)$
⟨*proof*⟩

lemma *Con-sym*:
assumes *Con* $\mathcal{T}\ \mathcal{U}$
shows *Con* $\mathcal{U}\ \mathcal{T}$
⟨*proof*⟩

lemma *is-Cong-class-Resid*:
assumes $\mathcal{T}\ \{\!\!\{\frown\}\!\!\}\ \mathcal{U}$
shows *N.is-Cong-class* $(\mathcal{T}\ \{\!\!\{\setminus\}\!\!\}\ \mathcal{U})$
⟨*proof*⟩

lemma *Con-witnesses*:
assumes $\mathcal{T}\ \{\!\!\{\frown\}\!\!\}\ \mathcal{U}$ **and** $t \in \mathcal{T}$ **and** $u \in \mathcal{U}$
shows $\exists\, v\ w.\ v \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge t \setminus v \frown u \setminus w$
⟨*proof*⟩

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

lemma *Arr-Resid*:
assumes *Con* $\mathcal{T}\ \mathcal{U}$
shows *Arr* $(\mathcal{T}\ \{\!\!\{\setminus\}\!\!\}\ \mathcal{U})$

⟨*proof*⟩

lemma *Cube*:
assumes *Con* $(\mathcal{V}\ \{\!\!\{\backslash\}\!\!\}\ \mathcal{T})\ (\mathcal{U}\ \{\!\!\{\backslash\}\!\!\}\ \mathcal{T})$
shows $(\mathcal{V}\ \{\!\!\{\backslash\}\!\!\}\ \mathcal{T})\ \{\!\!\{\backslash\}\!\!\}\ (\mathcal{U}\ \{\!\!\{\backslash\}\!\!\}\ \mathcal{T}) = (\mathcal{V}\ \{\!\!\{\backslash\}\!\!\}\ \mathcal{U})\ \{\!\!\{\backslash\}\!\!\}\ (\mathcal{T}\ \{\!\!\{\backslash\}\!\!\}\ \mathcal{U})$
⟨*proof*⟩

sublocale *residuation Resid*
⟨*proof*⟩

lemma *is-residuation*:
shows *residuation Resid*
⟨*proof*⟩

lemma *arr-char*:
shows *arr* $\mathcal{T} \longleftrightarrow$ *N.is-Cong-class* $\mathcal{T}$
⟨*proof*⟩

lemma *ide-char*:
shows *ide* $\mathcal{U} \longleftrightarrow$ *arr* $\mathcal{U} \wedge \mathcal{U} \cap \mathfrak{N} \neq \{\}$
⟨*proof*⟩

lemma *ide-char'*:
shows *ide* $\mathcal{A} \longleftrightarrow$ *arr* $\mathcal{A} \wedge \mathcal{A} \subseteq \mathfrak{N}$
⟨*proof*⟩

lemma *con-char*$_{QCN}$:
shows *con* $\mathcal{T}\ \mathcal{U} \longleftrightarrow$
N.is-Cong-class $\mathcal{T}$ $\wedge$ *N.is-Cong-class* $\mathcal{U}$ $\wedge$ $(\exists\, t\ u.\ t \in \mathcal{T} \wedge u \in \mathcal{U} \wedge t \frown u)$
⟨*proof*⟩

lemma *con-imp-coinitial-members-are-con*:
assumes *con* $\mathcal{T}\ \mathcal{U}$ **and** $t \in \mathcal{T}$ **and** $u \in \mathcal{U}$ **and** *R.sources* t = *R.sources* u
shows $t \frown u$
⟨*proof*⟩

sublocale *rts Resid*
⟨*proof*⟩

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

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

theorem *is-extensional-rts*:

shows *extensional-rts Resid*
⟨*proof*⟩

lemma *sources-char*$_{QCN}$:
shows *sources* $\mathcal{T} = \{\mathcal{A}.\ arr\ \mathcal{T} \wedge \mathcal{A} = \{a.\ \exists\, t\ a'.\ t \in \mathcal{T} \wedge a' \in R.sources\ t \wedge a' \approx a\}\}$
⟨*proof*⟩

lemma *targets-char*$_{QCN}$:
shows *targets* $\mathcal{T} = \{\mathcal{B}.\ arr\ \mathcal{T} \wedge \mathcal{B} = \mathcal{T}\ \{\!\!\{\backslash\}\!\!\}\ \mathcal{T}\}$
⟨*proof*⟩

lemma *src-char*$_{QCN}$:
shows *src* $\mathcal{T} = \{a.\ arr\ \mathcal{T} \wedge (\exists\, t\ a'.\ t \in \mathcal{T} \wedge a' \in R.sources\ t \wedge a' \approx a)\}$
⟨*proof*⟩

lemma *trg-char*$_{QCN}$:
shows *trg* $\mathcal{T} = \mathcal{T}\ \{\!\!\{\backslash\}\!\!\}\ \mathcal{T}$
⟨*proof*⟩

Quotient Map

abbreviation *quot*
where *quot* $t \equiv \{\!\!\{t\}\!\!\}$

sublocale *quot*: *simulation-to-extensional-rts resid Resid quot*
⟨*proof*⟩

lemma *quotient-is-simulation*:
shows *simulation resid Resid quot*
⟨*proof*⟩

lemma *ide-quot-normal*:
assumes $t \in \mathfrak{N}$
shows *ide* (*quot* t)
⟨*proof*⟩

If a simulation F from R to an extensional RTS B maps every element of $\mathfrak{N}$ to an identity, then it has a unique extension along the quotient map.

lemma *is-couniversal*:
assumes *extensional-rts B*
and *simulation resid B F*
and $\bigwedge t.\ t \in \mathfrak{N} \Longrightarrow$ *residuation.ide B* $(F\ t)$
shows $\exists!F'.$ *simulation Resid B* $F' \wedge F' \circ quot = F$
⟨*proof*⟩

definition *ext-to-quotient*
where *ext-to-quotient B F* $\equiv$ *THE* $F'.$ *simulation Resid B* $F' \wedge F' \circ quot = F$

lemma *ext-to-quotient-props*:

assumes *extensional-rts B*
and *simulation resid B F*
and $\bigwedge t.\ t \in \mathfrak{N} \Longrightarrow$ *residuation.ide B* $(F\ t)$
shows *simulation Resid B* (*ext-to-quotient B F*) **and** *ext-to-quotient B F* $\circ$ *quot* $= F$
⟨*proof*⟩

end

2.3.7 Identities form a Coherent Normal Sub-RTS

We now show that the collection of identities of an RTS form a coherent normal sub-RTS, and that the associated congruence $\approx$ coincides with $\sim$. Thus, every RTS can be factored by the relation $\sim$ to obtain an extensional RTS. Although we could have shown that fact much earlier, we have delayed proving it so that we could simply obtain it as a special case of our general quotient result without redundant work.

context *rts*
begin

interpretation *normal-sub-rts resid* ‹*Collect ide*›
⟨*proof*⟩

lemma *identities-form-normal-sub-rts*:
shows *normal-sub-rts resid* (*Collect ide*)
⟨*proof*⟩

interpretation *coherent-normal-sub-rts resid* ‹*Collect ide*›
⟨*proof*⟩

lemma *identities-form-coherent-normal-sub-rts*:
shows *coherent-normal-sub-rts resid* (*Collect ide*)
⟨*proof*⟩

lemma *Cong-iff-cong*:
shows *Cong t u* $\longleftrightarrow t \sim u$
⟨*proof*⟩

end

2.4 Paths

A *path* in an RTS is a nonempty list of arrows such that the set of targets of each arrow suitably matches the set of sources of its successor. The residuation on the given RTS extends inductively to a residuation on paths, so that paths also form an RTS. The append operation on lists yields a composite for each pair of compatible paths.

locale *paths-in-rts* =
R: *rts*
begin

type-synonym $'b\ arr = 'b\ list$

fun *Srcs*
where $Srcs\ [] = \{\}$
  $|\ Srcs\ [t] = R.sources\ t$
  $|\ Srcs\ (t\ \#\ T) = R.sources\ t$

fun *Trgs*
where $Trgs\ [] = \{\}$
  $|\ Trgs\ [t] = R.targets\ t$
  $|\ Trgs\ (t\ \#\ T) = Trgs\ T$

fun *Arr*
where $Arr\ [] = False$
  $|\ Arr\ [t] = R.arr\ t$
  $|\ Arr\ (t\ \#\ T) = (R.arr\ t \wedge Arr\ T \wedge R.targets\ t \subseteq Srcs\ T)$

fun *Ide*
where $Ide\ [] = False$
  $|\ Ide\ [t] = R.ide\ t$
  $|\ Ide\ (t\ \#\ T) = (R.ide\ t \wedge Ide\ T \wedge R.targets\ t \subseteq Srcs\ T)$

lemma *Arr-induct*:
assumes $\bigwedge t.\ Arr\ [t] \Longrightarrow P\ [t]$
and $\bigwedge t\ U.\ [\![Arr\ (t\ \#\ U);\ U \neq [];\ P\ U]\!] \Longrightarrow P\ (t\ \#\ U)$
shows $Arr\ T \Longrightarrow P\ T$
⟨*proof*⟩

lemma *Ide-induct*:
assumes $\bigwedge t.\ R.ide\ t \Longrightarrow P\ [t]$
and $\bigwedge t\ T.\ [\![R.ide\ t;\ R.targets\ t \subseteq Srcs\ T;\ P\ T]\!] \Longrightarrow P\ (t\ \#\ T)$
shows $Ide\ T \Longrightarrow P\ T$
⟨*proof*⟩

lemma *set-Arr-subset-arr*:
assumes $Arr\ T$
shows $set\ T \subseteq Collect\ R.arr$
  ⟨*proof*⟩

lemma *Arr-imp-arr-hd* [*simp*]:
assumes $Arr\ T$
shows $R.arr\ (hd\ T)$
  ⟨*proof*⟩

lemma *Arr-imp-arr-last* [*simp*]:
assumes $Arr\ T$
shows $R.arr\ (last\ T)$
  ⟨*proof*⟩

lemma *Arr-imp-Arr-tl* [*simp*]:
assumes *Arr T* **and** *tl T* $\neq$ []
shows *Arr* (*tl T*)
  ⟨*proof*⟩

lemma *set-Ide-subset-ide*:
assumes *Ide T*
shows *set T* $\subseteq$ *Collect R.ide*
  ⟨*proof*⟩

lemma *Ide-imp-Ide-hd* [*simp*]:
assumes *Ide T*
shows *R.ide* (*hd T*)
  ⟨*proof*⟩

lemma *Ide-imp-Ide-last* [*simp*]:
assumes *Ide T*
shows *R.ide* (*last T*)
  ⟨*proof*⟩

lemma *Ide-imp-Ide-tl* [*simp*]:
assumes *Ide T* **and** *tl T* $\neq$ []
shows *Ide* (*tl T*)
  ⟨*proof*⟩

lemma *Ide-implies-Arr*:
assumes *Ide T*
shows *Arr T*
  ⟨*proof*⟩

lemma *const-ide-is-Ide*:
shows ⟦*T* $\neq$ []; *R.ide* (*hd T*); *set T* $\subseteq$ {*hd T*}⟧ $\Longrightarrow$ *Ide T*
  ⟨*proof*⟩

lemma *Ide-char*:
shows *Ide T* $\longleftrightarrow$ *Arr T* $\wedge$ *set T* $\subseteq$ *Collect R.ide*
  ⟨*proof*⟩

lemma *IdeI* [*intro*]:
assumes *Arr T* **and** *set T* $\subseteq$ *Collect R.ide*
shows *Ide T*
  ⟨*proof*⟩

lemma *Arr-has-Src*:
shows *Arr T* $\Longrightarrow$ *Srcs T* $\neq$ {}
  ⟨*proof*⟩

lemma *Arr-has-Trg*:

shows $Arr\ T \Longrightarrow Trgs\ T \neq \{\}$
$\langle proof \rangle$

lemma *Srcs-are-ide*:
shows $Srcs\ T \subseteq Collect\ R.ide$
$\langle proof \rangle$

lemma *Trgs-are-ide*:
shows $Trgs\ T \subseteq Collect\ R.ide$
$\langle proof \rangle$

lemma *Srcs-are-con*:
assumes $a \in Srcs\ T$ **and** $a' \in Srcs\ T$
shows $a \frown a'$
$\langle proof \rangle$

lemma *Srcs-con-closed*:
assumes $a \in Srcs\ T$ **and** $R.ide\ a'$ **and** $a \frown a'$
shows $a' \in Srcs\ T$
$\langle proof \rangle$

lemma *Srcs-eqI*:
assumes $Srcs\ T \cap Srcs\ T' \neq \{\}$
shows $Srcs\ T = Srcs\ T'$
$\langle proof \rangle$

lemma *Trgs-are-con*:
shows $[\![b \in Trgs\ T;\ b' \in Trgs\ T]\!] \Longrightarrow b \frown b'$
$\langle proof \rangle$

lemma *Trgs-con-closed*:
shows $[\![b \in Trgs\ T;\ R.ide\ b';\ b \frown b']\!] \Longrightarrow b' \in Trgs\ T$
$\langle proof \rangle$

lemma *Trgs-eqI*:
assumes $Trgs\ T \cap Trgs\ T' \neq \{\}$
shows $Trgs\ T = Trgs\ T'$
$\langle proof \rangle$

lemma *Srcs-simp$_P$*:
assumes $Arr\ T$
shows $Srcs\ T = R.sources\ (hd\ T)$
$\langle proof \rangle$

lemma *Trgs-simp$_P$*:
shows $Arr\ T \Longrightarrow Trgs\ T = R.targets\ (last\ T)$
$\langle proof \rangle$

2.4.1 Residuation on Paths

It was more difficult than I thought to get a correct formal definition for residuation on paths and to prove things from it. Straightforward attempts to write a single recursive definition ran into problems with being able to prove termination, as well as getting the cases correct so that the domain of definition was symmetric. Eventually I found the definition below, which simplifies the termination proof to some extent through the use of two auxiliary functions, and which has a symmetric form that makes symmetry easier to prove. However, there was still some difficulty in proving the recursive expansions with respect to cons and append that I needed.

The following defines residuation of a single transition along a path, yielding a transition.

fun *Resid1x* (**infix** ‹$^1\backslash^*$› *70*)
where $t\ {}^1\backslash^*\ [] = R.null$
  | $t\ {}^1\backslash^*\ [u] = t \setminus u$
  | $t\ {}^1\backslash^*\ (u \# U) = (t \setminus u)\ {}^1\backslash^*\ U$

Next, we have residuation of a path along a single transition, yielding a path.

fun *Residx1* (**infix** ‹$^*\backslash^1$› *70*)
where $[]\ {}^*\backslash^1\ u = []$
  | $[t]\ {}^*\backslash^1\ u = (\mathit{if}\ t \frown u\ \mathit{then}\ [t \setminus u]\ \mathit{else}\ [])$
  | $(t \# T)\ {}^*\backslash^1\ u =$
    $(\mathit{if}\ t \frown u \wedge T\ {}^*\backslash^1\ (u \setminus t) \neq []\ \mathit{then}\ (t \setminus u) \# T\ {}^*\backslash^1\ (u \setminus t)\ \mathit{else}\ [])$

Finally, residuation of a path along a path, yielding a path.

function (*sequential*) *Resid* (**infix** ‹$^*\backslash^*$› *70*)
where $[]\ {}^*\backslash^*\ \text{-} = []$
  | $\text{-}\ {}^*\backslash^*\ [] = []$
  | $[t]\ {}^*\backslash^*\ [u] = (\mathit{if}\ t \frown u\ \mathit{then}\ [t \setminus u]\ \mathit{else}\ [])$
  | $[t]\ {}^*\backslash^*\ (u \# U) =$
    $(\mathit{if}\ t \frown u \wedge (t \setminus u)\ {}^1\backslash^*\ U \neq R.null\ \mathit{then}\ [(t \setminus u)\ {}^1\backslash^*\ U]\ \mathit{else}\ [])$
  | $(t \# T)\ {}^*\backslash^*\ [u] =$
    $(\mathit{if}\ t \frown u \wedge T\ {}^*\backslash^1\ (u \setminus t) \neq []\ \mathit{then}\ (t \setminus u) \# (T\ {}^*\backslash^1\ (u \setminus t))\ \mathit{else}\ [])$
  | $(t \# T)\ {}^*\backslash^*\ (u \# U) =$
    $(\mathit{if}\ t \frown u \wedge (t \setminus u)\ {}^1\backslash^*\ U \neq R.null \wedge$
      $(T\ {}^*\backslash^1\ (u \setminus t))\ {}^*\backslash^*\ (U\ {}^*\backslash^1\ (t \setminus u)) \neq []$
     $\mathit{then}\ (t \setminus u)\ {}^1\backslash^*\ U \# (T\ {}^*\backslash^1\ (u \setminus t))\ {}^*\backslash^*\ (U\ {}^*\backslash^1\ (t \setminus u))$
     $\mathit{else}\ [])$
 ⟨*proof*⟩

Residuation of a path along a single transition is length non-increasing. Actually, it is length-preserving, except in case the path and the transition are not consistent. We will show that later, but for now this is what we need to establish termination for (\).

lemma *length-Residx1*:
shows $\mathit{length}\ (T\ {}^*\backslash^1\ u) \leq \mathit{length}\ T$
⟨*proof*⟩

termination *Resid*
⟨*proof*⟩

lemma *Resid1x-null*:
shows $R.null\ {}^1\backslash^*\ T = R.null$
⟨*proof*⟩

lemma *Resid1x-ide*:
shows $[\![R.ide\ a;\ a\ {}^1\backslash^*\ T \neq R.null]\!] \Longrightarrow R.ide\ (a\ {}^1\backslash^*\ T)$
⟨*proof*⟩

abbreviation *Con* (**infix** ‹$^*\frown^*$› *50*)
where $T\ {}^*\frown^*\ U \equiv T\ {}^*\backslash^*\ U \neq []$

lemma *Con-sym1*:
shows $T\ {}^*\backslash^1\ u \neq [] \longleftrightarrow u\ {}^1\backslash^*\ T \neq R.null$
⟨*proof*⟩

lemma *Con-sym-ind*:
shows $\mathit{length}\ T + \mathit{length}\ U \leq n \Longrightarrow T\ {}^*\frown^*\ U \longleftrightarrow U\ {}^*\frown^*\ T$
⟨*proof*⟩

lemma *Con-sym*:
shows $T\ {}^*\frown^*\ U \longleftrightarrow U\ {}^*\frown^*\ T$
⟨*proof*⟩

lemma *Residx1-as-Resid*:
shows $T\ {}^*\backslash^1\ u = T\ {}^*\backslash^*\ [u]$
⟨*proof*⟩

lemma *Resid1x-as-Resid'*:
shows $t\ {}^1\backslash^*\ U = (\mathit{if}\ [t]\ {}^*\backslash^*\ U \neq []\ \mathit{then}\ \mathit{hd}\ ([t]\ {}^*\backslash^*\ U)\ \mathit{else}\ R.null)$
⟨*proof*⟩

The following recursive expansion for consistency of paths is an intermediate result that is not yet quite in the form we really want.

lemma *Con-rec*:
shows $[t]\ {}^*\frown^*\ [u] \longleftrightarrow t \frown u$
and $T \neq [] \Longrightarrow t\ \#\ T\ {}^*\frown^*\ [u] \longleftrightarrow t \frown u \wedge T\ {}^*\frown^*\ [u \setminus t]$
and $U \neq [] \Longrightarrow [t]\ {}^*\frown^*\ (u\ \#\ U) \longleftrightarrow t \frown u \wedge [t \setminus u]\ {}^*\frown^*\ U$
and $[\![T \neq [];\ U \neq []]\!] \Longrightarrow$
$t\ \#\ T\ {}^*\frown^*\ u\ \#\ U \longleftrightarrow t \frown u \wedge T\ {}^*\frown^*\ [u \setminus t] \wedge [t \setminus u]\ {}^*\frown^*\ U \wedge$
$T\ {}^*\backslash^*\ [u \setminus t]\ {}^*\frown^*\ U\ {}^*\backslash^*\ [t \setminus u]$
⟨*proof*⟩

This version is a more appealing form of the previously proved fact *Resid1x-as-Resid'*.

lemma *Resid1x-as-Resid*:
assumes $[t]\ {}^*\backslash^*\ U \neq []$

shows $[t]\ {}^*\backslash^*\ U = [t\ {}^1\backslash^*\ U]$
⟨*proof*⟩

The following is an intermediate version of a recursive expansion for residuation, to be improved subsequently.

lemma *Resid-rec*:
shows [*simp*]: $[t]\ {}^*\frown^*\ [u] \Longrightarrow [t]\ {}^*\backslash^*\ [u] = [t \backslash u]$
and $[\![T \neq [\,];\ t \# T\ {}^*\frown^*\ [u]]\!] \Longrightarrow (t \# T)\ {}^*\backslash^*\ [u] = (t \backslash u) \# (T\ {}^*\backslash^*\ [u \backslash t])$
and $[\![U \neq [\,];\ \mathit{Con}\ [t]\ (u \# U)]\!] \Longrightarrow [t]\ {}^*\backslash^*\ (u \# U) = [t \backslash u]\ {}^*\backslash^*\ U$
and $[\![T \neq [\,];\ U \neq [\,];\ \mathit{Con}\ (t \# T)\ (u \# U)]\!] \Longrightarrow$
$(t \# T)\ {}^*\backslash^*\ (u \# U) = ([t \backslash u]\ {}^*\backslash^*\ U)\ @\ ((T\ {}^*\backslash^*\ [u \backslash t])\ {}^*\backslash^*\ (U\ {}^*\backslash^*\ [t \backslash u]))$
⟨*proof*⟩

For consistent paths, residuation is length-preserving.

lemma *length-Resid-ind*:
shows $[\![\mathit{length}\ T + \mathit{length}\ U \leq n;\ T\ {}^*\frown^*\ U]\!] \Longrightarrow \mathit{length}\ (T\ {}^*\backslash^*\ U) = \mathit{length}\ T$
⟨*proof*⟩

lemma *length-Resid*:
assumes $T\ {}^*\frown^*\ U$
shows $\mathit{length}\ (T\ {}^*\backslash^*\ U) = \mathit{length}\ T$
⟨*proof*⟩

lemma *Con-initial-left*:
shows $t \# T\ {}^*\frown^*\ U \Longrightarrow [t]\ {}^*\frown^*\ U$
⟨*proof*⟩

lemma *Con-initial-right*:
shows $T\ {}^*\frown^*\ u \# U \Longrightarrow T\ {}^*\frown^*\ [u]$
⟨*proof*⟩

lemma *Resid-cons-ind*:
shows $[\![T \neq [\,];\ U \neq [\,];\ \mathit{length}\ T + \mathit{length}\ U \leq n]\!] \Longrightarrow$
$(\forall t.\ t \# T\ {}^*\frown^*\ U \longleftrightarrow [t]\ {}^*\frown^*\ U \wedge T\ {}^*\frown^*\ U\ {}^*\backslash^*\ [t]) \wedge$
$(\forall u.\ T\ {}^*\frown^*\ u \# U \longleftrightarrow T\ {}^*\frown^*\ [u] \wedge T\ {}^*\backslash^*\ [u]\ {}^*\frown^*\ U) \wedge$
$(\forall t.\ t \# T\ {}^*\frown^*\ U \longrightarrow (t \# T)\ {}^*\backslash^*\ U = [t]\ {}^*\backslash^*\ U\ @\ T\ {}^*\backslash^*\ (U\ {}^*\backslash^*\ [t])) \wedge$
$(\forall u.\ T\ {}^*\frown^*\ u \# U \longrightarrow T\ {}^*\backslash^*\ (u \# U) = (T\ {}^*\backslash^*\ [u])\ {}^*\backslash^*\ U)$
⟨*proof*⟩

The following are the final versions of recursive expansion for consistency and residuation on paths. These are what I really wanted the original definitions to look like, but if this is tried, then *Con* and *Resid* end up having to be mutually recursive, expressing the definitions so that they are single-valued becomes an issue, and proving termination is more problematic.

lemma *Con-cons*:
assumes $T \neq [\,]$ **and** $U \neq [\,]$
shows $t \# T\ {}^*\frown^*\ U \longleftrightarrow [t]\ {}^*\frown^*\ U \wedge T\ {}^*\frown^*\ U\ {}^*\backslash^*\ [t]$
and $T\ {}^*\frown^*\ u \# U \longleftrightarrow T\ {}^*\frown^*\ [u] \wedge T\ {}^*\backslash^*\ [u]\ {}^*\frown^*\ U$
⟨*proof*⟩

lemma *Con-consI* [*intro*, *simp*]:
shows $[\![T \neq []; U \neq []; [t] \ {}^*\frown^* \ U; \ T \ {}^*\frown^* \ U \ {}^*\backslash^* \ [t]]\!] \Longrightarrow t \ \# \ T \ {}^*\frown^* \ U$
and $[\![T \neq []; U \neq []; T \ {}^*\frown^* \ [u]; \ T \ {}^*\backslash^* \ [u] \ {}^*\frown^* \ U]\!] \Longrightarrow T \ {}^*\frown^* \ u \ \# \ U$
⟨*proof*⟩

lemma *Resid-cons*:
assumes $U \neq []$
shows $t \ \# \ T \ {}^*\frown^* \ U \Longrightarrow (t \ \# \ T) \ {}^*\backslash^* \ U = ([t] \ {}^*\backslash^* \ U) \ @ \ (T \ {}^*\backslash^* \ (U \ {}^*\backslash^* \ [t]))$
and $T \ {}^*\frown^* \ u \ \# \ U \Longrightarrow T \ {}^*\backslash^* \ (u \ \# \ U) = (T \ {}^*\backslash^* \ [u]) \ {}^*\backslash^* \ U$
⟨*proof*⟩

The following expansion of residuation with respect to the first argument is stated in terms of the more primitive cons, rather than list append, but as a result ${}^1\backslash^*$ has to be used.

lemma *Resid-cons'*:
assumes $T \neq []$
shows $t \ \# \ T \ {}^*\frown^* \ U \Longrightarrow (t \ \# \ T) \ {}^*\backslash^* \ U = (t \ {}^1\backslash^* \ U) \ \# \ (T \ {}^*\backslash^* \ (U \ {}^*\backslash^* \ [t]))$
⟨*proof*⟩

lemma *Srcs-Resid-Arr-single*:
assumes $T \ {}^*\frown^* \ [u]$
shows $Srcs \ (T \ {}^*\backslash^* \ [u]) = R.targets \ u$
⟨*proof*⟩

lemma *Srcs-Resid-single-Arr*:
shows $[u] \ {}^*\frown^* \ T \Longrightarrow Srcs \ ([u] \ {}^*\backslash^* \ T) = Trgs \ T$
⟨*proof*⟩

lemma *Trgs-Resid-sym-Arr-single*:
shows $T \ {}^*\frown^* \ [u] \Longrightarrow Trgs \ (T \ {}^*\backslash^* \ [u]) = Trgs \ ([u] \ {}^*\backslash^* \ T)$
⟨*proof*⟩

lemma *Srcs-Resid* [*simp*]:
shows $T \ {}^*\frown^* \ U \Longrightarrow Srcs \ (T \ {}^*\backslash^* \ U) = Trgs \ U$
⟨*proof*⟩

lemma *Trgs-Resid-sym* [*simp*]:
shows $T \ {}^*\frown^* \ U \Longrightarrow Trgs \ (T \ {}^*\backslash^* \ U) = Trgs \ (U \ {}^*\backslash^* \ T)$
⟨*proof*⟩

lemma *img-Resid-Srcs*:
shows $Arr \ T \Longrightarrow (\lambda a. \ [a] \ {}^*\backslash^* \ T) \ ` \ Srcs \ T \subseteq (\lambda b. \ [b]) \ ` \ Trgs \ T$
⟨*proof*⟩

lemma *Resid-Arr-Src*:
shows $[\![Arr \ T; \ a \in Srcs \ T]\!] \Longrightarrow T \ {}^*\backslash^* \ [a] = T$
⟨*proof*⟩

lemma *Con-single-ide-ind*:
shows $R.ide\ a \Longrightarrow [a]\ {}^*\frown^*\ T \longleftrightarrow Arr\ T \wedge a \in Srcs\ T$
⟨*proof*⟩

lemma *Con-single-ide-iff*:
assumes $R.ide\ a$
shows $[a]\ {}^*\frown^*\ T \longleftrightarrow Arr\ T \wedge a \in Srcs\ T$
⟨*proof*⟩

lemma *Con-single-ideI* [*intro*]:
assumes $R.ide\ a$ **and** $Arr\ T$ **and** $a \in Srcs\ T$
shows $[a]\ {}^*\frown^*\ T$ **and** $T\ {}^*\frown^*\ [a]$
⟨*proof*⟩

lemma *Resid-single-ide*:
assumes $R.ide\ a$ **and** $[a]\ {}^*\frown^*\ T$
shows $[a]\ {}^*\backslash^*\ T \in (\lambda b.\ [b])$ ‘ $Trgs\ T$ **and** [*simp*]: $T\ {}^*\backslash^*\ [a] = T$
⟨*proof*⟩

lemma *Resid-Arr-Ide-ind*:
shows $[\![Ide\ A;\ T\ {}^*\frown^*\ A]\!] \Longrightarrow T\ {}^*\backslash^*\ A = T$
⟨*proof*⟩

lemma *Resid-Ide-Arr-ind*:
shows $[\![Ide\ A;\ A\ {}^*\frown^*\ T]\!] \Longrightarrow Ide\ (A\ {}^*\backslash^*\ T)$
⟨*proof*⟩

lemma *Resid-Ide*:
assumes $Ide\ A$ **and** $A\ {}^*\frown^*\ T$
shows $T\ {}^*\backslash^*\ A = T$ **and** $Ide\ (A\ {}^*\backslash^*\ T)$
⟨*proof*⟩

lemma *Con-Ide-iff*:
shows $Ide\ A \Longrightarrow A\ {}^*\frown^*\ T \longleftrightarrow Arr\ T \wedge Srcs\ T = Srcs\ A$
⟨*proof*⟩

lemma *Con-IdeI*:
assumes $Ide\ A$ **and** $Arr\ T$ **and** $Srcs\ T = Srcs\ A$
shows $A\ {}^*\frown^*\ T$ **and** $T\ {}^*\frown^*\ A$
⟨*proof*⟩

lemma *Con-Arr-self*:
shows $Arr\ T \Longrightarrow T\ {}^*\frown^*\ T$
⟨*proof*⟩

lemma *Resid-Arr-self*:
shows $Arr\ T \Longrightarrow Ide\ (T\ {}^*\backslash^*\ T)$
⟨*proof*⟩

lemma *Con-imp-eq-Srcs*:
assumes $T \ {}^*\frown^* \ U$
shows *Srcs* T = *Srcs* U
⟨*proof*⟩

lemma *Arr-iff-Con-self*:
shows *Arr* $T \longleftrightarrow T \ {}^*\frown^* \ T$
⟨*proof*⟩

lemma *Arr-Resid-single*:
shows $T \ {}^*\frown^* \ [u] \Longrightarrow$ *Arr* $(T \ {}^*\backslash^* \ [u])$
⟨*proof*⟩

lemma *Con-imp-Arr-Resid*:
shows $T \ {}^*\frown^* \ U \Longrightarrow$ *Arr* $(T \ {}^*\backslash^* \ U)$
⟨*proof*⟩

lemma *Cube-ind*:
shows $[\![T \ {}^*\frown^* \ U; \ V \ {}^*\frown^* \ T;$ *length* T + *length* U + *length* $V \leq n]\!] \Longrightarrow$
$(V \ {}^*\backslash^* \ T \ {}^*\frown^* \ U \ {}^*\backslash^* \ T \longleftrightarrow V \ {}^*\backslash^* \ U \ {}^*\frown^* \ T \ {}^*\backslash^* \ U) \wedge$
$(V \ {}^*\backslash^* \ T \ {}^*\frown^* \ U \ {}^*\backslash^* \ T \longrightarrow$
$(V \ {}^*\backslash^* \ T) \ {}^*\backslash^* \ (U \ {}^*\backslash^* \ T) = (V \ {}^*\backslash^* \ U) \ {}^*\backslash^* \ (T \ {}^*\backslash^* \ U))$
⟨*proof*⟩

lemma *Cube*:
shows $T \ {}^*\backslash^* \ U \ {}^*\frown^* \ V \ {}^*\backslash^* \ U \longleftrightarrow T \ {}^*\backslash^* \ V \ {}^*\frown^* \ U \ {}^*\backslash^* \ V$
and $T \ {}^*\backslash^* \ U \ {}^*\frown^* \ V \ {}^*\backslash^* \ U \Longrightarrow (T \ {}^*\backslash^* \ U) \ {}^*\backslash^* \ (V \ {}^*\backslash^* \ U) = (T \ {}^*\backslash^* \ V) \ {}^*\backslash^* \ (U \ {}^*\backslash^* \ V)$
⟨*proof*⟩

lemma *Con-implies-Arr*:
assumes $T \ {}^*\frown^* \ U$
shows *Arr* T **and** *Arr* U
⟨*proof*⟩

sublocale *partial-magma Resid*
⟨*proof*⟩

lemma *is-partial-magma*:
shows *partial-magma Resid*
⟨*proof*⟩

lemma *null-char*:
shows *null* = []
⟨*proof*⟩

sublocale *residuation Resid*
⟨*proof*⟩

lemma *is-residuation*:
shows *residuation Resid*
⟨*proof*⟩

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

lemma *arrI*$_P$ [*intro*]:
assumes *Arr T*
shows *arr T*
⟨*proof*⟩

lemma *ide-char*:
shows *ide T* $\longleftrightarrow$ *Ide T*
⟨*proof*⟩

lemma *con-char*:
shows *con T U* $\longleftrightarrow$ *Con T U*
⟨*proof*⟩

lemma *conI*$_P$ [*intro*]:
assumes *Con T U*
shows *con T U*
⟨*proof*⟩

sublocale *rts Resid*
⟨*proof*⟩

theorem *is-rts*:
shows *rts Resid*
⟨*proof*⟩

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

lemma *sources-char*$_P$:
shows *sources T* = {*A*. *Ide A* $\wedge$ *Arr T* $\wedge$ *Srcs A* = *Srcs T*}
⟨*proof*⟩

lemma *sources-cons*:
shows *Arr* (*t* # *T*) $\Longrightarrow$ *sources* (*t* # *T*) = *sources* [*t*]
⟨*proof*⟩

lemma *targets-char*$_P$:
shows *targets T* = {*B*. *Ide B* $\wedge$ *Arr T* $\wedge$ *Srcs B* = *Trgs T*}
⟨*proof*⟩

lemma *seq-char'*:

shows $seq\ T\ U \longleftrightarrow Arr\ T \wedge Arr\ U \wedge Trgs\ T \cap Srcs\ U \neq \{\}$
⟨*proof*⟩

lemma *seq-char*:
shows $seq\ T\ U \longleftrightarrow Arr\ T \wedge Arr\ U \wedge Trgs\ T = Srcs\ U$
⟨*proof*⟩

lemma $seqI_P$ [*intro*]:
assumes $Arr\ T$ **and** $Arr\ U$ **and** $Trgs\ T \cap Srcs\ U \neq \{\}$
shows $seq\ T\ U$
⟨*proof*⟩

lemma *coinitial-char*:
shows $coinitial\ T\ U \Longrightarrow Arr\ T \wedge Arr\ U \wedge Srcs\ T = Srcs\ U$
and $Arr\ T \wedge Arr\ U \wedge Srcs\ T \cap Srcs\ U \neq \{\} \Longrightarrow coinitial\ T\ U$
⟨*proof*⟩

lemma $coinitialI_P$ [*intro*]:
assumes $Arr\ T$ **and** $Arr\ U$ **and** $Srcs\ T \cap Srcs\ U \neq \{\}$
shows $coinitial\ T\ U$
⟨*proof*⟩

lemma *Ide-imp-sources-eq-targets*:
assumes $Ide\ T$
shows $sources\ T = targets\ T$
⟨*proof*⟩

2.4.2 Inclusion Map

Inclusion of an RTS to the RTS of its paths.

abbreviation *incl*
where $incl \equiv \lambda t.\ if\ R.arr\ t\ then\ [t]\ else\ null$

sublocale *incl*: *simulation resid Resid incl*
⟨*proof*⟩

lemma *incl-is-simulation*:
shows *simulation resid Resid incl*
⟨*proof*⟩

lemma *incl-is-injective*:
shows *inj-on incl* (*Collect R.arr*)
⟨*proof*⟩

lemma *reflects-con*:
assumes $incl\ t\ {}^*\frown^*\ incl\ u$
shows $t \frown u$
⟨*proof*⟩

end

2.4.3 Composites of Paths

The RTS of paths has composites, given by the append operation on lists.

context *paths-in-rts*
begin

lemma *Srcs-append* [*simp*]:
assumes $T \neq []$
shows $Srcs\ (T\ @\ U) = Srcs\ T$
⟨*proof*⟩

lemma *Trgs-append* [*simp*]:
shows $U \neq [] \Longrightarrow Trgs\ (T\ @\ U) = Trgs\ U$
⟨*proof*⟩

lemma *seq-implies-Trgs-eq-Srcs*:
shows $[\![Arr\ T;\ Arr\ U;\ Trgs\ T \subseteq Srcs\ U]\!] \Longrightarrow Trgs\ T = Srcs\ U$
⟨*proof*⟩

lemma *Arr-append-iff*$_P$:
shows $[\![T \neq [];\ U \neq []]\!] \Longrightarrow Arr\ (T\ @\ U) \longleftrightarrow Arr\ T \wedge Arr\ U \wedge Trgs\ T \subseteq Srcs\ U$
⟨*proof*⟩

lemma *Arr-consI*$_P$ [*intro*, *simp*]:
assumes $R.arr\ t$ **and** $Arr\ U$ **and** $R.targets\ t \subseteq Srcs\ U$
shows $Arr\ (t\ \#\ U)$
⟨*proof*⟩

lemma *Arr-appendI*$_P$ [*intro*, *simp*]:
assumes $Arr\ T$ **and** $Arr\ U$ **and** $Trgs\ T \subseteq Srcs\ U$
shows $Arr\ (T\ @\ U)$
⟨*proof*⟩

lemma *Arr-appendE*$_P$ [*elim*]:
assumes $Arr\ (T\ @\ U)$ **and** $T \neq []$ **and** $U \neq []$
and $[\![Arr\ T;\ Arr\ U;\ Trgs\ T = Srcs\ U]\!] \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

lemma *Ide-append-iff*$_P$:
shows $[\![T \neq [];\ U \neq []]\!] \Longrightarrow Ide\ (T\ @\ U) \longleftrightarrow Ide\ T \wedge Ide\ U \wedge Trgs\ T \subseteq Srcs\ U$
⟨*proof*⟩

lemma *Ide-appendI*$_P$ [*intro*, *simp*]:
assumes $Ide\ T$ **and** $Ide\ U$ **and** $Trgs\ T \subseteq Srcs\ U$
shows $Ide\ (T\ @\ U)$
⟨*proof*⟩

lemma *Resid-append-ind*:
shows $[\![T \neq []; U \neq []; V \neq []]\!] \Longrightarrow$
$(V @ T \;{}^*\frown^*\; U \longleftrightarrow V \;{}^*\frown^*\; U \wedge T \;{}^*\frown^*\; U \;{}^*\backslash^*\; V) \wedge$
$(T \;{}^*\frown^*\; V @ U \longleftrightarrow T \;{}^*\frown^*\; V \wedge T \;{}^*\backslash^*\; V \;{}^*\frown^*\; U) \wedge$
$(V @ T \;{}^*\frown^*\; U \longrightarrow (V @ T) \;{}^*\backslash^*\; U = V \;{}^*\backslash^*\; U @ T \;{}^*\backslash^*\; (U \;{}^*\backslash^*\; V)) \wedge$
$(T \;{}^*\frown^*\; V @ U \longrightarrow T \;{}^*\backslash^*\; (V @ U) = (T \;{}^*\backslash^*\; V) \;{}^*\backslash^*\; U)$
⟨*proof*⟩

lemma *Con-append*:
assumes $T \neq []$ **and** $U \neq []$ **and** $V \neq []$
shows $T @ U \;{}^*\frown^*\; V \longleftrightarrow T \;{}^*\frown^*\; V \wedge U \;{}^*\frown^*\; V \;{}^*\backslash^*\; T$
and $T \;{}^*\frown^*\; U @ V \longleftrightarrow T \;{}^*\frown^*\; U \wedge T \;{}^*\backslash^*\; U \;{}^*\frown^*\; V$
⟨*proof*⟩

lemma *Con-appendI* [*intro*]:
shows $[\![T \;{}^*\frown^*\; V; U \;{}^*\frown^*\; V \;{}^*\backslash^*\; T]\!] \Longrightarrow T @ U \;{}^*\frown^*\; V$
and $[\![T \;{}^*\frown^*\; U; T \;{}^*\backslash^*\; U \;{}^*\frown^*\; V]\!] \Longrightarrow T \;{}^*\frown^*\; U @ V$
⟨*proof*⟩

lemma *Resid-append* [*intro*, *simp*]:
shows $[\![T \neq []; T @ U \;{}^*\frown^*\; V]\!] \Longrightarrow (T @ U) \;{}^*\backslash^*\; V = (T \;{}^*\backslash^*\; V) @ (U \;{}^*\backslash^*\; (V \;{}^*\backslash^*\; T))$
and $[\![U \neq []; V \neq []; T \;{}^*\frown^*\; U @ V]\!] \Longrightarrow T \;{}^*\backslash^*\; (U @ V) = (T \;{}^*\backslash^*\; U) \;{}^*\backslash^*\; V$
⟨*proof*⟩

lemma *Resid-append2* [*simp*]:
assumes $T \neq []$ **and** $U \neq []$ **and** $V \neq []$ **and** $W \neq []$
and $T @ U \;{}^*\frown^*\; V @ W$
shows $(T @ U) \;{}^*\backslash^*\; (V @ W) =$
$(T \;{}^*\backslash^*\; V) \;{}^*\backslash^*\; W @ (U \;{}^*\backslash^*\; (V \;{}^*\backslash^*\; T)) \;{}^*\backslash^*\; (W \;{}^*\backslash^*\; (T \;{}^*\backslash^*\; V))$
⟨*proof*⟩

lemma *append-is-composite-of*:
assumes *seq* T U
shows *composite-of* T U $(T @ U)$
⟨*proof*⟩

sublocale *rts-with-composites Resid*
⟨*proof*⟩

theorem *is-rts-with-composites*:
shows *rts-with-composites Resid*
⟨*proof*⟩

lemma *arr-append* [*intro*, *simp*]:
assumes *seq* T U
shows *arr* $(T @ U)$
⟨*proof*⟩

lemma *arr-append-imp-seq*:
assumes $T \neq []$ **and** $U \neq []$ **and** $arr\ (T\ @\ U)$
shows $seq\ T\ U$
⟨*proof*⟩

lemma *sources-append* [*simp*]:
assumes $seq\ T\ U$
shows $sources\ (T\ @\ U) = sources\ T$
⟨*proof*⟩

lemma *targets-append* [*simp*]:
assumes $seq\ T\ U$
shows $targets\ (T\ @\ U) = targets\ U$
⟨*proof*⟩

lemma *cong-respects-seq$_P$*:
assumes $seq\ T\ U$ **and** $T\ {}^*{\sim}^*\ T'$ **and** $U\ {}^*{\sim}^*\ U'$
shows $seq\ T'\ U'$
⟨*proof*⟩

lemma *cong-append* [*intro*]:
assumes $seq\ T\ U$ **and** $T\ {}^*{\sim}^*\ T'$ **and** $U\ {}^*{\sim}^*\ U'$
shows $T\ @\ U\ {}^*{\sim}^*\ T'\ @\ U'$
⟨*proof*⟩

lemma *cong-cons* [*intro*]:
assumes $seq\ [t]\ U$ **and** $t \sim t'$ **and** $U\ {}^*{\sim}^*\ U'$
shows $t\ \#\ U\ {}^*{\sim}^*\ t'\ \#\ U'$
⟨*proof*⟩

lemma *cong-append-ideI* [*intro*]:
assumes $seq\ T\ U$
shows $ide\ T \Longrightarrow T\ @\ U\ {}^*{\sim}^*\ U$ **and** $ide\ U \Longrightarrow T\ @\ U\ {}^*{\sim}^*\ T$
and $ide\ T \Longrightarrow U\ {}^*{\sim}^*\ T\ @\ U$ **and** $ide\ U \Longrightarrow T\ {}^*{\sim}^*\ T\ @\ U$
⟨*proof*⟩

lemma *cong-cons-ideI* [*intro*]:
assumes $seq\ [t]\ U$ **and** $R.ide\ t$
shows $t\ \#\ U\ {}^*{\sim}^*\ U$ **and** $U\ {}^*{\sim}^*\ t\ \#\ U$
⟨*proof*⟩

lemma *prfx-decomp*:
assumes $[t]\ {}^*{\lesssim}^*\ [u]$
shows $[t]\ @\ [u \setminus t]\ {}^*{\sim}^*\ [u]$
⟨*proof*⟩

lemma *composite-of-single-single*:
assumes $R.composite\text{-}of\ t\ u\ v$

shows *composite-of* $[t]$ $[u]$ $([t] @ [u])$
⟨*proof*⟩

end

2.4.4 Paths in a Weakly Extensional RTS

locale *paths-in-weakly-extensional-rts* =
R: *weakly-extensional-rts* +
paths-in-rts
begin

lemma *ex-un-Src*:
assumes *Arr T*
shows $\exists!a.\ a \in Srcs\ T$
⟨*proof*⟩

fun *Src*
where $Src\ T = R.src\ (hd\ T)$

lemma $Srcs\text{-}simp_{PWE}$:
assumes *Arr T*
shows $Srcs\ T = \{Src\ T\}$
⟨*proof*⟩

lemma *ex-un-Trg*:
assumes *Arr T*
shows $\exists!b.\ b \in Trgs\ T$
⟨*proof*⟩

fun *Trg*
where $Trg\ [] = R.null$
| $Trg\ [t] = R.trg\ t$
| $Trg\ (t\ \#\ T) = Trg\ T$

lemma *Trg-simp* [*simp*]:
shows $T \neq [] \Longrightarrow Trg\ T = R.trg\ (last\ T)$
⟨*proof*⟩

lemma $Trgs\text{-}simp_{PWE}$ [*simp*]:
assumes *Arr T*
shows $Trgs\ T = \{Trg\ T\}$
⟨*proof*⟩

lemma *Src-resid* [*simp*]:
assumes $T\ {}^*\frown^*\ U$
shows $Src\ (T\ {}^*\backslash^*\ U) = Trg\ U$
⟨*proof*⟩

lemma *Trg-resid-sym*:
assumes $T \ {}^*\frown^* \ U$
shows $Trg\ (T \ {}^*\backslash^* \ U) = Trg\ (U \ {}^*\backslash^* \ T)$
  ⟨*proof*⟩

lemma *Src-append* [*simp*]:
assumes *seq T U*
shows $Src\ (T\ @\ U) = Src\ T$
  ⟨*proof*⟩

lemma *Trg-append* [*simp*]:
assumes *seq T U*
shows $Trg\ (T\ @\ U) = Trg\ U$
  ⟨*proof*⟩

lemma *Arr-append-iff*$_{PWE}$:
assumes $T \neq []$ **and** $U \neq []$
shows $Arr\ (T\ @\ U) \longleftrightarrow Arr\ T \wedge Arr\ U \wedge Trg\ T = Src\ U$
  ⟨*proof*⟩

lemma *Arr-consI*$_{PWE}$ [*intro*, *simp*]:
assumes *R.arr t* **and** *Arr U* **and** $R.trg\ t = Src\ U$
shows $Arr\ (t\ \#\ U)$
  ⟨*proof*⟩

lemma *Arr-consE* [*elim*]:
assumes $Arr\ (t\ \#\ U)$
and $[\![R.arr\ t;\ U \neq [] \Longrightarrow Arr\ U;\ U \neq [] \Longrightarrow R.trg\ t = Src\ U]\!] \Longrightarrow thesis$
shows *thesis*
  ⟨*proof*⟩

lemma *Arr-appendI*$_{PWE}$ [*intro*, *simp*]:
assumes *Arr T* **and** *Arr U* **and** $Trg\ T = Src\ U$
shows $Arr\ (T\ @\ U)$
  ⟨*proof*⟩

lemma *Arr-appendE*$_{PWE}$ [*elim*]:
assumes $Arr\ (T\ @\ U)$ **and** $T \neq []$ **and** $U \neq []$
and $[\![Arr\ T;\ Arr\ U;\ Trg\ T = Src\ U]\!] \Longrightarrow thesis$
shows *thesis*
  ⟨*proof*⟩

lemma *Ide-append-iff*$_{PWE}$:
assumes $T \neq []$ **and** $U \neq []$
shows $Ide\ (T\ @\ U) \longleftrightarrow Ide\ T \wedge Ide\ U \wedge Trg\ T = Src\ U$
  ⟨*proof*⟩

lemma *Ide-appendI*$_{PWE}$ [*intro*, *simp*]:
assumes *Ide T* **and** *Ide U* **and** $Trg\ T = Src\ U$

shows *Ide* $(T\ @\ U)$
⟨*proof*⟩

lemma *Ide-appendE* [*elim*]:
assumes *Ide* $(T\ @\ U)$ **and** $T \neq []$ **and** $U \neq []$
and $[\![Ide\ T;\ Ide\ U;\ Trg\ T = Src\ U]\!] \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

lemma *Ide-consI* [*intro*, *simp*]:
assumes *R.ide t* **and** *Ide U* **and** $R.trg\ t = Src\ U$
shows *Ide* $(t\ \#\ U)$
⟨*proof*⟩

lemma *Ide-consE* [*elim*]:
assumes *Ide* $(t\ \#\ U)$
and $[\![R.ide\ t;\ U \neq [] \Longrightarrow Ide\ U;\ U \neq [] \Longrightarrow R.trg\ t = Src\ U]\!] \Longrightarrow thesis$
shows *thesis*
⟨*proof*⟩

lemma *Ide-imp-Src-eq-Trg*:
assumes *Ide T*
shows $Src\ T = Trg\ T$
⟨*proof*⟩

end

2.4.5 Paths in a Confluent RTS

Here we show that confluence of an RTS extends to confluence of the RTS of its paths.

locale *paths-in-confluent-rts* =
paths-in-rts +
R: *confluent-rts*
begin

lemma *confluence-single*:
assumes $\bigwedge t\ u.\ R.coinitial\ t\ u \Longrightarrow t \frown u$
shows $[\![R.arr\ t;\ Arr\ U;\ R.sources\ t = Srcs\ U]\!] \Longrightarrow [t]\ {}^*\!\frown^*\ U$
⟨*proof*⟩

lemma *confluence-ind*:
shows $[\![Arr\ T;\ Arr\ U;\ Srcs\ T = Srcs\ U]\!] \Longrightarrow T\ {}^*\!\frown^*\ U$
⟨*proof*⟩

lemma $confluence_P$:
assumes *coinitial T U*
shows *con T U*
⟨*proof*⟩

sublocale *confluent-rts Resid*
⟨*proof*⟩

lemma *is-confluent-rts*:
shows *confluent-rts Resid*
⟨*proof*⟩

end

2.4.6 Simulations Lift to Paths

In this section we show that a simulation from RTS A to RTS B determines a simulation from the RTS of paths in A to the RTS of paths in B. In other words, the path-RTS construction is functorial with respect to simulation.

context *simulation*
begin

interpretation P_A: *paths-in-rts A*
⟨*proof*⟩
interpretation P_B: *paths-in-rts B*
⟨*proof*⟩

lemma *map-Resid-single*:
shows $P_A.con\ T\ [u] \Longrightarrow map\ F\ (P_A.Resid\ T\ [u]) = P_B.Resid\ (map\ F\ T)\ [F\ u]$
⟨*proof*⟩

lemma *map-Resid*:
shows $P_A.con\ T\ U \Longrightarrow map\ F\ (P_A.Resid\ T\ U) = P_B.Resid\ (map\ F\ T)\ (map\ F\ U)$
⟨*proof*⟩

lemma *preserves-paths*:
shows $P_A.Arr\ T \Longrightarrow P_B.Arr\ (map\ F\ T)$
⟨*proof*⟩

interpretation *Fx*: *simulation* $P_A.Resid\ P_B.Resid$ ‹$\lambda T.\ if\ P_A.Arr\ T\ then\ map\ F\ T\ else\ []$›
⟨*proof*⟩

lemma *lifts-to-paths*:
shows *simulation* $P_A.Resid\ P_B.Resid\ (\lambda T.\ if\ P_A.Arr\ T\ then\ map\ F\ T\ else\ [])$
⟨*proof*⟩

end

2.4.7 Normal Sub-RTS's Lift to Paths

Here we show that a normal sub-RTS N of an RTS R lifts to a normal sub-RTS of the RTS of paths in N, and that it is coherent if N is.

locale *paths-in-rts-with-normal* =

R: *rts* +
N: *normal-sub-rts* +
paths-in-rts
begin

We define a "normal path" to be a path that consists entirely of normal transitions. We show that the collection of all normal paths is a normal sub-RTS of the RTS of paths.

definition *NPath*
where $NPath\ T \equiv (Arr\ T \land set\ T \subseteq \mathfrak{N})$

lemma *Ide-implies-NPath*:
assumes *Ide T*
shows *NPath T*
⟨*proof*⟩

lemma *NPath-implies-Arr*:
assumes *NPath T*
shows *Arr T*
⟨*proof*⟩

lemma *NPath-append*:
assumes $T \neq []$ **and** $U \neq []$
shows $NPath\ (T\ @\ U) \longleftrightarrow NPath\ T \land NPath\ U \land Trgs\ T \subseteq Srcs\ U$
⟨*proof*⟩

lemma *NPath-appendI* [*intro*, *simp*]:
assumes *NPath T* **and** *NPath U* **and** $Trgs\ T \subseteq Srcs\ U$
shows *NPath* (*T* @ *U*)
⟨*proof*⟩

lemma *NPath-Resid-single-Arr*:
shows $[\![t \in \mathfrak{N};\ Arr\ U;\ R.sources\ t = Srcs\ U]\!] \Longrightarrow NPath\ (Resid\ [t]\ U)$
⟨*proof*⟩

lemma *NPath-Resid-Arr-single*:
shows $[\![\ NPath\ T;\ R.arr\ u;\ Srcs\ T = R.sources\ u\ ]\!] \Longrightarrow NPath\ (Resid\ T\ [u])$
⟨*proof*⟩

lemma *NPath-Resid* [*simp*]:
shows $[\![NPath\ T;\ Arr\ U;\ Srcs\ T = Srcs\ U]\!] \Longrightarrow NPath\ (T\ {}^*\backslash^*\ U)$
⟨*proof*⟩

lemma *Backward-stable-single*:
shows $[\![NPath\ U;\ NPath\ ([t]\ {}^*\backslash^*\ U)]\!] \Longrightarrow NPath\ [t]$
⟨*proof*⟩

lemma *Backward-stable*:
shows $[\![NPath\ U;\ NPath\ (T\ {}^*\backslash^*\ U)]\!] \Longrightarrow NPath\ T$
⟨*proof*⟩

sublocale *normal-sub-rts Resid* ‹*Collect NPath*›
⟨*proof*⟩

theorem *normal-extends-to-paths*:
shows *normal-sub-rts Resid* (*Collect NPath*)
⟨*proof*⟩

lemma *Resid-NPath-preserves-reflects-Con*:
assumes *NPath U* **and** *Srcs T* = *Srcs U*
shows $T \ {}^*\backslash^*\ U \ {}^*\frown^*\ T' \ {}^*\backslash^*\ U \longleftrightarrow T \ {}^*\frown^*\ T'$
⟨*proof*⟩

notation $Cong_0$ (**infix** ‹$\approx^*{}_0$› *50*)
notation *Cong* (**infix** ‹$\approx^*$› *50*)

lemma $Cong_0$*-cancel-left*$_{CS}$:
assumes $T @ U \approx^*{}_0\ T @ U'$ **and** $T \neq []$ **and** $U \neq []$ **and** $U' \neq []$
shows $U \approx^*{}_0\ U'$
⟨*proof*⟩

lemma *Srcs-respects-Cong*:
assumes $T \approx^* T'$ **and** $a \in$ *Srcs T* **and** $a' \in$ *Srcs T'*
shows $[a] \approx^* [a']$
⟨*proof*⟩

lemma *Trgs-respects-Cong*:
assumes $T \approx^* T'$ **and** $b \in$ *Trgs T* **and** $b' \in$ *Trgs T'*
shows $[b] \approx^* [b']$
⟨*proof*⟩

lemma $Cong_0$*-append-resid-NPath*:
assumes *NPath* ($T \ {}^*\backslash^*\ U$)
shows $Cong_0$ (T @ ($U \ {}^*\backslash^*\ T$)) U
⟨*proof*⟩

end

locale *paths-in-rts-with-coherent-normal* =
R: *rts* +
N: *coherent-normal-sub-rts* +
paths-in-rts
begin

sublocale *paths-in-rts-with-normal resid* $\mathfrak{N}$ ⟨*proof*⟩

notation $Cong_0$ (**infix** ‹$\approx^*{}_0$› *50*)
notation *Cong* (**infix** ‹$\approx^*$› *50*)

Since composites of normal transitions are assumed to exist, normal paths can be "folded" by composition down to single transitions.

lemma *NPath-folding*:
shows $NPath\ U \Longrightarrow \exists\, u.\ u \in \mathfrak{N} \wedge R.sources\ u = Srcs\ U \wedge R.targets\ u = Trgs\ U \wedge$
$(\forall\, t.\ con\ [t]\ U \longrightarrow [t]\ {}^*\backslash^*\ U \approx^*{}_0\ [t \setminus u])$
⟨*proof*⟩

Coherence for single transitions extends inductively to paths.

lemma *Coherent-single*:
assumes $R.arr\ t$ **and** $NPath\ U$ **and** $NPath\ U'$
and $R.sources\ t = Srcs\ U$ **and** $Srcs\ U = Srcs\ U'$ **and** $Trgs\ U = Trgs\ U'$
shows $[t]\ {}^*\backslash^*\ U \approx^*{}_0\ [t]\ {}^*\backslash^*\ U'$
⟨*proof*⟩

lemma *Coherent*:
shows $[\![\ Arr\ T;\ NPath\ U;\ NPath\ U';\ Srcs\ T = Srcs\ U;$
$Srcs\ U = Srcs\ U';\ Trgs\ U = Trgs\ U'\ ]\!]$
$\Longrightarrow T\ {}^*\backslash^*\ U \approx^*{}_0\ T\ {}^*\backslash^*\ U'$
⟨*proof*⟩

sublocale *rts-with-composites Resid*
⟨*proof*⟩

sublocale *coherent-normal-sub-rts Resid* ‹*Collect NPath*›
⟨*proof*⟩

theorem *coherent-normal-extends-to-paths*:
shows *coherent-normal-sub-rts Resid* (*Collect NPath*)
⟨*proof*⟩

lemma $Cong_0$*-append-Arr-NPath*:
assumes $T \neq []$ **and** $Arr\ (T\ @\ U)$ **and** $NPath\ U$
shows $Cong_0\ (T\ @\ U)\ T$
⟨*proof*⟩

lemma *Cong-append-NPath-Arr*:
assumes $T \neq []$ **and** $Arr\ (U\ @\ T)$ **and** $NPath\ U$
shows $U\ @\ T \approx^*\ T$
⟨*proof*⟩

Permutation Congruence

Here we show that ${}^*\sim^*$ coincides with "permutation congruence": the least congruence respecting composition that relates $[t,\ u \setminus t]$ and $[u,\ t \setminus u]$ whenever $t \frown u$ and that relates $T\ @\ [b]$ and T whenever b is an identity such that $seq\ T\ [b]$.

inductive *PCong*
where $Arr\ T \Longrightarrow PCong\ T\ T$
| $PCong\ T\ U \Longrightarrow PCong\ U\ T$

| $\llbracket PCong\ T\ U;\ PCong\ U\ V \rrbracket \Longrightarrow PCong\ T\ V$
| $\llbracket seq\ T\ U;\ PCong\ T\ T';\ PCong\ U\ U' \rrbracket \Longrightarrow PCong\ (T\ @\ U)\ (T'\ @\ U')$
| $\llbracket seq\ T\ [b];\ R.ide\ b \rrbracket \Longrightarrow PCong\ (T\ @\ [b])\ T$
| $t \frown u \Longrightarrow PCong\ [t,\ u \setminus t]\ [u,\ t \setminus u]$

lemmas *PCong.intros(3)* [*trans*]

lemma *PCong-append-Ide*:
shows $\llbracket seq\ T\ B;\ Ide\ B \rrbracket \Longrightarrow PCong\ (T\ @\ B)\ T$
⟨*proof*⟩

lemma *PCong-imp-Cong*:
shows $PCong\ T\ U \Longrightarrow T\ {}^*\!\sim^*\ U$
⟨*proof*⟩

lemma *PCong-permute-single*:
shows $[t]\ {}^*\!\frown^*\ U \Longrightarrow PCong\ ([t]\ @\ (U\ {}^*\!\setminus^*\ [t]))\ (U\ @\ ([t]\ {}^*\!\setminus^*\ U))$
⟨*proof*⟩

lemma *PCong-permute*:
shows $T\ {}^*\!\frown^*\ U \Longrightarrow PCong\ (T\ @\ (U\ {}^*\!\setminus^*\ T))\ (U\ @\ (T\ {}^*\!\setminus^*\ U))$
⟨*proof*⟩

lemma *Cong-imp-PCong*:
assumes $T\ {}^*\!\sim^*\ U$
shows $PCong\ T\ U$
⟨*proof*⟩

lemma *Cong-iff-PCong*:
shows $T\ {}^*\!\sim^*\ U \longleftrightarrow PCong\ T\ U$
⟨*proof*⟩

end

2.5 Composite Completion

The RTS of paths in an RTS factors via the coherent normal sub-RTS of identity paths into an extensional RTS with composites, which can be regarded as a "composite completion" of the original RTS.

locale *composite-completion* =
R: *rts R*
for *R* :: *'a resid*
begin

type-synonym *'b arr* = *'b list set*

interpretation *N*: *coherent-normal-sub-rts R* ‹*Collect R.ide*›
⟨*proof*⟩

sublocale P: *paths-in-rts-with-coherent-normal R ‹Collect R.ide›* ⟨*proof*⟩
sublocale Q: *quotient-by-coherent-normal P.Resid ‹Collect P.NPath›* ⟨*proof*⟩

definition *resid* (**infix** ‹$\{\!\!\{^*\backslash^*\}\!\!\}$› *70*)
where *resid* $\equiv$ *Q.Resid*

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

notation *con* (**infix** ‹$\{\!\!\{^*\frown^*\}\!\!\}$› *50*)
notation *prfx* (**infix** ‹$\{\!\!\{^*\lesssim^*\}\!\!\}$› *50*)

notation *P.Resid* (**infix** ‹$^*\backslash^*$› *70*)
notation *P.Con* (**infix** ‹$^*\frown^*$› *50*)
notation *P.Cong* (**infix** ‹$^*\approx^*$› *50*)
notation $P.Cong_0$ (**infix** ‹$^*\approx_0{}^*$› *50*)
notation *P.Cong-class* (‹$\{\!\!\{$-$\}\!\!\}$›)

lemma *P-ide-iff-NPath*:
shows *P.ide T* $\longleftrightarrow$ *P.NPath T*
⟨*proof*⟩

lemma *Cong-eq-Cong*$_0$:
shows $T\ {}^*\approx^*\ T' \longleftrightarrow T\ {}^*\approx_0{}^*\ T'$
⟨*proof*⟩

lemma *Srcs-respects-Cong*:
assumes $T\ {}^*\approx^*\ T'$
shows *P.Srcs T* = *P.Srcs T'*
⟨*proof*⟩

lemma *sources-respects-Cong*:
assumes $T\ {}^*\approx^*\ T'$
shows *P.sources T* = *P.sources T'*
⟨*proof*⟩

lemma *Trgs-respects-Cong*:
assumes $T\ {}^*\approx^*\ T'$
shows *P.Trgs T* = *P.Trgs T'*
⟨*proof*⟩

lemma *targets-respects-Cong*:
assumes $T\ {}^*\approx^*\ T'$
shows *P.targets T* = *P.targets T'*
⟨*proof*⟩

lemma *ide-char*$_{CC}$:
shows *ide* $\mathcal{T} \longleftrightarrow$ *arr* $\mathcal{T} \wedge (\forall T.\ T \in \mathcal{T} \longrightarrow P.Ide\ T)$
⟨*proof*⟩

lemma *con-char*$_{CC}$:
shows $\mathcal{T}$ $\{\!\!\{{}^*\frown^*\}\!\!\}$ $\mathcal{U} \longleftrightarrow$ *arr* $\mathcal{T} \wedge$ *arr* $\mathcal{U} \wedge$ *P.Cong-class-rep* $\mathcal{T}$ $^*\frown^*$ *P.Cong-class-rep* $\mathcal{U}$
⟨*proof*⟩

lemma *con-char*$_{CC}$*'*:
shows $\mathcal{T}$ $\{\!\!\{{}^*\frown^*\}\!\!\}$ $\mathcal{U} \longleftrightarrow$ *arr* $\mathcal{T} \wedge$ *arr* $\mathcal{U} \wedge (\forall\, T\ U.\ T \in \mathcal{T} \wedge U \in \mathcal{U} \longrightarrow T\ {}^*\frown^*\ U)$
⟨*proof*⟩

lemma *resid-char*:
shows $\mathcal{T}$ $\{\!\!\{{}^*\backslash^*\}\!\!\}$ $\mathcal{U}$ =
(*if* $\mathcal{T}$ $\{\!\!\{{}^*\frown^*\}\!\!\}$ $\mathcal{U}$ *then* $\{\!\!\{$*P.Cong-class-rep* $\mathcal{T}$ $^*\backslash^*$ *P.Cong-class-rep* $\mathcal{U}\}\!\!\}$ *else* {})
⟨*proof*⟩

lemma *resid-simp*:
assumes $\mathcal{T}$ $\{\!\!\{{}^*\frown^*\}\!\!\}$ $\mathcal{U}$ **and** $T \in \mathcal{T}$ **and** $U \in \mathcal{U}$
shows $\mathcal{T}$ $\{\!\!\{{}^*\backslash^*\}\!\!\}$ $\mathcal{U}$ = $\{\!\!\{$*P.Resid T U*$\}\!\!\}$
⟨*proof*⟩

lemma *src-char'*:
shows *src* $\mathcal{T}$ = {*A*. *arr* $\mathcal{T} \wedge$ *P.Ide A* $\wedge$ *P.Srcs* (*P.Cong-class-rep* $\mathcal{T}$) = *P.Srcs A*}
⟨*proof*⟩

lemma *src-char*:
shows *src* $\mathcal{T}$ = {*A*. *arr* $\mathcal{T} \wedge$ *P.Ide A* $\wedge$ ($\forall$ *T*. $T \in \mathcal{T} \longrightarrow$ *P.Srcs T* = *P.Srcs A*)}
⟨*proof*⟩

lemma *trg-char'*:
shows *trg* $\mathcal{T}$ = {*B*. *arr* $\mathcal{T} \wedge$ *P.Ide B* $\wedge$ *P.Trgs* (*P.Cong-class-rep* $\mathcal{T}$) = *P.Srcs B*}
⟨*proof*⟩

lemma *trg-char*:
shows *trg* $\mathcal{T}$ = {*B*. *arr* $\mathcal{T} \wedge$ *P.Ide B* $\wedge$ ($\forall$ *T*. $T \in \mathcal{T} \longrightarrow$ *P.Trgs T* = *P.Srcs B*)}
⟨*proof*⟩

lemma *prfx-char*:
shows $\mathcal{T}$ $\{\!\!\{{}^*\lesssim^*\}\!\!\}$ $\mathcal{U} \longleftrightarrow$ *arr* $\mathcal{T} \wedge$ *arr* $\mathcal{U} \wedge (\forall\, T\ U.\ T \in \mathcal{T} \wedge U \in \mathcal{U} \longrightarrow$ *P.prfx T U*)
⟨*proof*⟩

lemma *quotient-reflects-con*:
assumes *con* (*Q.quot t*) (*Q.quot u*)
shows *P.con t u*
⟨*proof*⟩

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

sublocale *extensional-rts-with-composites resid*

⟨*proof*⟩

notation *comp* (**infixr** $\{\!\!\{^*\cdot^*\}\!\!\}$ *55*)

2.5.1 Inclusion Map

definition *incl*
where $incl \equiv Q.quot \circ P.incl$

sublocale *incl*: *simulation R resid incl*
⟨*proof*⟩
sublocale *incl*: *simulation-to-extensional-rts R resid incl* ⟨*proof*⟩

lemma *incl-is-simulation*:
shows *simulation R resid incl*
⟨*proof*⟩

lemma *incl-simp* [*simp*]:
shows $incl\ t = \{\!\!\{[t]\}\!\!\}$
⟨*proof*⟩

lemma *incl-reflects-con*:
assumes $incl\ t\ \{\!\!\{^*\frown^*\}\!\!\}\ incl\ u$
shows $R.con\ t\ u$
⟨*proof*⟩

lemma *cong-iff-eq-incl*:
assumes $R.arr\ t$ **and** $R.arr\ u$
shows $incl\ t = incl\ u \longleftrightarrow R.cong\ t\ u$
⟨*proof*⟩

lemma *incl-cancel-left*:
assumes $transformation\ X\ R\ F\ G\ T$ **and** $transformation\ X\ R\ F'\ G'\ T'$
and $extensional\text{-}rts\ R$
and $incl \circ T = incl \circ T'$
shows $T = T'$
⟨*proof*⟩

The inclusion is surjective on identities.

lemma *img-incl-ide*:
shows $incl\ `\ (Collect\ R.ide) = Collect\ ide$
⟨*proof*⟩

end

2.5.2 Composite Completion of a Weakly Extensional RTS

locale *composite-completion-of-weakly-extensional-rts* =
R: *weakly-extensional-rts R* +
composite-completion

begin

sublocale *P*: *paths-in-weakly-extensional-rts R* ⟨*proof*⟩
sublocale *incl*: *simulation-between-weakly-extensional-rts R resid incl* ⟨*proof*⟩

notation *comp* (**infixr** ⦃*·*⦄ *55*)

lemma *src-char*$_{CCWE}$:
shows $src\ \mathcal{T} = (if\ arr\ \mathcal{T}\ then\ incl\ (P.Src\ (P.Cong\text{-}class\text{-}rep\ \mathcal{T}))\ else\ null)$
⟨*proof*⟩

lemma *trg-char*$_{CCWE}$:
shows $trg\ \mathcal{T} = (if\ arr\ \mathcal{T}\ then\ incl\ (P.Trg\ (P.Cong\text{-}class\text{-}rep\ \mathcal{T}))\ else\ null)$
⟨*proof*⟩

When applied to a weakly extensional RTS, the composite completion construction does not identify any states that are distinct in the original RTS.

lemma *incl-injective-on-ide*:
shows *inj-on incl* (*Collect R.ide*)
⟨*proof*⟩

When applied to a weakly extensional RTS, the composite completion construction is a bijection between the states of the original RTS and the states of its completion.

lemma *incl-bijective-on-ide*:
shows $incl \in Collect\ R.ide \rightarrow Collect\ ide$
and $(\lambda\mathcal{A}.\ P.Src\ (P.Cong\text{-}class\text{-}rep\ \mathcal{A})) \in Collect\ ide \rightarrow Collect\ R.ide$
and $\bigwedge a.\ R.ide\ a \Longrightarrow (\lambda\mathcal{A}.\ P.Src\ (P.Cong\text{-}class\text{-}rep\ \mathcal{A}))\ (incl\ a) = a$
and $\bigwedge\mathcal{A}.\ ide\ \mathcal{A} \Longrightarrow incl\ ((\lambda\mathcal{A}.\ P.Src\ (P.Cong\text{-}class\text{-}rep\ \mathcal{A}))\ \mathcal{A}) = \mathcal{A}$
and *bij-betw incl* (*Collect R.ide*) (*Collect ide*)
and $bij\text{-}betw\ (\lambda\mathcal{A}.\ P.Src\ (P.Cong\text{-}class\text{-}rep\ \mathcal{A}))\ (Collect\ ide)\ (Collect\ R.ide)$
⟨*proof*⟩

end

2.5.3 Composite Completion of an Extensional RTS

locale *composite-completion-of-extensional-rts* =
R: *extensional-rts R* +
composite-completion
begin

sublocale *composite-completion-of-weakly-extensional-rts* ⟨*proof*⟩
sublocale *incl*: *simulation-between-extensional-rts R resid incl* ⟨*proof*⟩

end

2.5.4 Freeness of Composite Completion

In this section we show that the composite completion construction is free: any simulation from RTS A to an extensional RTS with composites B extends uniquely to a simulation on the composite completion of A.

type-synonym *'a comp* = *'a* ⇒ *'a* ⇒ *'a*

locale *rts-with-chosen-composites* =
rts +
fixes *comp* :: *'a comp* (**infixr** · *55*)
assumes *comp-extensionality-ax*: $\bigwedge t\ u :: 'a.\ \neg\ seq\ t\ u \Longrightarrow t \cdot u = null$
and *composite-of-comp-ax*: $\bigwedge t\ u\ v :: 'a.\ seq\ t\ u \Longrightarrow composite\text{-}of\ t\ u\ (t \cdot u)$
and *comp-assoc-ax*: $\bigwedge t\ u\ v :: 'a.\ [\![seq\ t\ u;\ seq\ u\ v]\!] \Longrightarrow (t \cdot u) \cdot v = t \cdot (u \cdot v)$
and *resid-comp-right-ax*: $t \cdot u \frown w \Longrightarrow w \setminus (t \cdot u) = (w \setminus t) \setminus u$
and *resid-comp-left-ax*: $(t \cdot u) \setminus w = (t \setminus w) \cdot (u \setminus (w \setminus t))$
begin

lemma *comp-assoc*$_{CC}$:
shows $t \cdot u \cdot v = (t \cdot u) \cdot v$
⟨*proof*⟩

lemma *comp-null*$_{CC}$:
shows $t \cdot null = null$ **and** $null \cdot t = null$
⟨*proof*⟩

lemma *composable-iff-arr-comp*$_{CC}$:
shows $composable\ t\ u \longleftrightarrow arr\ (t \cdot u)$
⟨*proof*⟩

lemma *composable-iff-comp-not-null*$_{CC}$:
shows $composable\ t\ u \longleftrightarrow t \cdot u \neq null$
⟨*proof*⟩

lemma *con-comp-iff*$_{CC}$:
shows $w \frown t \cdot u \longleftrightarrow composable\ t\ u \wedge w \setminus t \frown u$
⟨*proof*⟩

lemma *con-compI*$_{CC}$ [*intro*]:
assumes $composable\ t\ u$ **and** $w \setminus t \frown u$
shows $w \frown t \cdot u$ **and** $t \cdot u \frown w$
⟨*proof*⟩

sublocale *rts-with-composites resid*
⟨*proof*⟩

end

context *paths-in-weakly-extensional-rts*
begin

abbreviation *Comp*
where $Comp\ T\ U \equiv if\ seq\ T\ U\ then\ T\ @\ U\ else\ null$

sublocale *rts-with-chosen-composites Resid Comp*
⟨*proof*⟩

lemma *extends-to-rts-with-chosen-composites*:
shows *rts-with-chosen-composites Resid Comp*
⟨*proof*⟩

end

context *extensional-rts-with-composites*
begin

lemma *extends-to-rts-with-chosen-composites*:
shows *rts-with-chosen-composites resid comp*
⟨*proof*⟩

sublocale *rts-with-chosen-composites resid comp*
⟨*proof*⟩

end

locale *extension-to-paths* =
A: *rts A* +
B: *rts-with-chosen-composites B* $comp_B$ +
F: *simulation A B F* +
paths-in-rts A
for *A* :: *'a resid* (**infix** ‹$\backslash_A$› *70*)
and *B* :: *'b resid* (**infix** ‹$\backslash_B$› *70*)
and $comp_B$ :: *'b comp* (**infixr** ‹$\cdot_B$› *55*)
and *F* :: *'a* ⇒ *'b*
begin

notation *Resid* (**infix** ‹${}^*\backslash_A{}^*$› *70*)
notation *Resid1x* (**infix** ‹${}^1\backslash_A{}^*$› *70*)
notation *Residx1* (**infix** ‹${}^*\backslash_A{}^1$› *70*)
notation *Con* (**infix** ‹${}^*\frown_A{}^*$› *70*)
notation *B.con* (**infix** ‹$\frown_B$› *50*)

fun *map*
where *map* [] = *B.null*
| *map* [*t*] = *F t*
| *map* (*t* # *T*) = (*if arr* (*t* # *T*) *then F t* $\cdot_B$ *map T else B.null*)

lemma *map-o-incl-eq*:
shows *map* (*incl t*) = *F t*
⟨*proof*⟩

lemma *extensionality*:
shows ¬ *arr T* ⟹ *map T* = *B.null*

$\langle proof \rangle$

lemma *preserves-comp*:
shows $[\![T \neq []; U \neq []; Arr\ (T\ @\ U)]\!] \Longrightarrow map\ (T\ @\ U) = map\ T \cdot_B map\ U$
$\langle proof \rangle$

lemma *preserves-arr-ind*:
shows $[\![arr\ T; a \in Srcs\ T]\!] \Longrightarrow B.arr\ (map\ T) \wedge F\ a \in B.sources\ (map\ T)$
$\langle proof \rangle$

lemma *preserves-arr*:
shows $arr\ T \Longrightarrow B.arr\ (map\ T)$
$\langle proof \rangle$

lemma *preserves-sources*:
assumes $arr\ T$ **and** $a \in Srcs\ T$
shows $F\ a \in B.sources\ (map\ T)$
$\langle proof \rangle$

lemma *preserves-targets*:
shows $[\![arr\ T; b \in Trgs\ T]\!] \Longrightarrow F\ b \in B.targets\ (map\ T)$
$\langle proof \rangle$

lemma *preserves-Resid1x-ind*:
shows $t\ {}^1\backslash_A{}^*\ U \neq A.null \Longrightarrow F\ t \frown_B map\ U \wedge F\ (t\ {}^1\backslash_A{}^*\ U) = F\ t \backslash_B map\ U$
$\langle proof \rangle$

lemma *preserves-Residx1-ind*:
shows $U\ {}^*\backslash_A{}^1\ t \neq [] \Longrightarrow map\ U \frown_B F\ t \wedge map\ (U\ {}^*\backslash_A{}^1\ t) = map\ U \backslash_B F\ t$
$\langle proof \rangle$

lemma *preserves-resid-ind*:
shows $con\ T\ U \Longrightarrow map\ T \frown_B map\ U \wedge map\ (T\ {}^*\backslash_A{}^*\ U) = map\ T \backslash_B map\ U$
$\langle proof \rangle$

lemma *preserves-con*:
assumes $con\ T\ U$
shows $map\ T \frown_B map\ U$
$\langle proof \rangle$

lemma *preserves-resid*:
assumes $con\ T\ U$
shows $map\ (T\ {}^*\backslash_A{}^*\ U) = map\ T \backslash_B map\ U$
$\langle proof \rangle$

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

lemma *is-extension*:

shows *map* ∘ *incl* = *F*
⟨*proof*⟩

lemma *is-universal*:
shows *simulation Resid B map* **and** *map* ∘ *incl* = *F*
and $\bigwedge F'$. ⟦*simulation Resid B F'*; *F' o incl* = *F*⟧
$\Longrightarrow \forall\, T.\ arr\ T \longrightarrow B.cong\ (F'\ T)\ (map\ T)$
⟨*proof*⟩

end

lemma *extension-to-paths-comp*:
assumes *rts-with-chosen-composites B* $comp_B$
and *rts-with-chosen-composites C* $comp_C$
and *simulation A B F* **and** *simulation B C G*
and $\bigwedge t\ u.\ rts.composable\ B\ t\ u \Longrightarrow G\ (comp_B\ t\ u) = comp_C\ (G\ t)\ (G\ u)$
shows *extension-to-paths.map A C* $comp_C$ (*G* ∘ *F*) = *G* ∘ *extension-to-paths.map A B* $comp_B$
F
⟨*proof*⟩

locale *extension-to-composite-completion* =
A: *rts A* +
B: *extensional-rts-with-composites B* +
simulation A B F
for *A* :: *'a resid* (**infix** ‹$\backslash_A$› *70*)
and *B* :: *'b resid* (**infix** ‹$\backslash_B$› *70*)
and *F* :: *'a* ⇒ *'b*
begin

interpretation *N*: *coherent-normal-sub-rts A* ‹*Collect A.ide*›
⟨*proof*⟩
sublocale *P*: *paths-in-rts-with-coherent-normal A* ‹*Collect A.ide*› ⟨*proof*⟩
sublocale *Q*: *quotient-by-coherent-normal P.Resid* ‹*Collect P.NPath*› ⟨*proof*⟩
sublocale *Ac*: *composite-completion A* ⟨*proof*⟩

notation *P.Resid* (**infix** ‹${}^*\backslash_A{}^*$› *70*)
notation *P.Resid1x* (**infix** ‹${}^1\backslash_A{}^*$› *70*)
notation *P.Residx1* (**infix** ‹${}^*\backslash_A{}^1$› *70*)
notation *P.Con* (**infix** ‹${}^*\frown_A{}^*$› *70*)
notation *B.comp* (**infixr** ‹$\cdot_B$› *55*)
notation *B.con* (**infix** ‹$\frown_B$› *50*)

interpretation *F-ext*: *extension-to-paths A B B.comp F* ⟨*proof*⟩

definition *map*
where *map* = *Q.ext-to-quotient B F-ext.map*

sublocale *simulation Ac.resid B map*
⟨*proof*⟩

lemma *is-simulation*:
shows *simulation Ac.resid B map*
⟨*proof*⟩

lemma *is-extension*:
shows *map* ∘ *Ac.incl* = *F*
⟨*proof*⟩

lemma *is-universal*:
shows $\exists!F'$. *simulation Ac.resid B* $F' \wedge F' \circ$ *Ac.incl* = *F*
⟨*proof*⟩

end

context *composite-completion*
begin

lemma *arrows-factor-as-paths*:
assumes *arr* $\mathcal{T}$
shows $\exists$ *T*. *P.arr T* ∧ *extension-to-paths.map R resid comp incl T* = $\mathcal{T}$
⟨*proof*⟩

end

lemma *extension-to-composite-completion-comp*:
assumes *extensional-rts-with-composites B*
and *extensional-rts-with-composites C*
and *simulation A B F* **and** *simulation B C G*
shows *extension-to-composite-completion.map A C* (*G* ∘ *F*) =
G ∘ *extension-to-composite-completion.map A B F*
⟨*proof*⟩

lemma *composite-completion-of-rts*:
assumes *rts A*
shows $\exists$ (A' :: *'a list set resid*) *I*.
extensional-rts-with-composites A' ∧ *simulation A* A' *I* ∧
($\forall$ *B* (*J* :: $'a \Rightarrow 'c$). *extensional-rts-with-composites B* ∧ *simulation A B J*
$\longrightarrow$ ($\exists!J'$. *simulation* A' *B* J' ∧ J' *o I* = *J*))
⟨*proof*⟩

2.6 Constructions on RTS's

2.6.1 Products of RTS's

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

for A :: $'a$ *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 $A.cong$ (**infix** ‹$\sim_A$› *50*)

notation $B.con$ (**infix** ‹$\frown_B$› *50*)
notation $B.prfx$ (**infix** ‹$\lesssim_B$› *50*)
notation $B.cong$ (**infix** ‹$\sim_B$› *50*)

type-synonym $('c, 'd)$ *arr* $= 'c * 'd$

abbreviation (*input*) *Null* :: $('a, 'b)$ *arr*
where *Null* $\equiv$ (*A.null*, *B.null*)

definition *resid* :: $('a, 'b)$ *arr* $\Rightarrow$ $('a, 'b)$ *arr* $\Rightarrow$ $('a, 'b)$ *arr*
where *resid t u* = (*if fst t* $\frown_A$ *fst u* $\wedge$ *snd t* $\frown_B$ *snd u*
then (*fst t* $\backslash_A$ *fst u*, *snd t* $\backslash_B$ *snd u*)
else Null)

notation *resid* (**infix** ‹$\backslash$› *70*)

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

lemma *is-partial-magma*:
shows *partial-magma resid*
⟨*proof*⟩

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

sublocale *residuation resid*
⟨*proof*⟩

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

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

lemma *arr-char* [*iff*]:
shows *arr t* $\longleftrightarrow$ *A.arr* (*fst t*) $\wedge$ *B.arr* (*snd t*)
⟨*proof*⟩

lemma *ide-char* [*iff*]:

shows *ide t* $\longleftrightarrow$ *A.ide* (*fst t*) $\wedge$ *B.ide* (*snd t*)
⟨*proof*⟩

lemma *con-char* [*iff*]:
shows $t \frown u \longleftrightarrow$ *fst t* $\frown_A$ *fst u* $\wedge$ *snd t* $\frown_B$ *snd u*
⟨*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*⟩

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

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

lemma *sources-char*:
shows *sources t* = *A.sources* (*fst t*) $\times$ *B.sources* (*snd t*)
⟨*proof*⟩

lemma *targets-char*:
shows *targets t* = *A.targets* (*fst t*) $\times$ *B.targets* (*snd t*)
⟨*proof*⟩

lemma *prfx-char*:
shows $t \lesssim u \longleftrightarrow$ *fst t* $\lesssim_A$ *fst u* $\wedge$ *snd t* $\lesssim_B$ *snd u*
⟨*proof*⟩

lemma *cong-char*:
shows $t \sim u \longleftrightarrow$ *fst t* $\sim_A$ *fst u* $\wedge$ *snd t* $\sim_B$ *snd u*
⟨*proof*⟩

lemma *join-of-char*:
shows *join-of t u v* $\longleftrightarrow$ *A.join-of* (*fst t*) (*fst u*) (*fst v*) $\wedge$ *B.join-of* (*snd t*) (*snd u*) (*snd v*)
and *joinable t u* $\longleftrightarrow$ *A.joinable* (*fst t*) (*fst u*) $\wedge$ *B.joinable* (*snd t*) (*snd u*)
⟨*proof*⟩

end

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

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

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

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

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

end

Product Simulations

locale *product-simulation* =
A1: *rts A1* +
A0: *rts A0* +
B1: *rts B1* +
B0: *rts B0* +
A1xA0: *product-rts A1 A0* +
B1xB0: *product-rts B1 B0* +
F1: *simulation A1 B1 F1* +
F0: *simulation A0 B0 F0*
for *A1* :: *'a1 resid* (**infix** ‹$\backslash_{A1}$› *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*
begin

definition *map*

```
  where map = (λa. if A1xA0.arr a then (F1 (fst a), F0 (snd a))
                   else (F1 A1.null, F0 A0.null))

  lemma map-simp [simp]:
  assumes A1.arr a1 and A0.arr a0
  shows map (a1, a0) = (F1 a1, F0 a0)
    ⟨proof⟩

  sublocale simulation A1xA0.resid B1xB0.resid map
  ⟨proof⟩

  lemma is-simulation:
  shows simulation A1xA0.resid B1xB0.resid map
    ⟨proof⟩

end
```

Binary Simulations

```
locale binary-simulation =
  A1: rts A1 +
  A0: rts A0 +
  A: product-rts A1 A0 +
  B: rts B +
  simulation A.resid B F
for A1 :: 'a1 resid    (infix ‹\A1› 70)
and A0 :: 'a0 resid    (infix ‹\A0› 70)
and B :: 'b resid      (infix ‹\B› 70)
and F :: 'a1 * 'a0 ⇒ 'b
begin

  lemma fixing-ide-gives-simulation-1:
  assumes A1.ide a1
  shows simulation A0 B (λt0. F (a1, t0))
  ⟨proof⟩

  lemma fixing-ide-gives-simulation-0:
  assumes A0.ide a0
  shows simulation A1 B (λt1. F (t1, a0))
  ⟨proof⟩

end
```

2.6.2 Sub-RTS's

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

```
locale sub-rts =
```

R: *rts R*
for R :: *'a resid* (**infix** ‹$\backslash_R$› *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* ($t \backslash_R u$)
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

notation *R.con* (**infix** ‹$\frown_R$› *50*)
notation *R.prfx* (**infix** ‹$\lesssim_R$› *50*)
notation *R.cong* (**infix** ‹$\sim_R$› *50*)

definition *resid* :: *'a resid* (**infix** ‹$\backslash$› *70*)
where $t \backslash u \equiv$ *if Arr t* $\wedge$ *Arr u* $\wedge$ $t \frown_R u$ *then* $t \backslash_R u$ *else R.null*

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

lemma *is-partial-magma*:
shows *partial-magma resid*
⟨*proof*⟩

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

sublocale *residuation resid*
⟨*proof*⟩

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

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

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* = ($\lambda t.$ *if arr t then R.trg t else null*)
⟨*proof*⟩

sublocale *rts resid*
⟨*proof*⟩

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

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

lemma *sources-subset*:
shows *sources t* $\subseteq$ {*a. Arr t* $\wedge$ *a* $\in$ *R.sources t*}
⟨*proof*⟩

lemma *targets-subset*:
shows *targets t* $\subseteq$ {*b. Arr t* $\wedge$ *b* $\in$ *R.targets t*}
⟨*proof*⟩

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

lemma *cong-char*$_{SRTS}$:
shows $t \sim u \longleftrightarrow$ *Arr t* $\wedge$ *Arr u* $\wedge$ $t \sim_R u$
⟨*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*⟩

abbreviation *incl*
where *incl t* $\equiv$ *if arr t then t else null*

sublocale *Incl*: *simulation resid R incl*
⟨*proof*⟩

lemma *inclusion-is-simulation*:
shows *simulation resid R incl*
⟨*proof*⟩

lemma *incl-cancel-left*:
assumes *transformation X resid F G T* **and** *transformation X resid F′ G′ T′*
and $incl \circ T = incl \circ T'$
shows $T = T'$
⟨*proof*⟩

lemma *incl-reflects-con*:
assumes *R.con* (*incl t*) (*incl u*)
shows *con t u*
⟨*proof*⟩

lemma *corestriction-of-simulation*:
assumes *simulation X R F*
and $\bigwedge x.$ *residuation.arr X x* $\Longrightarrow$ *Arr* (*F x*)
shows *simulation X resid F* **and** $incl \circ F = F$
⟨*proof*⟩

lemma *corestriction-of-transformation*:
assumes *simulation X resid F* **and** *simulation X resid G*
and *transformation X R F G T*
and $\bigwedge x.$ *residuation.arr X x* $\Longrightarrow$ *Arr* (*T x*)
shows *transformation X resid F G T* **and** $incl \circ T = T$
⟨*proof*⟩

end

locale *source-replete-sub-rts* =
R: *rts R*
for *R* :: *′a resid* (**infix** $\backslash_R$ *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* ($t \backslash_R u$)
and *source-replete*: *Arr t* $\Longrightarrow$ *R.sources t* $\subseteq$ *Collect Arr*
begin

sublocale *sub-rts*
⟨*proof*⟩

lemma *is-sub-rts*:
shows *sub-rts R Arr*
⟨*proof*⟩

lemma *sources-char*$_{SRTS}$:
shows *sources* $t = \{a.\ Arr\ t \wedge a \in R.sources\ t\}$
⟨*proof*⟩

lemma *targets-char*$_{SRTS}$:
shows *targets* $t = \{b.\ Arr\ t \wedge b \in R.targets\ t\}$
⟨*proof*⟩

interpretation P_R: *paths-in-rts R*
⟨*proof*⟩
interpretation P: *paths-in-rts resid*
⟨*proof*⟩

lemma *path-reflection*:
shows $[\![P_R.Arr\ T;\ set\ T \subseteq Collect\ Arr]\!] \Longrightarrow P.Arr\ T$
⟨*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 *is-weakly-extensional-rts*:
shows *weakly-extensional-rts resid*
⟨*proof*⟩

lemma *src-char*:
shows $src = (\lambda t.\ if\ arr\ t\ then\ R.src\ t\ else\ null)$
⟨*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* ⇒ *bool*
begin

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

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

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

end

Here we justify the terminology "normal sub-RTS", which was introduced earlier, by showing that a normal sub-RTS really is a sub-RTS.

lemma (**in** *normal-sub-rts*) *is-sub-rts*:
shows *source-replete-sub-rts resid* ($\lambda t.\ t \in \mathfrak{N}$)
⟨*proof*⟩

end

Chapter 3

The Lambda Calculus

In this second part of the article, we apply the residuated transition system framework developed in the first part to the theory of reductions in Church's λ-calculus. The underlying idea is to exhibit λ-terms as states (identities) of an RTS, with reduction steps as non-identity transitions. We represent both states and transitions in a unified, variable-free syntax based on de Bruijn indices. A difficulty one faces in regarding the λ-calculus as an RTS is that "elementary reductions", in which just one redex is contracted, are not preserved by residuation: an elementary reduction can have zero or more residuals along another elementary reduction. However, "parallel reductions", which permit the contraction of multiple redexes existing in a term to be contracted in a single step, are preserved by residuation. For this reason, in our syntax each term represents a parallel reduction of zero or more redexes; a parallel reduction of zero redexes representing an identity. We have syntactic constructors for variables, λ-abstractions, and applications. An additional constructor represents a β-redex that has been marked for contraction. This is a slightly different approach than that taken by other authors (*e.g.* [1] or [7]), in which it is the application constructor that is marked to indicate a redex to be contracted, but it seems more natural in the present setting in which a single syntax is used to represent both terms and reductions.

Once the syntax has been defined, we define the residuation operation and prove that it satisfies the conditions for a weakly extensional RTS. In this RTS, the source of a term is obtained by "erasing" the markings on redexes, leaving an identity term. The target of a term is the contractum of the parallel reduction it represents. As the definition of residuation involves the use of substitution, a necessary prerequisite is to develop the theory of substitution using de Bruijn indices. In addition, various properties concerning the commutation of residuation and substitution have to be proved. This part of the work has benefited greatly from previous work of Huet [7], in which the theory of residuation was formalized in the proof assistant Coq. In particular, it was very helpful to have already available known-correct statements of various lemmas regarding indices, substitution, and residuation. The development of the theory culminates in the proof of Lévy's "Cube Lemma" [8], which is the key axiom in the definition of RTS.

Once reductions in the λ-calculus have been cast as transitions of an RTS, we are able to take advantage of generic results already proved for RTS's; in particular, the construction of the RTS of paths, which represent reduction sequences. Very little additional effort is required at this point to prove the Church-Rosser Theorem. Then, after proving a series of miscellaneous lemmas about reduction paths, we turn to the study of developments. A development of a term is a reduction path from that term in which the only redexes that are contracted are those that are residuals of redexes in the original term. We prove the Finite Developments Theorem: all developments are finite. The proof given here follows that given by de Vrijer [5], except that here we make the adaptations necessary for a syntax based on de Bruijn indices, rather than the classical named-variable syntax used by de Vrijer. Using the Finite Developments Theorem, we define a function that takes a term and constructs a "complete development" of that term, which is a development in which no residuals of original redexes remain to be contracted.

We then turn our attention to "standard reduction paths", which are reduction paths in which redexes are contracted in a left-to-right order, perhaps with some skips. After giving a definition of standard reduction paths, we define a function that takes a term and constructs a complete development that is also standard. Using this function as a base case, we then define a function that takes an arbitrary parallel reduction path and transforms it into a standard reduction path that is congruent to the given path. The algorithm used is roughly analogous to insertion sort. We use this function to prove strong form of the Standardization Theorem: every reduction path is congruent to a standard reduction path. As a corollary of the Standardization Theorem, we prove the Leftmost Reduction Theorem: leftmost reduction is a normalizing reduction strategy.

It should be noted that, in this article, we consider only the $\lambda\beta$-calculus. In the early stages of this work, I made an exploratory attempt to incorporate η-reduction as well, but after encountering some unanticipated difficulties I decided not to attempt that extension until the β-only case had been well-developed.

theory *LambdaCalculus*
imports *Main ResiduatedTransitionSystem*
begin

3.1 Syntax

locale *lambda-calculus*
begin

The syntax of terms has constructors *Var* for variables, *Lam* for λ-abstraction, and *App* for application. In addition, there is a constructor *Beta* which is used to represent a β-redex that has been marked for contraction. The idea is that a term *Beta t u* represents a marked version of the term *App* (*Lam t*) *u*. Finally, there is a constructor *Nil* which is used to represent the null element required for the residuation operation.

datatype (*discs-sels*) *lambda* =
Nil
| *Var nat*
| *Lam lambda*
| *App lambda lambda*

| *Beta lambda lambda*

The following notation renders *Beta t u* as a "marked" version of *App* (*Lam t*) *u*, even though the former is a single constructor, whereas the latter contains two constructors.

notation *Nil* (‹♯›)
notation *Var* (‹«-»›)
notation *Lam* (‹**λ**[-]›)
notation *App* (**infixl** ‹∘› *55*)
notation *Beta* (‹(**λ**[-] • -)› [*55*, *56*] *55*)

The following function computes the set of free variables of a term. Note that since variables are represented by numeric indices, this is a set of numbers.

fun *FV*
where *FV* ♯ = {}
  | *FV* «*i*» = {*i*}
  | *FV* **λ**[*t*] = (λ*n*. *n* − *1*) ‘ (*FV t* − {*0*})
  | *FV* (*t* ∘ *u*) = *FV t* ∪ *FV u*
  | *FV* (**λ**[*t*] • *u*) = (λ*n*. *n* − *1*) ‘ (*FV t* − {*0*}) ∪ *FV u*

3.1.1 Some Orderings for Induction

We will need to do some simultaneous inductions on pairs and triples of subterms of given terms. We prove the well-foundedness of the associated relations using the following size measure.

fun *size* :: *lambda* ⇒ *nat*
where *size* ♯ = *0*
  | *size* «-» = *1*
  | *size* **λ**[*t*] = *size t* + *1*
  | *size* (*t* ∘ *u*) = *size t* + *size u* + *1*
  | *size* (**λ**[*t*] • *u*) = (*size t* + *1*) + *size u* + *1*

lemma *wf-if-img-lt*:
fixes *r* :: (*'a* ∗ *'a*) *set* **and** *f* :: *'a* ⇒ *nat*
assumes ⋀*x y*. (*x*, *y*) ∈ *r* ⟹ *f x* < *f y*
shows *wf r*
 ⟨*proof*⟩

inductive *subterm*
where ⋀*t*. *subterm t* **λ**[*t*]
  | ⋀*t u*. *subterm t* (*t* ∘ *u*)
  | ⋀*t u*. *subterm u* (*t* ∘ *u*)
  | ⋀*t u*. *subterm t* (**λ**[*t*] • *u*)
  | ⋀*t u*. *subterm u* (**λ**[*t*] • *u*)
  | ⋀*t u v*. ⟦*subterm t u*; *subterm u v*⟧ ⟹ *subterm t v*

lemma *subterm-implies-smaller*:
shows *subterm t u* ⟹ *size t* < *size u*
 ⟨*proof*⟩

abbreviation *subterm-rel*
where *subterm-rel* ≡ {(*t*, *u*). *subterm t u*}

lemma *wf-subterm-rel*:
shows *wf subterm-rel*
⟨*proof*⟩

abbreviation *subterm-pair-rel*
where *subterm-pair-rel* ≡ {((*t1*, *t2*), *u1*, *u2*). *subterm t1 u1* ∧ *subterm t2 u2*}

lemma *wf-subterm-pair-rel*:
shows *wf subterm-pair-rel*
⟨*proof*⟩

abbreviation *subterm-triple-rel*
where *subterm-triple-rel* ≡
{((*t1*, *t2*, *t3*), *u1*, *u2*, *u3*). *subterm t1 u1* ∧ *subterm t2 u2* ∧ *subterm t3 u3*}

lemma *wf-subterm-triple-rel*:
shows *wf subterm-triple-rel*
⟨*proof*⟩

lemma *subterm-lemmas*:
shows *subterm t* **λ**[*t*]
and *subterm t* (**λ**[*t*] ∘ *u*) ∧ *subterm u* (**λ**[*t*] ∘ *u*)
and *subterm t* (*t* ∘ *u*) ∧ *subterm u* (*t* ∘ *u*)
and *subterm t* (**λ**[*t*] • *u*) ∧ *subterm u* (**λ**[*t*] • *u*)
⟨*proof*⟩

3.1.2 Arrows and Identities

Here we define some special classes of terms. An "arrow" is a term that contains no occurrences of *Nil*. An "identity" is an arrow that contains no occurrences of *Beta*. It will be important for the commutation of substitution and residuation later on that substitution not be used in a way that could create any marked redexes; for example, we don't want the substitution of *Lam* (*Var 0*) for *Var 0* in an application *App* (*Var 0*) (*Var 0*) to create a new "marked" redex. The use of the separate constructor *Beta* for marked redexes automatically avoids this.

fun *Arr*
where *Arr* ♯ = *False*
| *Arr* «-» = *True*
| *Arr* **λ**[*t*] = *Arr t*
| *Arr* (*t* ∘ *u*) = (*Arr t* ∧ *Arr u*)
| *Arr* (**λ**[*t*] • *u*) = (*Arr t* ∧ *Arr u*)

lemma *Arr-not-Nil*:
assumes *Arr t*

shows $t \neq \sharp$
  $\langle proof \rangle$

fun *Ide*
where $Ide\ \sharp = False$
  $\mid Ide$ «-» $= True$
  $\mid Ide\ \boldsymbol{\lambda}[t] = Ide\ t$
  $\mid Ide\ (t \circ u) = (Ide\ t \wedge Ide\ u)$
  $\mid Ide\ (\boldsymbol{\lambda}[t] \bullet u) = False$

lemma *Ide-implies-Arr*:
shows $Ide\ t \Longrightarrow Arr\ t$
  $\langle proof \rangle$

lemma *ArrE* [*elim*]:
assumes $Arr\ t$
and $\bigwedge i.\ t =$ «*i*» $\Longrightarrow T$
and $\bigwedge u.\ t = \boldsymbol{\lambda}[u] \Longrightarrow T$
and $\bigwedge u\ v.\ t = u \circ v \Longrightarrow T$
and $\bigwedge u\ v.\ t = \boldsymbol{\lambda}[u] \bullet v \Longrightarrow T$
shows T
  $\langle proof \rangle$

3.1.3 Raising Indices

For substitution, we need to be able to raise the indices of all free variables in a subterm by a specified amount. To do this recursively, we need to keep track of the depth of nesting of λ's and only raise the indices of variables that are already greater than or equal to that depth, as these are the variables that are free in the current context. This leads to defining a function *Raise* that has two arguments: the depth threshold d and the increment n to be added to indices above that threshold.

fun *Raise*
where *Raise* - - $\sharp = \sharp$
  $\mid Raise\ d\ n$ «*i*» $= (if\ i \geq d\ then$ «*i+n*» *else* «*i*»$)$
  $\mid Raise\ d\ n\ \boldsymbol{\lambda}[t] = \boldsymbol{\lambda}[Raise\ (Suc\ d)\ n\ t]$
  $\mid Raise\ d\ n\ (t \circ u) = Raise\ d\ n\ t \circ Raise\ d\ n\ u$
  $\mid Raise\ d\ n\ (\boldsymbol{\lambda}[t] \bullet u) = \boldsymbol{\lambda}[Raise\ (Suc\ d)\ n\ t] \bullet Raise\ d\ n\ u$

Ultimately, the definition of substitution will only directly involve the function that raises all indices of variables that are free in the outermost context; in a term, so we introduce an abbreviation for this special case.

abbreviation *raise*
where *raise* == *Raise 0*

lemma *size-Raise*:
shows $\bigwedge d.\ size\ (Raise\ d\ n\ t) = size\ t$
  $\langle proof \rangle$

lemma *Raise-not-Nil*:
assumes $t \neq \sharp$
shows $Raise\ d\ n\ t \neq \sharp$
⟨*proof*⟩

lemma *FV-Raise*:
shows $FV\ (Raise\ d\ n\ t) = (\lambda x.\ if\ x \geq d\ then\ x + n\ else\ x)\ `\ FV\ t$
⟨*proof*⟩

lemma *Arr-Raise*:
shows $Arr\ t \longleftrightarrow Arr\ (Raise\ d\ n\ t)$
⟨*proof*⟩

lemma *Ide-Raise*:
shows $Ide\ t \longleftrightarrow Ide\ (Raise\ d\ n\ t)$
⟨*proof*⟩

lemma *Raise-0*:
shows $Raise\ d\ 0\ t = t$
⟨*proof*⟩

lemma *Raise-Suc*:
shows $Raise\ d\ (Suc\ n)\ t = Raise\ d\ 1\ (Raise\ d\ n\ t)$
⟨*proof*⟩

lemma *Raise-Var*:
shows $Raise\ d\ n\ «i» = «if\ i < d\ then\ i\ else\ i + n»$
⟨*proof*⟩

The following development of the properties of raising indices, substitution, and residuation has benefited greatly from the previous work by Huet [7]. In particular, it was very helpful to have correct statements of various lemmas available, rather than having to reconstruct them.

lemma *Raise-plus*:
shows $Raise\ d\ (m + n)\ t = Raise\ (d + m)\ n\ (Raise\ d\ m\ t)$
⟨*proof*⟩

lemma *Raise-plus'*:
shows $[\![d' \leq d + n;\ d \leq d']\!] \Longrightarrow Raise\ d\ (m + n)\ t = Raise\ d'\ m\ (Raise\ d\ n\ t)$
⟨*proof*⟩

lemma *Raise-Raise*:
shows $i \leq n \Longrightarrow Raise\ i\ p\ (Raise\ n\ k\ t) = Raise\ (p + n)\ k\ (Raise\ i\ p\ t)$
⟨*proof*⟩

lemma *raise-plus*:
shows $d \leq n \Longrightarrow raise\ (m + n)\ t = Raise\ d\ m\ (raise\ n\ t)$
⟨*proof*⟩

lemma *raise-Raise*:
shows $raise\ p\ (Raise\ n\ k\ t) = Raise\ (p + n)\ k\ (raise\ p\ t)$
$\langle proof \rangle$

lemma *Raise-inj*:
shows $Raise\ d\ n\ t = Raise\ d\ n\ u \Longrightarrow t = u$
$\langle proof \rangle$

3.1.4 Substitution

Following [7], we now define a generalized substitution operation with adjustment of indices. The ultimate goal is to define the result of contraction of a marked redex *Beta t u* to be *subst u t*. However, to be able to give a proper recursive definition of *subst*, we need to introduce a parameter *n* to keep track of the depth of nesting of *Lam*'s as we descend into the the term *t*. So, instead of *subst u t* simply substituting *u* for occurrences of *Var 0*, *Subst n u t* will be substituting for occurrences of *Var n*, and the term *u* will have the indices of its free variables raised by *n* before replacing *Var n*. In addition, any variables in *t* that have indices greater than *n* will have these indices lowered by one, to account for the outermost *Lam* that is being removed by the contraction. We can then define *subst u t* to be *Subst 0 u t*.

fun *Subst*
where $Subst\ \text{-}\ \text{-}\ \sharp = \sharp$
$\quad |\ Subst\ n\ v\ «i» = (if\ n < i\ then\ «i-1»\ else\ if\ n = i\ then\ raise\ n\ v\ else\ «i»)$
$\quad |\ Subst\ n\ v\ \boldsymbol{\lambda}[t] = \boldsymbol{\lambda}[Subst\ (Suc\ n)\ v\ t]$
$\quad |\ Subst\ n\ v\ (t \circ u) = Subst\ n\ v\ t \circ Subst\ n\ v\ u$
$\quad |\ Subst\ n\ v\ (\boldsymbol{\lambda}[t] \bullet u) = \boldsymbol{\lambda}[Subst\ (Suc\ n)\ v\ t] \bullet Subst\ n\ v\ u$

abbreviation *subst*
where $subst \equiv Subst\ 0$

lemma *Subst-Nil*:
shows $Subst\ n\ v\ \sharp = \sharp$
$\langle proof \rangle$

lemma *Subst-not-Nil*:
assumes $v \neq \sharp$ **and** $t \neq \sharp$
shows $t \neq \sharp \Longrightarrow Subst\ n\ v\ t \neq \sharp$
$\langle proof \rangle$

The following expression summarizes how the set of free variables of a term *Subst d u t*, obtained by substituting *u* into *t* at depth *d*, relates to the sets of free variables of *t* and *u*. This expression is not used in the subsequent formal development, but it has been left here as an aid to understanding.

abbreviation *FVS*
where $FVS\ d\ v\ t \equiv (FV\ t \cap \{x.\ x < d\}) \cup$
$\qquad (\lambda x.\ x - 1)\ `\ \{x.\ x > d \wedge x \in FV\ t\} \cup$
$\qquad (\lambda x.\ x + d)\ `\ \{x.\ d \in FV\ t \wedge x \in FV\ v\}$

lemma *FV-Subst*:
shows $FV\ (Subst\ d\ v\ t) = FVS\ d\ v\ t$
⟨*proof*⟩

lemma *Arr-Subst*:
assumes $Arr\ v$
shows $Arr\ t \Longrightarrow Arr\ (Subst\ n\ v\ t)$
⟨*proof*⟩

lemma *vacuous-Subst*:
shows $[\![Arr\ v;\ i \notin FV\ t]\!] \Longrightarrow Raise\ i\ 1\ (Subst\ i\ v\ t) = t$
⟨*proof*⟩

lemma *Ide-Subst-iff*:
shows $Ide\ (Subst\ n\ v\ t) \longleftrightarrow Ide\ t \land (n \in FV\ t \longrightarrow Ide\ v)$
⟨*proof*⟩

lemma *Ide-Subst*:
shows $[\![Ide\ t;\ Ide\ v]\!] \Longrightarrow Ide\ (Subst\ n\ v\ t)$
⟨*proof*⟩

lemma *Raise-Subst*:
shows $Raise\ (p + n)\ k\ (Subst\ p\ v\ t) = Subst\ p\ (Raise\ n\ k\ v)\ (Raise\ (Suc\ (p + n))\ k\ t)$
⟨*proof*⟩

lemma *Raise-Subst'*:
assumes $t \neq \sharp$
shows $[\![v \neq \sharp;\ k \leq n]\!] \Longrightarrow Raise\ k\ p\ (Subst\ n\ v\ t) = Subst\ (p + n)\ v\ (Raise\ k\ p\ t)$
⟨*proof*⟩

lemma *Raise-subst*:
shows $Raise\ n\ k\ (subst\ v\ t) = subst\ (Raise\ n\ k\ v)\ (Raise\ (Suc\ n)\ k\ t)$
⟨*proof*⟩

lemma *raise-Subst*:
assumes $t \neq \sharp$
shows $v \neq \sharp \Longrightarrow raise\ p\ (Subst\ n\ v\ t) = Subst\ (p + n)\ v\ (raise\ p\ t)$
⟨*proof*⟩

lemma *Subst-Raise*:
shows $[\![v \neq \sharp;\ d \leq m;\ m \leq n + d]\!] \Longrightarrow Subst\ m\ v\ (Raise\ d\ (Suc\ n)\ t) = Raise\ d\ n\ t$
⟨*proof*⟩

lemma *Subst-raise*:
shows $[\![v \neq \sharp;\ m \leq n]\!] \Longrightarrow Subst\ m\ v\ (raise\ (Suc\ n)\ t) = raise\ n\ t$
⟨*proof*⟩

lemma *Subst-Subst*:

shows $[\![v \neq \sharp;\ w \neq \sharp]\!] \Longrightarrow$
$\quad Subst\ (m + n)\ w\ (Subst\ m\ v\ t) = Subst\ m\ (Subst\ n\ w\ v)\ (Subst\ (Suc\ (m + n))\ w\ t)$
$\langle proof \rangle$

The Substitution Lemma, as given by Huet [7].

lemma *substitution-lemma*:
shows $[\![v \neq \sharp;\ w \neq \sharp]\!] \Longrightarrow Subst\ n\ v\ (subst\ w\ t) = subst\ (Subst\ n\ v\ w)\ (Subst\ (Suc\ n)\ v\ t)$
$\langle proof \rangle$

3.2 Lambda-Calculus as an RTS

3.2.1 Residuation

We now define residuation on terms. Residuation is an operation which, when defined for terms t and u, produces terms $t \setminus u$ and $u \setminus t$ that represent, respectively, what remains of the reductions of t after performing the reductions in u, and what remains of the reductions of u after performing the reductions in t.

The definition ensures that, if residuation is defined for two terms, then those terms in must be arrows that are *coinitial* (*i.e.* they are the same after erasing marks on redexes). The residual $t \setminus u$ then has marked redexes at positions corresponding to redexes that were originally marked in t and that were not contracted by any of the reductions of u.

This definition has also benefited from the presentation in [7].

fun *resid* (**infix** ‹\› *70*)
where $«i» \setminus «i'» = (if\ i = i'\ then\ «i»\ else\ \sharp)$
$\quad |\ \boldsymbol{\lambda}[t] \setminus \boldsymbol{\lambda}[t'] = (if\ t \setminus t' = \sharp\ then\ \sharp\ else\ \boldsymbol{\lambda}[t \setminus t'])$
$\quad |\ (t \circ u) \setminus (t' \circ u') = (if\ t \setminus t' = \sharp \vee u \setminus u' = \sharp\ then\ \sharp\ else\ (t \setminus t') \circ (u \setminus u'))$
$\quad |\ (\boldsymbol{\lambda}[t] \bullet u) \setminus (\boldsymbol{\lambda}[t'] \bullet u') = (if\ t \setminus t' = \sharp \vee u \setminus u' = \sharp\ then\ \sharp\ else\ subst\ (u \setminus u')\ (t \setminus t'))$
$\quad |\ (\boldsymbol{\lambda}[t] \circ u) \setminus (\boldsymbol{\lambda}[t'] \bullet u') = (if\ t \setminus t' = \sharp \vee u \setminus u' = \sharp\ then\ \sharp\ else\ subst\ (u \setminus u')\ (t \setminus t'))$
$\quad |\ (\boldsymbol{\lambda}[t] \bullet u) \setminus (\boldsymbol{\lambda}[t'] \circ u') = (if\ t \setminus t' = \sharp \vee u \setminus u' = \sharp\ then\ \sharp\ else\ \boldsymbol{\lambda}[t \setminus t'] \bullet (u \setminus u'))$
$\quad |\ resid\ \text{-}\ \ \text{-} = \sharp$

Terms t and u are *consistent* if residuation is defined for them.

abbreviation *Con* (**infix** ‹⌢› *50*)
where $Con\ t\ u \equiv resid\ t\ u \neq \sharp$

lemma *ConE* [*elim*]:
assumes $t \frown t'$
and $\bigwedge i.\ [\![t = «i»;\ t' = «i»;\ resid\ t\ t' = «i»]\!] \Longrightarrow T$
and $\bigwedge u\ u'.\ [\![t = \boldsymbol{\lambda}[u];\ t' = \boldsymbol{\lambda}[u'];\ u \frown u';\ t \setminus t' = \boldsymbol{\lambda}[u \setminus u']]\!] \Longrightarrow T$
and $\bigwedge u\ v\ u'\ v'.\ [\![t = u \circ v;\ t' = u' \circ v';\ u \frown u';\ v \frown v';$
$\qquad t \setminus t' = (u \setminus u') \circ (v \setminus v')]\!] \Longrightarrow T$
and $\bigwedge u\ v\ u'\ v'.\ [\![t = \boldsymbol{\lambda}[u] \bullet v;\ t' = \boldsymbol{\lambda}[u'] \bullet v';\ u \frown u';\ v \frown v';$
$\qquad t \setminus t' = subst\ (v \setminus v')\ (u \setminus u')]\!] \Longrightarrow T$
and $\bigwedge u\ v\ u'\ v'.\ [\![t = \boldsymbol{\lambda}[u] \circ v;\ t' = Beta\ u'\ v';\ u \frown u';\ v \frown v';$
$\qquad t \setminus t' = subst\ (v \setminus v')\ (u \setminus u')]\!] \Longrightarrow T$
and $\bigwedge u\ v\ u'\ v'.\ [\![t = \boldsymbol{\lambda}[u] \bullet v;\ t' = \boldsymbol{\lambda}[u'] \circ v';\ u \frown u';\ v \frown v';$
$\qquad t \setminus t' = \boldsymbol{\lambda}[u \setminus u'] \bullet (v \setminus v')]\!] \Longrightarrow T$

shows T
⟨*proof*⟩

A term can only be consistent with another if both terms are "arrows".

lemma *Con-implies-Arr1*:
shows $t \frown u \Longrightarrow Arr\ t$
⟨*proof*⟩

lemma *Con-implies-Arr2*:
shows $t \frown u \Longrightarrow Arr\ u$
⟨*proof*⟩

lemma *ConD*:
shows $t \circ u \frown t' \circ u' \Longrightarrow t \frown t' \wedge u \frown u'$
and $\boldsymbol{\lambda}[v] \bullet u \frown \boldsymbol{\lambda}[v'] \bullet u' \Longrightarrow \boldsymbol{\lambda}[v] \frown \boldsymbol{\lambda}[v'] \wedge u \frown u'$
and $\boldsymbol{\lambda}[v] \bullet u \frown t' \circ u' \Longrightarrow \boldsymbol{\lambda}[v] \frown t' \wedge u \frown u'$
and $t \circ u \frown \boldsymbol{\lambda}[v'] \bullet u' \Longrightarrow t \frown \boldsymbol{\lambda}[v'] \wedge u \frown u'$
⟨*proof*⟩

Residuation on consistent terms preserves arrows.

lemma *Arr-resid*:
shows $t \frown u \Longrightarrow Arr\ (t \setminus u)$
⟨*proof*⟩

3.2.2 Source and Target

Here we give syntactic versions of the *source* and *target* of a term. These will later be shown to agree (on arrows) with the versions derived from the residuation. The underlying idea here is that a term stands for a reduction sequence in which all marked redexes (corresponding to instances of the constructor *Beta*) are contracted in a bottom-up fashion. A term without any marked redexes stands for an empty reduction sequence; such terms will be shown to be the identities derived from the residuation. The source of term is the identity obtained by erasing all markings; that is, by replacing all subterms of the form *Beta t u* by *App* (*Lam t*) *u*. The target of a term is the identity that is the result of contracting all the marked redexes.

fun *Src*
where $Src\ \sharp = \sharp$
  | Src «i» = «i»
  | $Src\ \boldsymbol{\lambda}[t] = \boldsymbol{\lambda}[Src\ t]$
  | $Src\ (t \circ u) = Src\ t \circ Src\ u$
  | $Src\ (\boldsymbol{\lambda}[t] \bullet u) = \boldsymbol{\lambda}[Src\ t] \circ Src\ u$

fun *Trg*
where Trg «i» = «i»
  | $Trg\ \boldsymbol{\lambda}[t] = \boldsymbol{\lambda}[Trg\ t]$
  | $Trg\ (t \circ u) = Trg\ t \circ Trg\ u$
  | $Trg\ (\boldsymbol{\lambda}[t] \bullet u) = subst\ (Trg\ u)\ (Trg\ t)$
  | Trg - $= \sharp$

lemma *Ide-Src*:
shows $Arr\ t \Longrightarrow Ide\ (Src\ t)$
$\langle proof \rangle$

lemma *Ide-iff-Src-self*:
assumes $Arr\ t$
shows $Ide\ t \longleftrightarrow Src\ t = t$
$\langle proof \rangle$

lemma *Arr-Src* [*simp*]:
assumes $Arr\ t$
shows $Arr\ (Src\ t)$
$\langle proof \rangle$

lemma *Con-Src*:
shows $[\![size\ t + size\ u \leq n;\ t \frown u]\!] \Longrightarrow Src\ t \frown Src\ u$
$\langle proof \rangle$

lemma *Src-eq-iff*:
shows $Src\ «i» = Src\ «i'» \longleftrightarrow i = i'$
and $Src\ (t \circ u) = Src\ (t' \circ u') \longleftrightarrow Src\ t = Src\ t' \wedge Src\ u = Src\ u'$
and $Src\ (\boldsymbol{\lambda}[t] \bullet u) = Src\ (\boldsymbol{\lambda}[t'] \bullet u') \longleftrightarrow Src\ t = Src\ t' \wedge Src\ u = Src\ u'$
and $Src\ (\boldsymbol{\lambda}[t] \circ u) = Src\ (\boldsymbol{\lambda}[t'] \bullet u') \longleftrightarrow Src\ t = Src\ t' \wedge Src\ u = Src\ u'$
$\langle proof \rangle$

lemma *Src-Raise*:
shows $Src\ (Raise\ d\ n\ t) = Raise\ d\ n\ (Src\ t)$
$\langle proof \rangle$

lemma *Src-Subst* [*simp*]:
shows $[\![Arr\ t;\ Arr\ u]\!] \Longrightarrow Src\ (Subst\ d\ t\ u) = Subst\ d\ (Src\ t)\ (Src\ u)$
$\langle proof \rangle$

lemma *Ide-Trg*:
shows $Arr\ t \Longrightarrow Ide\ (Trg\ t)$
$\langle proof \rangle$

lemma *Ide-iff-Trg-self*:
shows $Arr\ t \Longrightarrow Ide\ t \longleftrightarrow Trg\ t = t$
$\langle proof \rangle$

lemma *Arr-Trg* [*simp*]:
assumes $Arr\ X$
shows $Arr\ (Trg\ X)$
$\langle proof \rangle$

lemma *Src-Src* [*simp*]:
assumes $Arr\ t$

shows $Src\ (Src\ t) = Src\ t$
$\langle proof \rangle$

lemma *Trg-Src* [*simp*]:
assumes $Arr\ t$
shows $Trg\ (Src\ t) = Src\ t$
$\langle proof \rangle$

lemma *Trg-Trg* [*simp*]:
assumes $Arr\ t$
shows $Trg\ (Trg\ t) = Trg\ t$
$\langle proof \rangle$

lemma *Src-Trg* [*simp*]:
assumes $Arr\ t$
shows $Src\ (Trg\ t) = Trg\ t$
$\langle proof \rangle$

Two terms are syntactically *coinitial* if they are arrows with the same source; that is, they represent two reductions from the same starting term.

abbreviation *Coinitial*
where $Coinitial\ t\ u \equiv Arr\ t \wedge Arr\ u \wedge Src\ t = Src\ u$

We now show that terms are consistent if and only if they are coinitial.

lemma *Coinitial-cases*:
assumes $Arr\ t$ **and** $Arr\ t'$ **and** $Src\ t = Src\ t'$
shows $(t = \sharp \wedge t' = \sharp) \vee$
$(\exists x.\ t = «x» \wedge t' = «x») \vee$
$(\exists u\ u'.\ t = \boldsymbol{\lambda}[u] \wedge t' = \boldsymbol{\lambda}[u']) \vee$
$(\exists u\ v\ u'\ v'.\ t = u \circ v \wedge t' = u' \circ v') \vee$
$(\exists u\ v\ u'\ v'.\ t = \boldsymbol{\lambda}[u] \bullet v \wedge t' = \boldsymbol{\lambda}[u'] \bullet v') \vee$
$(\exists u\ v\ u'\ v'.\ t = \boldsymbol{\lambda}[u] \circ v \wedge t' = \boldsymbol{\lambda}[u'] \bullet v') \vee$
$(\exists u\ v\ u'\ v'.\ t = \boldsymbol{\lambda}[u] \bullet v \wedge t' = \boldsymbol{\lambda}[u'] \circ v')$
$\langle proof \rangle$

lemma *Con-implies-Coinitial-ind*:
shows $[\![size\ t + size\ u \leq n;\ t \frown u]\!] \Longrightarrow Coinitial\ t\ u$
$\langle proof \rangle$

lemma *Coinitial-implies-Con-ind*:
shows $[\![size\ (Src\ t) \leq n;\ Coinitial\ t\ u]\!] \Longrightarrow t \frown u$
$\langle proof \rangle$

lemma *Coinitial-iff-Con*:
shows $Coinitial\ t\ u \longleftrightarrow t \frown u$
$\langle proof \rangle$

lemma *Coinitial-Raise-Raise*:
shows $Coinitial\ t\ u \Longrightarrow Coinitial\ (Raise\ d\ n\ t)\ (Raise\ d\ n\ u)$

⟨*proof*⟩

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

lemma *ConI* [*intro*, *simp*]:
assumes *Arr t* **and** *Arr u* **and** *Src t* = *Src u*
shows *Con t u*
⟨*proof*⟩

lemma *Con-Arr-Src* [*simp*]:
assumes *Arr t*
shows $t \frown$ *Src t* **and** *Src t* $\frown t$
⟨*proof*⟩

lemma *resid-Arr-self*:
shows *Arr t* $\Longrightarrow t \setminus t =$ *Trg t*
⟨*proof*⟩

The following result is not used in the formal development that follows, but it requires some proof and might eventually be useful.

lemma *finite-branching*:
shows *Ide a* $\Longrightarrow$ *finite* {*t*. *Arr t* $\wedge$ *Src t* = *a*}
⟨*proof*⟩

3.2.3 Residuation and Substitution

We now develop a series of lemmas that involve the interaction of residuation and substitution.

lemma *Raise-resid*:
shows $t \frown u \Longrightarrow$ *Raise k n* $(t \setminus u) =$ *Raise k n t* $\setminus$ *Raise k n u*
⟨*proof*⟩

lemma *Con-Raise*:
shows $t \frown u \Longrightarrow$ *Raise d n t* $\frown$ *Raise d n u*
⟨*proof*⟩

The following is Huet's Commutation Theorem [7]: "substitution commutes with residuation".

lemma *resid-Subst*:
assumes $t \frown t'$ **and** $u \frown u'$
shows *Subst n t u* $\setminus$ *Subst n* t' u' = *Subst n* $(t \setminus t')$ $(u \setminus u')$
⟨*proof*⟩

lemma *Trg-Subst* [*simp*]:
shows ⟦*Arr t*; *Arr u*⟧ $\Longrightarrow$ *Trg* (*Subst d t u*) = *Subst d* (*Trg t*) (*Trg u*)
⟨*proof*⟩

lemma *Src-resid*:
shows $t \frown u \Longrightarrow Src\ (t \setminus u) = Trg\ u$
⟨*proof*⟩

lemma *Coinitial-resid-resid*:
assumes $t \frown v$ **and** $u \frown v$
shows $Coinitial\ (t \setminus v)\ (u \setminus v)$
⟨*proof*⟩

lemma *Con-implies-is-Lam-iff-is-Lam*:
assumes $t \frown u$
shows $is\text{-}Lam\ t \longleftrightarrow is\text{-}Lam\ u$
⟨*proof*⟩

lemma *Con-implies-Coinitial3*:
assumes $t \setminus v \frown u \setminus v$
shows $Coinitial\ v\ u$ **and** $Coinitial\ v\ t$ **and** $Coinitial\ u\ t$
⟨*proof*⟩

We can now prove Lévy's "Cube Lemma" [8], which is the key axiom for a residuated transition system.

lemma *Cube*:
shows $v \setminus t \frown u \setminus t \Longrightarrow (v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$
⟨*proof*⟩

3.2.4 Residuation Determines an RTS

We are now in a position to verify that the residuation operation that we have defined satisfies the axioms for a residuated transition system, and that various notions which we have defined syntactically above (*e.g.* arrow, source, target) agree with the versions derived abstractly from residuation.

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

lemma *null-char* [*simp*]:
shows $null = \sharp$
⟨*proof*⟩

sublocale *residuation resid*
⟨*proof*⟩

notation *resid* (**infix** ‹\› *70*)

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

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

lemma *ide-char* [*iff*]:
shows *ide* $t \longleftrightarrow$ *Ide* t
  ⟨*proof*⟩

lemma *resid-Arr-Ide*:
shows $[\![Ide\ a;\ Coinitial\ t\ a]\!] \Longrightarrow t \setminus a = t$
  ⟨*proof*⟩

lemma *resid-Ide-Arr*:
shows $[\![Ide\ a;\ Coinitial\ a\ t]\!] \Longrightarrow Ide\ (a \setminus t)$
  ⟨*proof*⟩

lemma *resid-Arr-Src* [*simp*]:
assumes *Arr* t
shows $t \setminus Src\ t = t$
  ⟨*proof*⟩

lemma *resid-Src-Arr* [*simp*]:
assumes *Arr* t
shows $Src\ t \setminus t = Trg\ t$
  ⟨*proof*⟩

sublocale *rts resid*
⟨*proof*⟩

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

lemma *sources-char*$_\Lambda$:
shows $sources\ t = (if\ Arr\ t\ then\ \{Src\ t\}\ else\ \{\})$
⟨*proof*⟩

lemma *sources-simp* [*simp*]:
assumes *Arr* t
shows $sources\ t = \{Src\ t\}$
  ⟨*proof*⟩

lemma *sources-simps* [*simp*]:
shows $sources\ \sharp = \{\}$
and $sources\ «x» = \{«x»\}$
and $arr\ t \Longrightarrow sources\ \boldsymbol{\lambda}[t] = \{\boldsymbol{\lambda}[Src\ t]\}$
and $[\![arr\ t;\ arr\ u]\!] \Longrightarrow sources\ (t \circ u) = \{Src\ t \circ Src\ u\}$
and $[\![arr\ t;\ arr\ u]\!] \Longrightarrow sources\ (\boldsymbol{\lambda}[t] \bullet u) = \{\boldsymbol{\lambda}[Src\ t] \circ Src\ u\}$

$\langle proof \rangle$

lemma *targets-char*$_\Lambda$:
shows *targets t* = (*if Arr t then* {*Trg t*} *else* {})
$\langle proof \rangle$

lemma *targets-simp* [*simp*]:
assumes *Arr t*
shows *targets t* = {*Trg t*}
$\langle proof \rangle$

lemma *targets-simps* [*simp*]:
shows *targets* ♯ = {}
and *targets* «*x*» = {«*x*»}
and *arr t* $\Longrightarrow$ *targets* **λ**[*t*] = {**λ**[*Trg t*]}
and ⟦*arr t*; *arr u*⟧ $\Longrightarrow$ *targets* (*t* ∘ *u*) = {*Trg t* ∘ *Trg u*}
and ⟦*arr t*; *arr u*⟧ $\Longrightarrow$ *targets* (**λ**[*t*] • *u*) = {*subst* (*Trg u*) (*Trg t*)}
$\langle proof \rangle$

lemma *seq-char*:
shows *seq t u* $\longleftrightarrow$ *Arr t* ∧ *Arr u* ∧ *Trg t* = *Src u*
$\langle proof \rangle$

lemma *seqI*$_\Lambda$ [*intro*, *simp*]:
assumes *Arr t* **and** *Arr u* **and** *Trg t* = *Src u*
shows *seq t u*
$\langle proof \rangle$

lemma *seqE*$_\Lambda$ [*elim*]:
assumes *seq t u*
and ⟦*Arr t*; *Arr u*; *Trg t* = *Src u*⟧ $\Longrightarrow$ *T*
shows *T*
$\langle proof \rangle$

The following classifies the ways that transitions can be sequential. It is useful for later proofs by case analysis.

lemma *seq-cases*:
assumes *seq t u*
shows (*is-Var t* ∧ *is-Var u*) ∨
(*is-Lam t* ∧ *is-Lam u*) ∨
(*is-App t* ∧ *is-App u*) ∨
(*is-App t* ∧ *is-Beta u* ∧ *is-Lam* (*un-App1 t*)) ∨
(*is-App t* ∧ *is-Beta u* ∧ *is-Beta* (*un-App1 t*)) ∨
is-Beta t
$\langle proof \rangle$

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

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

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

lemma *coinitial-char* [*iff*]:
shows *coinitial t u* $\longleftrightarrow$ *Coinitial t u*
⟨*proof*⟩

lemma *sources-Raise*:
assumes *Arr t*
shows *sources* (*Raise d n t*) = {*Raise d n* (*Src t*)}
⟨*proof*⟩

lemma *targets-Raise*:
assumes *Arr t*
shows *targets* (*Raise d n t*) = {*Raise d n* (*Trg t*)}
⟨*proof*⟩

lemma *sources-subst* [*simp*]:
assumes *Arr t* **and** *Arr u*
shows *sources* (*subst t u*) = {*subst* (*Src t*) (*Src u*)}
⟨*proof*⟩

lemma *targets-subst* [*simp*]:
assumes *Arr t* **and** *Arr u*
shows *targets* (*subst t u*) = {*subst* (*Trg t*) (*Trg u*)}
⟨*proof*⟩

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

lemma *prfx-char* [*iff*]:
shows *t* $\lesssim$ *u* $\longleftrightarrow$ *Ide* (*t* \ *u*)
⟨*proof*⟩

lemma *prfx-Var-iff*:
shows *u* $\lesssim$ *«i»* $\longleftrightarrow$ *u* = *«i»*
⟨*proof*⟩

lemma *prfx-Lam-iff*:
shows *u* $\lesssim$ *Lam t* $\longleftrightarrow$ *is-Lam u* $\wedge$ *un-Lam u* $\lesssim$ *t*
⟨*proof*⟩

lemma *prfx-App-iff*:
shows *u* $\lesssim$ *t1* $\circ$ *t2* $\longleftrightarrow$ *is-App u* $\wedge$ *un-App1 u* $\lesssim$ *t1* $\wedge$ *un-App2 u* $\lesssim$ *t2*

⟨*proof*⟩

lemma *prfx-Beta-iff*:
shows $u \lesssim \boldsymbol{\lambda}[t1] \bullet t2 \longleftrightarrow$
$(is\text{-}App\ u \wedge un\text{-}App1\ u \lesssim \boldsymbol{\lambda}[t1] \wedge un\text{-}App2\ u \frown t2 \wedge$
$(0 \in FV\ (un\text{-}Lam\ (un\text{-}App1\ u) \setminus t1) \longrightarrow un\text{-}App2\ u \lesssim t2)) \vee$
$(is\text{-}Beta\ u \wedge un\text{-}Beta1\ u \lesssim t1 \wedge un\text{-}Beta2\ u \frown t2 \wedge$
$(0 \in FV\ (un\text{-}Beta1\ u \setminus t1) \longrightarrow un\text{-}Beta2\ u \lesssim t2))$
⟨*proof*⟩

lemma *cong-Ide-are-eq*:
assumes $t \sim u$ **and** *Ide t* **and** *Ide u*
shows $t = u$
⟨*proof*⟩

lemma *eq-Ide-are-cong*:
assumes $t = u$ **and** *Ide t*
shows $t \sim u$
⟨*proof*⟩

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

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

lemma *src-char* [*simp*]:
shows $src\ t = (if\ Arr\ t\ then\ Src\ t\ else\ \sharp)$
⟨*proof*⟩

lemma *trg-char* [*simp*]:
shows $trg\ t = (if\ Arr\ t\ then\ Trg\ t\ else\ \sharp)$
⟨*proof*⟩

We "almost" have an extensional RTS. The case that fails is $\boldsymbol{\lambda}[t1] \bullet t2 \sim u \Longrightarrow \boldsymbol{\lambda}[t1] \bullet t2 = u$. This is because *t1* might ignore its argument, so that $subst\ t2\ t1 = subst\ t2'\ t1$, with both sides being identities, even if $t2 \neq t2'$.

The following gives a concrete example of such a situation.

abbreviation *non-extensional-ex1*
where $non\text{-}extensional\text{-}ex1 \equiv \boldsymbol{\lambda}[\boldsymbol{\lambda}[«0»] \circ \boldsymbol{\lambda}[«0»]] \bullet (\boldsymbol{\lambda}[«0»] \bullet \boldsymbol{\lambda}[«0»])$

abbreviation *non-extensional-ex2*
where $non\text{-}extensional\text{-}ex2 \equiv \boldsymbol{\lambda}[\boldsymbol{\lambda}[«0»] \circ \boldsymbol{\lambda}[«0»]] \bullet (\boldsymbol{\lambda}[«0»] \circ \boldsymbol{\lambda}[«0»])$

lemma *non-extensional*:
shows $\boldsymbol{\lambda}[«1»] \bullet non\text{-}extensional\text{-}ex1 \sim \boldsymbol{\lambda}[«1»] \bullet non\text{-}extensional\text{-}ex2$
and $\boldsymbol{\lambda}[«1»] \bullet non\text{-}extensional\text{-}ex1 \neq \boldsymbol{\lambda}[«1»] \bullet non\text{-}extensional\text{-}ex2$
⟨*proof*⟩

The following gives an example of two terms that are both coinitial and coterminal, but which are not congruent.

abbreviation *cong-nontrivial-ex1*
where *cong-nontrivial-ex1* $\equiv$
$\boldsymbol{\lambda}[\text{«}0\text{»} \circ \text{«}0\text{»}] \circ \boldsymbol{\lambda}[\text{«}0\text{»} \circ \text{«}0\text{»}] \circ (\boldsymbol{\lambda}[\text{«}0\text{»} \circ \text{«}0\text{»}] \bullet \boldsymbol{\lambda}[\text{«}0\text{»} \circ \text{«}0\text{»}])$

abbreviation *cong-nontrivial-ex2*
where *cong-nontrivial-ex2* $\equiv$
$\boldsymbol{\lambda}[\text{«}0\text{»} \circ \text{«}0\text{»}] \bullet \boldsymbol{\lambda}[\text{«}0\text{»} \circ \text{«}0\text{»}] \circ (\boldsymbol{\lambda}[\text{«}0\text{»} \circ \text{«}0\text{»}] \circ \boldsymbol{\lambda}[\text{«}0\text{»} \circ \text{«}0\text{»}])$

lemma *cong-nontrivial*:
shows *coinitial cong-nontrivial-ex1 cong-nontrivial-ex2*
and *coterminal cong-nontrivial-ex1 cong-nontrivial-ex2*
and $\neg$ *cong cong-nontrivial-ex1 cong-nontrivial-ex2*
⟨*proof*⟩

Every two coinitial transitions have a join, obtained structurally by unioning the sets of marked redexes.

fun *Join* (**infix** ‹⊔› *52*)
where $\text{«}x\text{»} \sqcup \text{«}x'\text{»} = (\mathit{if}\ x = x'\ \mathit{then}\ \text{«}x\text{»}\ \mathit{else}\ \sharp)$
| $\boldsymbol{\lambda}[t] \sqcup \boldsymbol{\lambda}[t'] = \boldsymbol{\lambda}[t \sqcup t']$
| $\boldsymbol{\lambda}[t] \circ u \sqcup \boldsymbol{\lambda}[t'] \bullet u' = \boldsymbol{\lambda}[(t \sqcup t')] \bullet (u \sqcup u')$
| $\boldsymbol{\lambda}[t] \bullet u \sqcup \boldsymbol{\lambda}[t'] \circ u' = \boldsymbol{\lambda}[(t \sqcup t')] \bullet (u \sqcup u')$
| $t \circ u \sqcup t' \circ u' = (t \sqcup t') \circ (u \sqcup u')$
| $\boldsymbol{\lambda}[t] \bullet u \sqcup \boldsymbol{\lambda}[t'] \bullet u' = \boldsymbol{\lambda}[(t \sqcup t')] \bullet (u \sqcup u')$
| - $\sqcup$ - $= \sharp$

lemma *Join-sym*:
shows $t \sqcup u = u \sqcup t$
⟨*proof*⟩

lemma *Src-Join*:
shows *Coinitial* $t\ u \Longrightarrow$ *Src* $(t \sqcup u) =$ *Src* t
⟨*proof*⟩

lemma *resid-Join*:
shows *Coinitial* $t\ u \Longrightarrow (t \sqcup u) \setminus u = t \setminus u$
⟨*proof*⟩

lemma *prfx-Join*:
shows *Coinitial* $t\ u \Longrightarrow u \lesssim t \sqcup u$
⟨*proof*⟩

lemma *Ide-resid-Join*:
shows *Coinitial* $t\ u \Longrightarrow$ *Ide* $(u \setminus (t \sqcup u))$
⟨*proof*⟩

lemma *join-of-Join*:
assumes *Coinitial* $t\ u$

shows *join-of* t u $(t \sqcup u)$
⟨*proof*⟩

sublocale *rts-with-joins resid*
⟨*proof*⟩

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

3.2.5 Simulations from Syntactic Constructors

Here we show that the syntactic constructors *Lam* and *App*, as well as the substitution operation *subst*, determine simulations. In addition, we show that *Beta* determines a transformation from *App* ∘ (*Lam* × *Id*) to *subst*.

abbreviation Lam_{ext}
where $Lam_{ext}\ t \equiv$ *if arr t then* $\boldsymbol{\lambda}[t]$ *else* ♯

lemma *Lam-is-simulation*:
shows *simulation resid resid* Lam_{ext}
⟨*proof*⟩

interpretation *Lam*: *simulation resid resid* Lam_{ext}
⟨*proof*⟩

interpretation $\Lambda x\Lambda$: *product-of-weakly-extensional-rts resid resid*
⟨*proof*⟩

abbreviation App_{ext}
where $App_{ext}\ t \equiv$ *if* $\Lambda x\Lambda$.*arr t then fst t* ∘ *snd t else* ♯

lemma *App-is-binary-simulation*:
shows *binary-simulation resid resid resid* App_{ext}
⟨*proof*⟩

interpretation *App*: *binary-simulation resid resid resid* App_{ext}
⟨*proof*⟩

abbreviation $subst_{ext}$
where $subst_{ext} \equiv \lambda t.$ *if* $\Lambda x\Lambda$.*arr t then subst* (*snd t*) (*fst t*) *else* ♯

lemma *subst-is-binary-simulation*:
shows *binary-simulation resid resid resid* $subst_{ext}$
⟨*proof*⟩

interpretation *subst*: *binary-simulation resid resid resid* $subst_{ext}$
⟨*proof*⟩

interpretation *Id*: *identity-simulation resid*

⟨*proof*⟩
interpretation *Lam-Id*: *product-simulation resid resid resid resid* Lam_{ext} *Id.map*
⟨*proof*⟩
interpretation *App-o-Lam-Id*: *composite-simulation* $\Lambda x\Lambda$*.resid* $\Lambda x\Lambda$*.resid resid Lam-Id.map* App_{ext}
⟨*proof*⟩

abbreviation $Beta_{ext}$
where $Beta_{ext}$ $t \equiv$ *if* $\Lambda x\Lambda$*.arr t then* $\boldsymbol{\lambda}$[*fst t*] $\bullet$ *snd t else* $\sharp$

lemma *Beta-is-transformation*:
shows *transformation* $\Lambda x\Lambda$*.resid resid App-o-Lam-Id.map* $subst_{ext}$ $Beta_{ext}$
⟨*proof*⟩

The next two results are used to show that mapping App over lists of transitions preserves paths.

lemma *App-is-simulation1*:
assumes *ide a*
shows *simulation resid resid* (λt. *if arr t then t* $\circ$ *a else* $\sharp$)
⟨*proof*⟩

lemma *App-is-simulation2*:
assumes *ide a*
shows *simulation resid resid* (λt. *if arr t then a* $\circ$ *t else* $\sharp$)
⟨*proof*⟩

3.2.6 Reduction and Conversion

Here we define the usual relations of reduction and conversion. Reduction is the least transitive relation that relates a to b if there exists an arrow t having a as its source and b as its target. Conversion is the least transitive relation that relates a to b if there exists an arrow t in either direction between a and b.

inductive *red*
where *Arr t* $\Longrightarrow$ *red* (*Src t*) (*Trg t*)
| ⟦*red a b*; *red b c*⟧ $\Longrightarrow$ *red a c*

inductive *cnv*
where *Arr t* $\Longrightarrow$ *cnv* (*Src t*) (*Trg t*)
| *Arr t* $\Longrightarrow$ *cnv* (*Trg t*) (*Src t*)
| ⟦*cnv a b*; *cnv b c*⟧ $\Longrightarrow$ *cnv a c*

lemma *cnv-refl*:
assumes *Ide a*
shows *cnv a a*
⟨*proof*⟩

lemma *cnv-sym*:
shows *cnv a b* $\Longrightarrow$ *cnv b a*

⟨*proof*⟩

lemma *red-imp-cnv*:
shows *red a b* $\Longrightarrow$ *cnv a b*
⟨*proof*⟩

end

We now define a locale that extends the residuation operation defined above to paths, using general results that have already been shown for paths in an RTS. In particular, we are taking advantage of the general proof of the Cube Lemma for residuation on paths.

Our immediate goal is to prove the Church-Rosser theorem, so we first prove a lemma that connects the reduction relation to paths. Later, we will prove many more facts in this locale, thereby developing a general framework for reasoning about reduction paths in the λ-calculus.

locale *reduction-paths* =
Λ: *lambda-calculus*
begin

sublocale Λ: *rts* Λ.*resid*
⟨*proof*⟩
sublocale *paths-in-weakly-extensional-rts* Λ.*resid*
⟨*proof*⟩
sublocale *paths-in-confluent-rts* Λ.*resid*
⟨*proof*⟩

notation Λ.*resid* (**infix** ‹$\backslash$› *70*)
notation Λ.*con* (**infix** ‹$\frown$› *50*)
notation Λ.*prfx* (**infix** ‹$\lesssim$› *50*)
notation Λ.*cong* (**infix** ‹$\sim$› *50*)

notation *Resid* (**infix** ‹$^*\backslash^*$› *70*)
notation *Resid1x* (**infix** ‹$^1\backslash^*$› *70*)
notation *Residx1* (**infix** ‹$^*\backslash^1$› *70*)
notation *con* (**infix** ‹$^*\frown^*$› *50*)
notation *prfx* (**infix** ‹$^*\lesssim^*$› *50*)
notation *cong* (**infix** ‹$^*\sim^*$› *50*)

lemma *red-iff*:
shows Λ.*red a b* $\longleftrightarrow$ ($\exists$ *T*. *Arr T* $\wedge$ *Src T* = *a* $\wedge$ *Trg T* = *b*)
⟨*proof*⟩

end

3.2.7 The Church-Rosser Theorem

context *lambda-calculus*
begin

interpretation Λx: *reduction-paths* $\langle proof \rangle$

theorem *church-rosser*:
shows $cnv\ a\ b \Longrightarrow \exists\, c.\ red\ a\ c \wedge red\ b\ c$
$\langle proof \rangle$

corollary *weak-diamond*:
assumes $red\ a\ b$ **and** $red\ a\ b'$
obtains c **where** $red\ b\ c$ **and** $red\ b'\ c$
$\langle proof \rangle$

As a consequence of the Church-Rosser Theorem, the collection of all reduction paths forms a coherent normal sub-RTS of the RTS of reduction paths, and on identities the congruence induced by this normal sub-RTS coincides with convertibility. The quotient of the λ-calculus RTS by this congruence is then obviously discrete: the only transitions are identities.

interpretation *Red*: *normal-sub-rts* $\Lambda x.Resid$ ‹$Collect\ \Lambda x.Arr$›
$\langle proof \rangle$

interpretation *Red*: *coherent-normal-sub-rts* $\Lambda x.Resid$ ‹$Collect\ \Lambda x.Arr$›
$\langle proof \rangle$

lemma *cnv-iff-Cong*:
assumes $ide\ a$ **and** $ide\ b$
shows $cnv\ a\ b \longleftrightarrow Red.Cong\ [a]\ [b]$
$\langle proof \rangle$

interpretation Λq: *quotient-by-coherent-normal* $\Lambda x.Resid$ ‹$Collect\ \Lambda x.Arr$›
$\langle proof \rangle$

lemma *quotient-by-cnv-is-discrete*:
shows $\Lambda q.arr\ t \longleftrightarrow \Lambda q.ide\ t$
$\langle proof \rangle$

3.2.8 Normalization

A *normal form* is an identity that is not the source of any non-identity arrow.

definition *NF*
where $NF\ a \equiv Ide\ a \wedge (\forall\, t.\ Arr\ t \wedge Src\ t = a \longrightarrow Ide\ t)$

lemma (**in** *reduction-paths*) *path-from-NF-is-Ide*:
assumes $\Lambda.NF\ a$
shows $[\![Arr\ U;\ Src\ U = a]\!] \Longrightarrow Ide\ U$
$\langle proof \rangle$

lemma *NF-reduct-is-trivial*:
assumes $NF\ a$ **and** $red\ a\ b$
shows $a = b$

$\langle proof \rangle$

lemma *NF-unique*:
assumes *red t u* **and** *red t u'* **and** *NF u* **and** *NF u'*
shows $u = u'$
$\langle proof \rangle$

A term is *normalizable* if it is an identity that is reducible to a normal form.

definition *normalizable*
where *normalizable a* $\equiv$ *Ide a* $\wedge$ ($\exists$ *b*. *red a b* $\wedge$ *NF b*)

end

3.3 Reduction Paths

In this section we develop further facts about reduction paths for the λ-calculus.

context *reduction-paths*
begin

3.3.1 Sources and Targets

lemma *Srcs-simp*$_{\Lambda P}$:
shows *Arr t* $\Longrightarrow$ *Srcs t* = $\{\Lambda.Src\ (hd\ t)\}$
$\langle proof \rangle$

lemma *Trgs-simp*$_{\Lambda P}$:
shows *Arr t* $\Longrightarrow$ *Trgs t* = $\{\Lambda.Trg\ (last\ t)\}$
$\langle proof \rangle$

lemma *sources-single-Src* [*simp*]:
assumes $\Lambda.Arr\ t$
shows *sources* [$\Lambda.Src\ t$] = *sources* [*t*]
$\langle proof \rangle$

lemma *targets-single-Trg* [*simp*]:
assumes $\Lambda.Arr\ t$
shows *targets* [$\Lambda.Trg\ t$] = *targets* [*t*]
$\langle proof \rangle$

lemma *sources-single-Trg* [*simp*]:
assumes $\Lambda.Arr\ t$
shows *sources* [$\Lambda.Trg\ t$] = *targets* [*t*]
$\langle proof \rangle$

lemma *targets-single-Src* [*simp*]:
assumes $\Lambda.Arr\ t$
shows *targets* [$\Lambda.Src\ t$] = *sources* [*t*]
$\langle proof \rangle$

lemma *single-Src-hd-in-sources*:
assumes *Arr T*
shows $[\Lambda.Src\ (hd\ T)] \in sources\ T$
⟨*proof*⟩

lemma *single-Trg-last-in-targets*:
assumes *Arr T*
shows $[\Lambda.Trg\ (last\ T)] \in targets\ T$
⟨*proof*⟩

lemma *in-sources-iff*:
assumes *Arr T*
shows $A \in sources\ T \longleftrightarrow A\ {}^*{\sim}^*\ [\Lambda.Src\ (hd\ T)]$
⟨*proof*⟩

lemma *in-targets-iff*:
assumes *Arr T*
shows $B \in targets\ T \longleftrightarrow B\ {}^*{\sim}^*\ [\Lambda.Trg\ (last\ T)]$
⟨*proof*⟩

lemma *seq-imp-cong-Trg-last-Src-hd*:
assumes *seq T U*
shows $\Lambda.Trg\ (last\ T) \sim \Lambda.Src\ (hd\ U)$
⟨*proof*⟩

lemma $sources\text{-}char_{\Lambda P}$:
shows $sources\ T = \{A.\ Arr\ T \wedge A\ {}^*{\sim}^*\ [\Lambda.Src\ (hd\ T)]\}$
⟨*proof*⟩

lemma $targets\text{-}char_{\Lambda P}$:
shows $targets\ T = \{B.\ Arr\ T \wedge B\ {}^*{\sim}^*\ [\Lambda.Trg\ (last\ T)]\}$
⟨*proof*⟩

lemma *Src-hd-eqI*:
assumes $T\ {}^*{\sim}^*\ U$
shows $\Lambda.Src\ (hd\ T) = \Lambda.Src\ (hd\ U)$
⟨*proof*⟩

lemma *Trg-last-eqI*:
assumes $T\ {}^*{\sim}^*\ U$
shows $\Lambda.Trg\ (last\ T) = \Lambda.Trg\ (last\ U)$
⟨*proof*⟩

lemma *Trg-last-Src-hd-eqI*:
assumes *seq T U*
shows $\Lambda.Trg\ (last\ T) = \Lambda.Src\ (hd\ U)$
⟨*proof*⟩

lemma $seqI_{\Lambda P}$ [*intro*]:
assumes *Arr T* **and** *Arr U* **and** $\Lambda.Trg$ (*last T*) = $\Lambda.Src$ (*hd U*)
shows *seq T U*
⟨*proof*⟩

lemma $conI_{\Lambda P}$ [*intro*]:
assumes *arr T* **and** *arr U* **and** $\Lambda.Src$ (*hd T*) = $\Lambda.Src$ (*hd U*)
shows $T \ {}^*\frown^* \ U$
⟨*proof*⟩

3.3.2 Mapping Constructors over Paths

lemma *Arr-map-Lam*:
assumes *Arr T*
shows *Arr* (*map* Λ.*Lam T*)
⟨*proof*⟩

lemma *Arr-map-App1*:
assumes Λ.*Ide b* **and** *Arr T*
shows *Arr* (*map* ($\lambda t.\ t \circ b$) *T*)
⟨*proof*⟩

lemma *Arr-map-App2*:
assumes Λ.*Ide a* **and** *Arr T*
shows *Arr* (*map* (Λ.*App a*) *T*)
⟨*proof*⟩

interpretation Λ_{Lam}: *source-replete-sub-rts* Λ.*resid* ‹$\lambda t.\ \Lambda.Arr\ t \wedge \Lambda.is\text{-}Lam\ t$›
⟨*proof*⟩

interpretation *un-Lam*: *simulation* Λ_{Lam}.*resid* Λ.*resid*
‹$\lambda t.$ *if* Λ_{Lam}.*arr t then* Λ.*un-Lam t else* ♯›
⟨*proof*⟩

lemma *Arr-map-un-Lam*:
assumes *Arr T* **and** *set T* ⊆ *Collect* Λ.*is-Lam*
shows *Arr* (*map* Λ.*un-Lam T*)
⟨*proof*⟩

interpretation Λ_{App}: *source-replete-sub-rts* Λ.*resid* ‹$\lambda t.\ \Lambda.Arr\ t \wedge \Lambda.is\text{-}App\ t$›
⟨*proof*⟩

interpretation *un-App1*: *simulation* Λ_{App}.*resid* Λ.*resid*
‹$\lambda t.$ *if* Λ_{App}.*arr t then* Λ.*un-App1 t else* ♯›
⟨*proof*⟩

interpretation *un-App2*: *simulation* Λ_{App}.*resid* Λ.*resid*
‹$\lambda t.$ *if* Λ_{App}.*arr t then* Λ.*un-App2 t else* ♯›
⟨*proof*⟩

lemma *Arr-map-un-App1*:
assumes *Arr T* **and** *set T* $\subseteq$ *Collect* Λ.*is-App*
shows *Arr* (*map* Λ.*un-App1 T*)
⟨*proof*⟩

lemma *Arr-map-un-App2*:
assumes *Arr T* **and** *set T* $\subseteq$ *Collect* Λ.*is-App*
shows *Arr* (*map* Λ.*un-App2 T*)
⟨*proof*⟩

lemma *map-App-map-un-App1*:
shows ⟦*Arr U*; *set U* $\subseteq$ *Collect* Λ.*is-App*; Λ.*Ide b*; Λ.*un-App2* ‘ *set U* $\subseteq \{b\}$⟧ $\Longrightarrow$
map (λt. Λ.*App t b*) (*map* Λ.*un-App1 U*) = *U*
⟨*proof*⟩

lemma *map-App-map-un-App2*:
shows ⟦*Arr U*; *set U* $\subseteq$ *Collect* Λ.*is-App*; Λ.*Ide a*; Λ.*un-App1* ‘ *set U* $\subseteq \{a\}$⟧ $\Longrightarrow$
map (Λ.*App a*) (*map* Λ.*un-App2 U*) = *U*
⟨*proof*⟩

lemma *map-Lam-Resid*:
assumes *coinitial T U*
shows *map* Λ.*Lam* (T $^*\backslash^*$ U) = *map* Λ.*Lam* T $^*\backslash^*$ *map* Λ.*Lam U*
⟨*proof*⟩

lemma *map-App1-Resid*:
assumes Λ.*Ide x* **and** *coinitial T U*
shows *map* (Λ.*App x*) (T $^*\backslash^*$ U) = *map* (Λ.*App x*) T $^*\backslash^*$ *map* (Λ.*App x*) *U*
⟨*proof*⟩

lemma *map-App2-Resid*:
assumes Λ.*Ide x* **and** *coinitial T U*
shows *map* (λt. $t \circ x$) (T $^*\backslash^*$ U) = *map* (λt. $t \circ x$) T $^*\backslash^*$ *map* (λt. $t \circ x$) *U*
⟨*proof*⟩

lemma *cong-map-Lam*:
shows T $^*\sim^*$ U $\Longrightarrow$ *map* Λ.*Lam* T $^*\sim^*$ *map* Λ.*Lam U*
⟨*proof*⟩

lemma *cong-map-App1*:
shows ⟦Λ.*Ide x*; T $^*\sim^*$ U⟧ $\Longrightarrow$ *map* (Λ.*App x*) T $^*\sim^*$ *map* (Λ.*App x*) *U*
⟨*proof*⟩

lemma *cong-map-App2*:
shows ⟦Λ.*Ide x*; T $^*\sim^*$ U⟧ $\Longrightarrow$ *map* (λX. $X \circ x$) T $^*\sim^*$ *map* (λX. $X \circ x$) *U*
⟨*proof*⟩

3.3.3 Decomposition of 'App Paths'

The following series of results is aimed at showing that a reduction path, all of whose transitions have *App* as their top-level constructor, can be factored up to congruence into a reduction path in which only the "rator" components are reduced, followed by a reduction path in which only the "rand" components are reduced.

lemma *orthogonal-App-single-single*:
assumes $\Lambda.\mathit{Arr}\ t$ **and** $\Lambda.\mathit{Arr}\ u$
shows $[\Lambda.\mathit{Src}\ t \circ u]\ {}^*\backslash^*\ [t \circ \Lambda.\mathit{Src}\ u] = [\Lambda.\mathit{Trg}\ t \circ u]$
and $[t \circ \Lambda.\mathit{Src}\ u]\ {}^*\backslash^*\ [\Lambda.\mathit{Src}\ t \circ u] = [t \circ \Lambda.\mathit{Trg}\ u]$
$\langle \mathit{proof} \rangle$

lemma *orthogonal-App-single-Arr*:
shows $[\![\mathit{Arr}\ [t];\ \mathit{Arr}\ U]\!] \Longrightarrow$
$\mathit{map}\ (\Lambda.\mathit{App}\ (\Lambda.\mathit{Src}\ t))\ U\ {}^*\backslash^*\ [t \circ \Lambda.\mathit{Src}\ (\mathit{hd}\ U)] = \mathit{map}\ (\Lambda.\mathit{App}\ (\Lambda.\mathit{Trg}\ t))\ U\ \wedge$
$[t \circ \Lambda.\mathit{Src}\ (\mathit{hd}\ U)]\ {}^*\backslash^*\ \mathit{map}\ (\Lambda.\mathit{App}\ (\Lambda.\mathit{Src}\ t))\ U = [t \circ \Lambda.\mathit{Trg}\ (\mathit{last}\ U)]$
$\langle \mathit{proof} \rangle$

lemma *orthogonal-App-Arr-Arr*:
shows $[\![\mathit{Arr}\ T;\ \mathit{Arr}\ U]\!] \Longrightarrow$
$\mathit{map}\ (\Lambda.\mathit{App}\ (\Lambda.\mathit{Src}\ (\mathit{hd}\ T)))\ U\ {}^*\backslash^*\ \mathit{map}\ (\lambda X.\ \Lambda.\mathit{App}\ X\ (\Lambda.\mathit{Src}\ (\mathit{hd}\ U)))\ T =$
$\mathit{map}\ (\Lambda.\mathit{App}\ (\Lambda.\mathit{Trg}\ (\mathit{last}\ T)))\ U\ \wedge$
$\mathit{map}\ (\lambda X.\ X \circ \Lambda.\mathit{Src}\ (\mathit{hd}\ U))\ T\ {}^*\backslash^*\ \mathit{map}\ (\Lambda.\mathit{App}\ (\Lambda.\mathit{Src}\ (\mathit{hd}\ T)))\ U =$
$\mathit{map}\ (\lambda X.\ X \circ \Lambda.\mathit{Trg}\ (\mathit{last}\ U))\ T$
$\langle \mathit{proof} \rangle$

lemma *orthogonal-App-cong*:
assumes $\mathit{Arr}\ T$ **and** $\mathit{Arr}\ U$
shows $\mathit{map}\ (\lambda X.\ X \circ \Lambda.\mathit{Src}\ (\mathit{hd}\ U))\ T\ @\ \mathit{map}\ (\Lambda.\mathit{App}\ (\Lambda.\mathit{Trg}\ (\mathit{last}\ T)))\ U\ {}^*\!\sim^*$
$\mathit{map}\ (\Lambda.\mathit{App}\ (\Lambda.\mathit{Src}\ (\mathit{hd}\ T)))\ U\ @\ \mathit{map}\ (\lambda X.\ X \circ \Lambda.\mathit{Trg}\ (\mathit{last}\ U))\ T$

$\langle \mathit{proof} \rangle$

We arrive at the final objective of this section: factorization, up to congruence, of a path whose transitions all have *App* as the top-level constructor, into the composite of a path that reduces only the "rators" and a path that reduces only the "rands".

lemma *map-App-decomp*:
shows $[\![\mathit{Arr}\ U;\ \mathit{set}\ U \subseteq \mathit{Collect}\ \Lambda.\mathit{is\text{-}App}]\!] \Longrightarrow$
$\mathit{map}\ (\lambda X.\ X \circ \Lambda.\mathit{Src}\ (\Lambda.\mathit{un\text{-}App2}\ (\mathit{hd}\ U)))\ (\mathit{map}\ \Lambda.\mathit{un\text{-}App1}\ U)\ @$
$\mathit{map}\ (\lambda X.\ \Lambda.\mathit{Trg}\ (\Lambda.\mathit{un\text{-}App1}\ (\mathit{last}\ U)) \circ X)\ (\mathit{map}\ \Lambda.\mathit{un\text{-}App2}\ U)\ {}^*\!\sim^*$
U
$\langle \mathit{proof} \rangle$

3.3.4 Miscellaneous

lemma *Resid-parallel*:
assumes $\mathit{cong}\ t\ t'$ **and** $\mathit{coinitial}\ t\ u$
shows $u\ {}^*\backslash^*\ t = u\ {}^*\backslash^*\ t'$
$\langle \mathit{proof} \rangle$

lemma *set-Ide-subset-single-hd*:
shows $Ide\ T \Longrightarrow set\ T \subseteq \{hd\ T\}$
⟨*proof*⟩

A single parallel reduction with *Beta* as the top-level operator factors, up to congruence, either as a path in which the top-level redex is contracted first, or as a path in which the top-level redex is contracted last.

lemma *Beta-decomp*:
assumes $\Lambda.Arr\ t$ **and** $\Lambda.Arr\ u$
shows $[\boldsymbol{\lambda}[\Lambda.Src\ t] \bullet \Lambda.Src\ u]\ @\ [\Lambda.subst\ u\ t]\ {}^*\!\sim^*\ [\boldsymbol{\lambda}[t] \bullet u]$
and $[\boldsymbol{\lambda}[t] \circ u]\ @\ [\boldsymbol{\lambda}[\Lambda.Trg\ t] \bullet \Lambda.Trg\ u]\ {}^*\!\sim^*\ [\boldsymbol{\lambda}[t] \bullet u]$
⟨*proof*⟩

If a reduction path follows an initial reduction whose top-level constructor is *Lam*, then all the terms in the path have *Lam* as their top-level constructor.

lemma *seq-Lam-Arr-implies*:
shows $[\![seq\ [t]\ U;\ \Lambda.is\text{-}Lam\ t]\!] \Longrightarrow set\ U \subseteq Collect\ \Lambda.is\text{-}Lam$
⟨*proof*⟩

lemma *seq-map-un-Lam*:
assumes $seq\ [\boldsymbol{\lambda}[t]]\ U$
shows $seq\ [t]\ (map\ \Lambda.un\text{-}Lam\ U)$
⟨*proof*⟩

end

3.4 Developments

A *development* is a reduction path from a term in which at each step exactly one redex is contracted, and the only redexes that are contracted are those that are residuals of redexes present in the original term. That is, no redexes are contracted that were newly created as a result of the previous reductions. The main theorem about developments is the Finite Developments Theorem, which states that all developments are finite. A proof of this theorem was published by Hindley [6], who attributes the result to Schroer [9]. Other proofs were published subsequently. Here we follow the paper by de Vrijer [5], which may in some sense be considered the definitive work because de Vrijer's proof gives an exact bound on the number of steps in a development. Since de Vrijer used a classical, named-variable representation of λ-terms, for the formalization given in the present article it was necessary to find the correct way to adapt de Vrijer's proof to the de Bruijn index representation of terms. I found this to be a somewhat delicate matter and to my knowledge it has not been done previously.

context *lambda-calculus*
begin

We define an *elementary reduction* defined to be a term with exactly one marked redex. These correspond to the most basic computational steps.

fun *elementary-reduction*
where $\textit{elementary-reduction}\ \sharp \longleftrightarrow \textit{False}$
  $\mid \textit{elementary-reduction}\ («\text{-}») \longleftrightarrow \textit{False}$
  $\mid \textit{elementary-reduction}\ \boldsymbol{\lambda}[t] \longleftrightarrow \textit{elementary-reduction}\ t$
  $\mid \textit{elementary-reduction}\ (t \circ u) \longleftrightarrow$
    $(\textit{elementary-reduction}\ t \wedge \textit{Ide}\ u) \vee (\textit{Ide}\ t \wedge \textit{elementary-reduction}\ u)$
  $\mid \textit{elementary-reduction}\ (\boldsymbol{\lambda}[t] \bullet u) \longleftrightarrow \textit{Ide}\ t \wedge \textit{Ide}\ u$

It is tempting to imagine that elementary reductions would be atoms with respect to the preorder $\lesssim$, but this is not necessarily the case. For example, suppose $t = \boldsymbol{\lambda}[«1»] \bullet (\boldsymbol{\lambda}[«0»] \circ «0»)$ and $u = \boldsymbol{\lambda}[«1»] \bullet (\boldsymbol{\lambda}[«0»] \bullet «0»)$. Then t is an elementary reduction, $u \lesssim t$ (in fact $u \sim t$) but u is not an identity, nor is it elementary.

lemma *elementary-reduction-is-arr*:
shows $\textit{elementary-reduction}\ t \Longrightarrow \textit{arr}\ t$
  ⟨*proof*⟩

lemma *elementary-reduction-not-ide*:
shows $\textit{elementary-reduction}\ t \Longrightarrow \neg\ \textit{ide}\ t$
  ⟨*proof*⟩

lemma *elementary-reduction-Raise-iff*:
shows $\bigwedge d\ n.\ \textit{elementary-reduction}\ (\textit{Raise}\ d\ n\ t) \longleftrightarrow \textit{elementary-reduction}\ t$
  ⟨*proof*⟩

lemma *elementary-reduction-Lam-iff*:
shows $\textit{is-Lam}\ t \Longrightarrow \textit{elementary-reduction}\ t \longleftrightarrow \textit{elementary-reduction}\ (\textit{un-Lam}\ t)$
  ⟨*proof*⟩

lemma *elementary-reduction-App-iff*:
shows $\textit{is-App}\ t \Longrightarrow \textit{elementary-reduction}\ t \longleftrightarrow$
    $(\textit{elementary-reduction}\ (\textit{un-App1}\ t) \wedge \textit{ide}\ (\textit{un-App2}\ t)) \vee$
    $(\textit{ide}\ (\textit{un-App1}\ t) \wedge \textit{elementary-reduction}\ (\textit{un-App2}\ t))$
  ⟨*proof*⟩

lemma *elementary-reduction-Beta-iff*:
shows $\textit{is-Beta}\ t \Longrightarrow \textit{elementary-reduction}\ t \longleftrightarrow \textit{ide}\ (\textit{un-Beta1}\ t) \wedge \textit{ide}\ (\textit{un-Beta2}\ t)$
  ⟨*proof*⟩

lemma *cong-elementary-reductions-are-equal*:
shows $[\![\textit{elementary-reduction}\ t;\ \textit{elementary-reduction}\ u;\ t \sim u]\!] \Longrightarrow t = u$
⟨*proof*⟩

An *elementary reduction path* is a path in which each step is an elementary reduction. It will be convenient to regard the empty list as an elementary reduction path, even though it is not actually a path according to our previous definition of that notion.

definition (**in** *reduction-paths*) *elementary-reduction-path*
where $\textit{elementary-reduction-path}\ T \longleftrightarrow$
  $(T = [] \vee \textit{Arr}\ T \wedge \textit{set}\ T \subseteq \textit{Collect}\ \Lambda.\textit{elementary-reduction})$

In the formal definition of "development" given below, we represent a set of redexes simply by a term, in which the occurrences of *Beta* correspond to the redexes in the set. To express the idea that an elementary reduction u is a member of the set of redexes represented by term t, it is not adequate to say $u \lesssim t$. To see this, consider the developments of a term of the form $\boldsymbol{\lambda}[t1] \bullet t2$. Intuitively, such developments should consist of a (possibly empty) initial segment containing only transitions of the form $t1 \circ t2$, followed by a transition of the form $\boldsymbol{\lambda}[u1'] \bullet u2'$, followed by a development of the residual of the original $\boldsymbol{\lambda}[t1] \bullet t2$ after what has come so far. The requirement $u \lesssim \boldsymbol{\lambda}[t1] \bullet t2$ is not a strong enough constraint on the transitions in the initial segment, because $\boldsymbol{\lambda}[u1] \bullet u2 \lesssim \boldsymbol{\lambda}[t1] \bullet t2$ can hold for $t2$ and $u2$ coinitial, but otherwise without any particular relationship between their sets of marked redexes. In particular, this can occur when $u2$ and $t2$ occur as subterms that can be deleted by the contraction of an outer redex. So we need to introduce a notion of containment between terms that is stronger and more "syntactic" than $\lesssim$. The notion "subsumed by" defined below serves this purpose. Term u is subsumed by term t if both terms are arrows with exactly the same form except that t may contain $\boldsymbol{\lambda}[t1] \bullet t2$ (a marked redex) in places where u contains $\boldsymbol{\lambda}[t1] \circ t2$.

fun *subs* (**infix** ‹$\sqsubseteq$› *50*)
where $«i» \sqsubseteq «i'» \longleftrightarrow i = i'$
| $\boldsymbol{\lambda}[t] \sqsubseteq \boldsymbol{\lambda}[t'] \longleftrightarrow t \sqsubseteq t'$
| $t \circ u \sqsubseteq t' \circ u' \longleftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u'$
| $\boldsymbol{\lambda}[t] \circ u \sqsubseteq \boldsymbol{\lambda}[t'] \bullet u' \longleftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u'$
| $\boldsymbol{\lambda}[t] \bullet u \sqsubseteq \boldsymbol{\lambda}[t'] \bullet u' \longleftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u'$
| - $\sqsubseteq$ - $\longleftrightarrow$ *False*

lemma *subs-implies-prfx*:
shows $t \sqsubseteq u \Longrightarrow t \lesssim u$
⟨*proof*⟩

The following is an example showing that two terms can be related by $\lesssim$ without being related by $\sqsubseteq$.

lemma *subs-example*:
shows $\boldsymbol{\lambda}[«1»] \bullet (\boldsymbol{\lambda}[«0»] \bullet «0») \lesssim \boldsymbol{\lambda}[«1»] \bullet (\boldsymbol{\lambda}[«0»] \circ «0») = True$
and $\boldsymbol{\lambda}[«1»] \bullet (\boldsymbol{\lambda}[«0»] \bullet «0») \sqsubseteq \boldsymbol{\lambda}[«1»] \bullet (\boldsymbol{\lambda}[«0»] \circ «0») = False$
⟨*proof*⟩

lemma *subs-Ide*:
shows $[\![ide\ u;\ Src\ t = Src\ u]\!] \Longrightarrow u \sqsubseteq t$
⟨*proof*⟩

lemma *subs-App*:
shows $u \sqsubseteq t1 \circ t2 \longleftrightarrow is\text{-}App\ u \wedge un\text{-}App1\ u \sqsubseteq t1 \wedge un\text{-}App2\ u \sqsubseteq t2$
⟨*proof*⟩

end

context *reduction-paths*

begin

We now formally define a *development of t* to be an elementary reduction path U that is coinitial with $[t]$ and is such that each transition u in U is subsumed by the residual of t along the prefix of U coming before u. Stated another way, each transition in U corresponds to the contraction of a single redex that is the residual of a redex originally marked in t.

fun *development*
where $\mathit{development}\ t\ []\longleftrightarrow \Lambda.\mathit{Arr}\ t$
$\quad|\ \mathit{development}\ t\ (u\ \#\ U)\longleftrightarrow$
$\qquad \Lambda.\mathit{elementary\text{-}reduction}\ u \wedge u \sqsubseteq t \wedge \mathit{development}\ (t\ \backslash\ u)\ U$

lemma *development-imp-Arr*:
assumes *development t U*
shows $\Lambda.\mathit{Arr}\ t$
$\langle \mathit{proof}\rangle$

lemma *development-Ide*:
shows $\Lambda.\mathit{Ide}\ t \Longrightarrow \mathit{development}\ t\ U \longleftrightarrow U = []$
$\langle \mathit{proof}\rangle$

lemma *development-implies*:
shows $\mathit{development}\ t\ U \Longrightarrow \mathit{elementary\text{-}reduction\text{-}path}\ U \wedge (U \neq [] \longrightarrow U\ {}^*\lesssim^*\ [t])$
$\langle \mathit{proof}\rangle$

The converse of the previous result does not hold, because there could be a stage i at which $u_i \lesssim t_i$, but t_i deletes the redex contracted in u_i, so there is nothing forcing that redex to have been originally marked in t. So U being a development of t is a stronger property than U just being an elementary reduction path such that $U\ {}^*\lesssim^*\ [t]$.

lemma *development-append*:
shows $[\![\mathit{development}\ t\ U;\ \mathit{development}\ (t\ {}^1\backslash^*\ U)\ V]\!] \Longrightarrow \mathit{development}\ t\ (U\ @\ V)$
$\langle \mathit{proof}\rangle$

lemma *development-map-Lam*:
shows $\mathit{development}\ t\ T \Longrightarrow \mathit{development}\ \boldsymbol{\lambda}[t]\ (\mathit{map}\ \Lambda.\mathit{Lam}\ T)$
$\langle \mathit{proof}\rangle$

lemma *development-map-App-1*:
shows $[\![\mathit{development}\ t\ T;\ \Lambda.\mathit{Arr}\ u]\!] \Longrightarrow \mathit{development}\ (t \circ u)\ (\mathit{map}\ (\lambda x.\ x \circ \Lambda.\mathit{Src}\ u)\ T)$
$\langle \mathit{proof}\rangle$

lemma *development-map-App-2*:
shows $[\![\Lambda.\mathit{Arr}\ t;\ \mathit{development}\ u\ U]\!] \Longrightarrow \mathit{development}\ (t \circ u)\ (\mathit{map}\ (\lambda x.\ \Lambda.\mathit{App}\ (\Lambda.\mathit{Src}\ t)\ x)$
$U)$
$\langle \mathit{proof}\rangle$

3.4.1 Finiteness of Developments

A term t has the finite developments property if there exists a finite value that bounds the length of all developments of t. The goal of this section is to prove the Finite Developments Theorem: every term has the finite developments property.

definition *FD*
where $FD\ t \equiv \exists\, n.\ \forall\, U.\ development\ t\ U \longrightarrow length\ U \leq n$

end

In [6], Hindley proceeds by using structural induction to establish a bound on the length of a development of a term. The only case that poses any difficulty is the case of a β-redex, which is $\boldsymbol{\lambda}[t] \bullet u$ in the notation used here. He notes that there is an easy bound on the length of a development of a special form in which all the contractions of residuals of t occur before the contraction of the top-level redex. The development first takes $\boldsymbol{\lambda}[t] \bullet u$ to $\boldsymbol{\lambda}[t'] \bullet u'$, then to $subst\ u'\ t'$, then continues with independent developments of u'. The number of independent developments of u' is given by the number of free occurrences of *Var 0* in t'. As there can be only finitely many such t', we can use the maximum number of free occurrences of *Var 0* over all such t' to bound the steps in the independent developments of u'.

In the general case, the problem is that reductions of residuals of t can increase the number of free occurrences of *Var 0*, so we can't readily count them at any particular stage. Hindley shows that developments in which there are reductions of residuals of t that occur after the contraction of the top-level redex are equivalent to reductions of the special form, by a transformation with a bounded increase in length. This can be considered as a weak form of standardization for developments.

A later paper by de Vrijer [5] obtains an explicit function for the exact number of steps in a development of maximal length. His proof is very straightforward and amenable to formalization, and it is what we follow here. The main issue for us is that de Vrijer uses a classical representation of λ-terms, with variable names and α-equivalence, whereas here we are using de Bruijn indices. This means that we have to discover the correct modification of de Vrijer's definitions to apply to the present situation.

context *lambda-calculus*
begin

Our first definition is that of the "multiplicity" of a free variable in a term. This is a count of the maximum number of times a variable could occur free in a term reachable in a development. The main issue in adjusting to de Bruijn indices is that the same variable will have different indices depending on the depth at which it occurs in the term. So, we need to keep track of how the indices of variables change as we move through the term. Our modified definitions adjust the parameter to the multiplicity function on each recursive call, to account for the contextual depth (*i.e.* the number of binders on a path from the root of the term).

The definition of this function is readily understandable, except perhaps for the *Beta* case. The multiplicity $mtp\ x\ (\boldsymbol{\lambda}[t] \bullet u)$ has to be at least as large as $mtp\ x\ (\boldsymbol{\lambda}[t] \circ u)$, to account for developments in which the top-level redex is not contracted. However, if the top-level redex $\boldsymbol{\lambda}[t] \bullet u$ is contracted, then the contractum is *subst u t*, so the multiplicity has to be at least as large as *mtp x (subst u t)*. This leads to the relation:

$$mtp\ x\ (\boldsymbol{\lambda}[t] \bullet u) = max\ (mtp\ x\ (\boldsymbol{\lambda}[t] \circ u))\ (mtp\ x\ (subst\ u\ t))$$

This is not directly suitable for use in a definition of the function *mtp*, because proving the termination is problematic. Instead, we have to guess the correct expression for *mtp x (subst u t)* and use that.

Now, each variable x in *subst u t* other than the variable *0* that is substituted for still has all the occurrences that it does in $\boldsymbol{\lambda}[t]$. In addition, the variable being substituted for (which has index *0* in the outermost context of t) will in general have multiple free occurrences in t, with a total multiplicity given by *mtp 0 t*. The substitution operation replaces each free occurrence by u, which has the effect of multiplying the multiplicity of a variable x in t by a factor of *mtp 0 t*. These considerations lead to the following:

$$mtp\ x\ (\boldsymbol{\lambda}[t] \bullet u) = max\ (mtp\ x\ \boldsymbol{\lambda}[t] + mtp\ x\ u)\ (mtp\ x\ \boldsymbol{\lambda}[t] + mtp\ x\ u * mtp\ 0\ t)$$

However, we can simplify this to:

$$mtp\ x\ (\boldsymbol{\lambda}[t] \bullet u) = mtp\ x\ \boldsymbol{\lambda}[t] + mtp\ x\ u * max\ 1\ (mtp\ 0\ t)$$

and replace the *mtp x* $\boldsymbol{\lambda}[t]$ by *mtp (Suc x) t* to simplify the ordering necessary for the termination proof and allow it to be done automatically.

The final result is perhaps about the first thing one would think to write down, but there are possible ways to go wrong and it is of course still necessary to discover the proper form required for the various induction proofs. I followed a long path of rather more complicated-looking definitions, until I eventually managed to find the proper inductive forms for all the lemmas and eventually arrive back at this definition.

fun $mtp :: nat \Rightarrow lambda \Rightarrow nat$
where $mtp\ x\ \sharp = 0$
  $|\ mtp\ x\ «z» = (if\ z = x\ then\ 1\ else\ 0)$
  $|\ mtp\ x\ \boldsymbol{\lambda}[t] = mtp\ (Suc\ x)\ t$
  $|\ mtp\ x\ (t \circ u) = mtp\ x\ t + mtp\ x\ u$
  $|\ mtp\ x\ (\boldsymbol{\lambda}[t] \bullet u) = mtp\ (Suc\ x)\ t + mtp\ x\ u * max\ 1\ (mtp\ 0\ t)$

The multiplicity function generalizes the free variable predicate. This is not actually used, but is included for explanatory purposes.

lemma *mtp-gt-0-iff-in-FV*:
shows $mtp\ x\ t > 0 \longleftrightarrow x \in FV\ t$
⟨*proof*⟩

We now establish a fact about commutation of multiplicity and Raise that will be needed subsequently.

lemma *mtpE-eq-Raise*:
shows $x < d \Longrightarrow mtp\ x\ (Raise\ d\ k\ t) = mtp\ x\ t$
⟨*proof*⟩

lemma *mtp-Raise-ind*:
shows $[\![l \leq d;\ size\ t \leq s]\!] \Longrightarrow mtp\ (x + d + k)\ (Raise\ l\ k\ t) = mtp\ (x + d)\ t$
⟨*proof*⟩

lemma *mtp-Raise*:
assumes $l \leq d$
shows $mtp\ (x + d + k)\ (Raise\ l\ k\ t) = mtp\ (x + d)\ t$
⟨*proof*⟩

lemma *mtp-Raise′*:
shows $mtp\ l\ (Raise\ l\ (Suc\ k)\ t) = 0$
⟨*proof*⟩

lemma *mtp-raise*:
shows $mtp\ (x + Suc\ d)\ (raise\ d\ t) = mtp\ (Suc\ x)\ t$
⟨*proof*⟩

lemma *mtp-Subst-cancel*:
shows $mtp\ k\ (Subst\ (Suc\ d + k)\ u\ t) = mtp\ k\ t$
⟨*proof*⟩

lemma mtp_0*-Subst-cancel*:
shows $mtp\ 0\ (Subst\ (Suc\ d)\ u\ t) = mtp\ 0\ t$
⟨*proof*⟩

We can now (!) prove the desired generalization of de Vrijer's formula for the commutation of multiplicity and substitution. This is the main lemma whose form is difficult to find. To get this right, the proper relationships have to exist between the various depth parameters to *Subst* and the arguments to *mtp*.

lemma *mtp-Subst′*:
shows $mtp\ (x + Suc\ d)\ (Subst\ d\ u\ t) = mtp\ (x + Suc\ (Suc\ d))\ t + mtp\ (Suc\ x)\ u * mtp\ d\ t$
⟨*proof*⟩

The following lemma provides expansions that apply when the parameter to *mtp* is *0*, as opposed to the previous lemma, which only applies for parameters greater than *0*.

lemma *mtp-Subst*:
shows $mtp\ k\ (Subst\ k\ u\ t) = mtp\ (Suc\ k)\ t + mtp\ k\ (raise\ k\ u) * mtp\ k\ t$
⟨*proof*⟩

lemma *mtp0-subst-le*:
shows $mtp\ 0\ (subst\ u\ t) \leq mtp\ 1\ t + mtp\ 0\ u * max\ 1\ (mtp\ 0\ t)$
⟨*proof*⟩

lemma *elementary-reduction-nonincreases-mtp*:
shows $[\![elementary\text{-}reduction\ u;\ u \sqsubseteq t]\!] \Longrightarrow mtp\ x\ (resid\ t\ u) \leq mtp\ x\ t$
⟨*proof*⟩

Next we define the "height" of a term. This counts the number of steps in a development of maximal length of the given term.

fun *hgt*
where $hgt\ \sharp = 0$
  $|\ hgt$ «-» $= 0$
  $|\ hgt\ \boldsymbol{\lambda}[t] = hgt\ t$
  $|\ hgt\ (t \circ u) = hgt\ t + hgt\ u$
  $|\ hgt\ (\boldsymbol{\lambda}[t] \bullet u) = Suc\ (hgt\ t + hgt\ u * max\ 1\ (mtp\ 0\ t))$

lemma *hgt-resid-ide*:
shows $[\![ide\ u;\ u \sqsubseteq t]\!] \Longrightarrow hgt\ (resid\ t\ u) \leq hgt\ t$
  ⟨*proof*⟩

lemma *hgt-Raise*:
shows $hgt\ (Raise\ l\ k\ t) = hgt\ t$
  ⟨*proof*⟩

lemma *hgt-Subst*:
shows $Arr\ u \Longrightarrow hgt\ (Subst\ k\ u\ t) = hgt\ t + hgt\ u * mtp\ k\ t$
⟨*proof*⟩

lemma *elementary-reduction-decreases-hgt*:
shows $[\![elementary\text{-}reduction\ u;\ u \sqsubseteq t]\!] \Longrightarrow hgt\ (t \setminus u) < hgt\ t$
⟨*proof*⟩

end

context *reduction-paths*
begin

lemma *length-devel-le-hgt*:
shows $development\ t\ U \Longrightarrow length\ U \leq \Lambda.hgt\ t$
  ⟨*proof*⟩

We finally arrive at the main result of this section: the Finite Developments Theorem.

theorem *finite-developments*:
shows $FD\ t$
  ⟨*proof*⟩

3.4.2 Complete Developments

A *complete development* is a development in which there are no residuals of originally marked redexes left to contract.

definition *complete-development*
where $complete\text{-}development\ t\ U \equiv development\ t\ U \wedge (\Lambda.Ide\ t \vee [t]\ {}^*\lesssim^*\ U)$

lemma *complete-development-Ide-iff*:
shows $complete\text{-}development\ t\ U \Longrightarrow \Lambda.Ide\ t \longleftrightarrow U = []$
  ⟨*proof*⟩

lemma *complete-development-cons*:

assumes *complete-development t (u # U)*
shows *complete-development (t \ u) U*
⟨*proof*⟩

lemma *complete-development-cong*:
shows ⟦*complete-development t U*; ¬ Λ.*Ide t*⟧ ⟹ [*t*] *∼* *U*
⟨*proof*⟩

lemma *complete-developments-cong*:
assumes ¬ Λ.*Ide t* **and** *complete-development t U* **and** *complete-development t V*
shows *U* *∼* *V*
⟨*proof*⟩

lemma *Trgs-complete-development*:
shows ⟦*complete-development t U*; ¬ Λ.*Ide t*⟧ ⟹ *Trgs U* = {Λ.*Trg t*}
⟨*proof*⟩

Now that we know all developments are finite, it is easy to construct a complete development by an iterative process that at each stage contracts one of the remaining marked redexes at each stage. It is also possible to construct a complete development by structural induction without using the finite developments property, but it is more work to prove the correctness.

fun (**in** *lambda-calculus*) *bottom-up-redex*
where *bottom-up-redex* ♯ = ♯
| *bottom-up-redex «x»* = *«x»*
| *bottom-up-redex* **λ**[*M*] = **λ**[*bottom-up-redex M*]
| *bottom-up-redex* (*M* ∘ *N*) =
(*if* ¬ *Ide M then bottom-up-redex M* ∘ *Src N else M* ∘ *bottom-up-redex N*)
| *bottom-up-redex* (**λ**[*M*] • *N*) =
(*if* ¬ *Ide M then* **λ**[*bottom-up-redex M*] ∘ *Src N*
else if ¬ *Ide N then* **λ**[*M*] ∘ *bottom-up-redex N*
else **λ**[*M*] • *N*)

lemma (**in** *lambda-calculus*) *elementary-reduction-bottom-up-redex*:
shows ⟦*Arr t*; ¬ *Ide t*⟧ ⟹ *elementary-reduction* (*bottom-up-redex t*)
⟨*proof*⟩

lemma (**in** *lambda-calculus*) *subs-bottom-up-redex*:
shows *Arr t* ⟹ *bottom-up-redex t* ⊑ *t*
⟨*proof*⟩

function (*sequential*) *bottom-up-development*
where *bottom-up-development t* =
(*if* ¬ Λ.*Arr t* ∨ Λ.*Ide t then* []
else Λ.*bottom-up-redex t* # (*bottom-up-development* (*t* \ Λ.*bottom-up-redex t*)))
⟨*proof*⟩

termination *bottom-up-development*
⟨*proof*⟩

lemma *complete-development-bottom-up-development-ind*:
shows $[\![\Lambda.Arr\ t;\ length\ (bottom\text{-}up\text{-}development\ t) \leq n]\!]$
$\Longrightarrow complete\text{-}development\ t\ (bottom\text{-}up\text{-}development\ t)$
$\langle proof \rangle$

lemma *complete-development-bottom-up-development*:
assumes $\Lambda.Arr\ t$
shows $complete\text{-}development\ t\ (bottom\text{-}up\text{-}development\ t)$
$\langle proof \rangle$

end

3.5 Reduction Strategies

context *lambda-calculus*
begin

A *reduction strategy* is a function taking an identity term to an arrow having that identity as its source.

definition *reduction-strategy*
where $reduction\text{-}strategy\ f \longleftrightarrow (\forall t.\ Ide\ t \longrightarrow Coinitial\ (f\ t)\ t)$

The following defines the iterated application of a reduction strategy to an identity term.

fun *reduce*
where $reduce\ f\ a\ 0 = a$
$\mid reduce\ f\ a\ (Suc\ n) = reduce\ f\ (Trg\ (f\ a))\ n$

lemma *red-reduce*:
assumes $reduction\text{-}strategy\ f$
shows $Ide\ a \Longrightarrow red\ a\ (reduce\ f\ a\ n)$
$\langle proof \rangle$

A reduction strategy is *normalizing* if iterated application of it to a normalizable term eventually yields a normal form.

definition *normalizing-strategy*
where $normalizing\text{-}strategy\ f \longleftrightarrow (\forall a.\ normalizable\ a \longrightarrow (\exists n.\ NF\ (reduce\ f\ a\ n)))$

end

context *reduction-paths*
begin

The following function constructs the reduction path that results by iterating the application of a reduction strategy to a term.

fun *apply-strategy*
where $apply\text{-}strategy\ f\ a\ 0 = []$

| *apply-strategy f a* (*Suc n*) = *f a* # *apply-strategy f* (Λ.*Trg* (*f a*)) *n*

lemma *apply-strategy-gives-path-ind*:
assumes Λ.*reduction-strategy f*
shows ⟦Λ.*Ide a*; *n* > *0*⟧ ⟹ *Arr* (*apply-strategy f a n*) ∧
length (*apply-strategy f a n*) = *n* ∧
Src (*apply-strategy f a n*) = *a* ∧
Trg (*apply-strategy f a n*) = Λ.*reduce f a n*
⟨*proof*⟩

lemma *apply-strategy-gives-path*:
assumes Λ.*reduction-strategy f* **and** Λ.*Ide a* **and** *n* > *0*
shows *Arr* (*apply-strategy f a n*)
and *length* (*apply-strategy f a n*) = *n*
and *Src* (*apply-strategy f a n*) = *a*
and *Trg* (*apply-strategy f a n*) = Λ.*reduce f a n*
⟨*proof*⟩

lemma *reduce-eq-Trg-apply-strategy*:
assumes Λ.*reduction-strategy S* **and** Λ.*Ide a*
shows *n* > *0* ⟹ Λ.*reduce S a n* = *Trg* (*apply-strategy S a n*)
⟨*proof*⟩

end

3.5.1 Parallel Reduction

context *lambda-calculus*
begin

Parallel reduction is the strategy that contracts all available redexes at each step.

fun *parallel-strategy*
where *parallel-strategy* «*i*» = «*i*»
| *parallel-strategy* **λ**[*t*] = **λ**[*parallel-strategy t*]
| *parallel-strategy* (**λ**[*t*] ∘ *u*) = **λ**[*parallel-strategy t*] • *parallel-strategy u*
| *parallel-strategy* (*t* ∘ *u*) = *parallel-strategy t* ∘ *parallel-strategy u*
| *parallel-strategy* (**λ**[*t*] • *u*) = **λ**[*parallel-strategy t*] • *parallel-strategy u*
| *parallel-strategy* ♯ = ♯

lemma *parallel-strategy-is-reduction-strategy*:
shows *reduction-strategy parallel-strategy*
⟨*proof*⟩

lemma *parallel-strategy-Src-eq*:
shows *Arr t* ⟹ *parallel-strategy* (*Src t*) = *parallel-strategy t*
⟨*proof*⟩

lemma *subs-parallel-strategy-Src*:
shows *Arr t* ⟹ *t* ⊑ *parallel-strategy* (*Src t*)
⟨*proof*⟩

end

context *reduction-paths*
begin

Parallel reduction is a universal strategy in the sense that every reduction path is $^*\lesssim^*$-below the path generated by the parallel reduction strategy.

lemma *parallel-strategy-is-universal*:
shows $[\![n > 0;\ n \leq \mathit{length}\ U;\ \mathit{Arr}\ U]\!]$
$\Longrightarrow \mathit{take}\ n\ U\ {}^*\lesssim^*\ \mathit{apply\text{-}strategy}\ \Lambda.\mathit{parallel\text{-}strategy}\ (\mathit{Src}\ U)\ n$
⟨*proof*⟩

end

context *lambda-calculus*
begin

Parallel reduction is a normalizing strategy.

lemma *parallel-strategy-is-normalizing*:
shows *normalizing-strategy parallel-strategy*
⟨*proof*⟩

An alternative characterization of a normal form is a term on which the parallel reduction strategy yields an identity.

abbreviation *has-redex*
where $\mathit{has\text{-}redex}\ t \equiv \mathit{Arr}\ t \wedge \neg\ \mathit{Ide}\ (\mathit{parallel\text{-}strategy}\ t)$

lemma *NF-iff-has-no-redex*:
shows $\mathit{Arr}\ t \Longrightarrow \mathit{NF}\ t \longleftrightarrow \neg\ \mathit{has\text{-}redex}\ t$
⟨*proof*⟩

lemma (**in** *lambda-calculus*) *not-NF-elim*:
assumes $\neg\ \mathit{NF}\ t$ **and** *Ide t*
obtains *u* **where** $\mathit{coinitial}\ t\ u \wedge \neg\ \mathit{Ide}\ u$
⟨*proof*⟩

lemma (**in** *lambda-calculus*) *NF-Lam-iff*:
shows $\mathit{NF}\ \boldsymbol{\lambda}[t] \longleftrightarrow \mathit{NF}\ t$
⟨*proof*⟩

lemma (**in** *lambda-calculus*) *NF-App-iff*:
shows $\mathit{NF}\ (t1 \circ t2) \longleftrightarrow \neg\ \mathit{is\text{-}Lam}\ t1 \wedge \mathit{NF}\ t1 \wedge \mathit{NF}\ t2$
⟨*proof*⟩

3.5.2 Head Reduction

Head reduction is the strategy that only contracts a redex at the "head" position, which is found at the end of the "left spine" of applications, and does nothing if there is no such redex.

The following function applies to an arbitrary arrow t, and it marks the redex at the head position, if any, otherwise it yields *Src t*.

fun *head-strategy*
where *head-strategy* «*i*» = «*i*»
| *head-strategy* $\boldsymbol{\lambda}[t]$ = $\boldsymbol{\lambda}$[*head-strategy t*]
| *head-strategy* ($\boldsymbol{\lambda}[t] \circ u$) = $\boldsymbol{\lambda}$[*Src t*] $\bullet$ *Src u*
| *head-strategy* ($t \circ u$) = *head-strategy t* $\circ$ *Src u*
| *head-strategy* ($\boldsymbol{\lambda}[t] \bullet u$) = $\boldsymbol{\lambda}$[*Src t*] $\bullet$ *Src u*
| *head-strategy* $\sharp$ = $\sharp$

lemma *Arr-head-strategy*:
shows *Arr t* $\Longrightarrow$ *Arr* (*head-strategy t*)
⟨*proof*⟩

lemma *Src-head-strategy*:
shows *Arr t* $\Longrightarrow$ *Src* (*head-strategy t*) = *Src t*
⟨*proof*⟩

lemma *Con-head-strategy*:
shows *Arr t* $\Longrightarrow$ *Con t* (*head-strategy t*)
⟨*proof*⟩

lemma *head-strategy-Src*:
shows *Arr t* $\Longrightarrow$ *head-strategy* (*Src t*) = *head-strategy t*
⟨*proof*⟩

lemma *head-strategy-is-elementary*:
shows $[\![$*Arr t*; $\neg$ *Ide* (*head-strategy t*)$]\!]$ $\Longrightarrow$ *elementary-reduction* (*head-strategy t*)
⟨*proof*⟩

lemma *head-strategy-is-reduction-strategy*:
shows *reduction-strategy head-strategy*
⟨*proof*⟩

The following function tests whether a term is an elementary reduction of the head redex.

fun *is-head-reduction*
where *is-head-reduction* «-» $\longleftrightarrow$ *False*
| *is-head-reduction* $\boldsymbol{\lambda}[t]$ $\longleftrightarrow$ *is-head-reduction t*
| *is-head-reduction* ($\boldsymbol{\lambda}$[-] $\circ$ -) $\longleftrightarrow$ *False*
| *is-head-reduction* ($t \circ u$) $\longleftrightarrow$ *is-head-reduction t* $\wedge$ *Ide u*
| *is-head-reduction* ($\boldsymbol{\lambda}[t] \bullet u$) $\longleftrightarrow$ *Ide t* $\wedge$ *Ide u*
| *is-head-reduction* $\sharp$ $\longleftrightarrow$ *False*

lemma *is-head-reduction-char*:
shows *is-head-reduction t* $\longleftrightarrow$ *elementary-reduction t* $\wedge$ *head-strategy* (*Src t*) = *t*
⟨*proof*⟩

lemma *is-head-reductionI*:
assumes *Arr t* **and** *elementary-reduction t* **and** *head-strategy* (*Src t*) = *t*
shows *is-head-reduction t*
⟨*proof*⟩

The following function tests whether a redex in the head position of a term is marked.

fun *contains-head-reduction*
where *contains-head-reduction* «-» ⟷ *False*
| *contains-head-reduction* $\boldsymbol{\lambda}[t]$ ⟷ *contains-head-reduction t*
| *contains-head-reduction* ($\boldsymbol{\lambda}$[-] ∘ -) ⟷ *False*
| *contains-head-reduction* (*t* ∘ *u*) ⟷ *contains-head-reduction t* ∧ *Arr u*
| *contains-head-reduction* ($\boldsymbol{\lambda}[t]$ • *u*) ⟷ *Arr t* ∧ *Arr u*
| *contains-head-reduction* ♯ ⟷ *False*

lemma *is-head-reduction-imp-contains-head-reduction*:
shows *is-head-reduction t* ⟹ *contains-head-reduction t*
⟨*proof*⟩

An *internal reduction* is one that does not contract any redex at the head position.

fun *is-internal-reduction*
where *is-internal-reduction* «-» ⟷ *True*
| *is-internal-reduction* $\boldsymbol{\lambda}[t]$ ⟷ *is-internal-reduction t*
| *is-internal-reduction* ($\boldsymbol{\lambda}[t]$ ∘ *u*) ⟷ *Arr t* ∧ *Arr u*
| *is-internal-reduction* (*t* ∘ *u*) ⟷ *is-internal-reduction t* ∧ *Arr u*
| *is-internal-reduction* ($\boldsymbol{\lambda}$[-] • -) ⟷ *False*
| *is-internal-reduction* ♯ ⟷ *False*

lemma *is-internal-reduction-iff*:
shows *is-internal-reduction t* ⟷ *Arr t* ∧ ¬ *contains-head-reduction t*
⟨*proof*⟩

Head reduction steps are either $\lesssim$-prefixes of, or are preserved by, residuation along arbitrary reductions.

lemma *is-head-reduction-resid*:
shows ⟦*is-head-reduction t*; *Arr u*; *Src t* = *Src u*⟧ ⟹ $t \lesssim u$ ∨ *is-head-reduction* (*t* \ *u*)
⟨*proof*⟩

Internal reductions are closed under residuation.

lemma *is-internal-reduction-resid*:
shows ⟦*is-internal-reduction t*; *is-internal-reduction u*; *Src t* = *Src u*⟧
⟹ *is-internal-reduction* (*t* \ *u*)
⟨*proof*⟩

A head reduction is preserved by residuation along an internal reduction, so a head reduction can only be canceled by a transition that contains a head reduction.

lemma *is-head-reduction-resid′*:
shows ⟦*is-head-reduction t*; *is-internal-reduction u*; *Src t* = *Src u*⟧
⟹ *is-head-reduction* (*t* \ *u*)
⟨*proof*⟩

The following function differs from *head-strategy* in that it only selects an already-marked redex, whereas *head-strategy* marks the redex at the head position.

fun *head-redex*
where *head-redex* $\sharp = \sharp$
  | *head-redex* «x» = «x»
  | *head-redex* $\boldsymbol{\lambda}[t] = \boldsymbol{\lambda}[\textit{head-redex } t]$
  | *head-redex* $(\boldsymbol{\lambda}[t] \circ u) = \boldsymbol{\lambda}[\textit{Src } t] \circ \textit{Src } u$
  | *head-redex* $(t \circ u) = \textit{head-redex } t \circ \textit{Src } u$
  | *head-redex* $(\boldsymbol{\lambda}[t] \bullet u) = (\boldsymbol{\lambda}[\textit{Src } t] \bullet \textit{Src } u)$

lemma *elementary-reduction-head-redex*:
shows $[\![\textit{Arr } t; \neg \textit{ Ide } (\textit{head-redex } t)]\!] \Longrightarrow \textit{elementary-reduction } (\textit{head-redex } t)$
  ⟨*proof*⟩

lemma *subs-head-redex*:
shows $\textit{Arr } t \Longrightarrow \textit{head-redex } t \sqsubseteq t$
  ⟨*proof*⟩

lemma *contains-head-reduction-iff*:
shows $\textit{contains-head-reduction } t \longleftrightarrow \textit{Arr } t \wedge \neg \textit{ Ide } (\textit{head-redex } t)$
  ⟨*proof*⟩

lemma *head-redex-is-head-reduction*:
shows $[\![\textit{Arr } t; \textit{contains-head-reduction } t]\!] \Longrightarrow \textit{is-head-reduction } (\textit{head-redex } t)$
  ⟨*proof*⟩

lemma *Arr-head-redex*:
assumes *Arr t*
shows *Arr* (*head-redex t*)
  ⟨*proof*⟩

lemma *Src-head-redex*:
assumes *Arr t*
shows $\textit{Src } (\textit{head-redex } t) = \textit{Src } t$
  ⟨*proof*⟩

lemma *Con-Arr-head-redex*:
assumes *Arr t*
shows *Con t* (*head-redex t*)
  ⟨*proof*⟩

lemma *is-head-reduction-if*:
shows $[\![\textit{contains-head-reduction } u; \textit{elementary-reduction } u]\!] \Longrightarrow \textit{is-head-reduction } u$
  ⟨*proof*⟩

lemma (**in** *reduction-paths*) *head-redex-decomp*:
assumes $\Lambda.\textit{Arr } t$
shows $[\Lambda.\textit{head-redex } t] @ [t \setminus \Lambda.\textit{head-redex } t] \ {}^*{\sim}^* \ [t]$
  ⟨*proof*⟩

An internal reduction cannot create a new head redex.

lemma *internal-reduction-preserves-no-head-redex*:
shows $[\![$*is-internal-reduction* u; *Ide* (*head-strategy* (*Src* u))$]\!]$
$\Longrightarrow$ *Ide* (*head-strategy* (*Trg* u))
⟨*proof*⟩

lemma *head-reduction-unique*:
shows $[\![$*is-head-reduction* t; *is-head-reduction* u; *coinitial* t $u]\!] \Longrightarrow t = u$
⟨*proof*⟩

Residuation along internal reductions preserves head reductions.

lemma *resid-head-strategy-internal*:
shows *is-internal-reduction* $u \Longrightarrow$ *head-strategy* (*Src* u) $\backslash$ u = *head-strategy* (*Trg* u)
⟨*proof*⟩

An internal reduction followed by a head reduction can be expressed as a join of the internal reduction with a head reduction.

lemma *resid-head-strategy-Src*:
assumes *is-internal-reduction* t **and** *is-head-reduction* u
and *seq* t u
shows *head-strategy* (*Src* t) $\backslash$ $t = u$
and *composite-of* t u (*Join* (*head-strategy* (*Src* t)) t)
⟨*proof*⟩

lemma *App-Var-contains-no-head-reduction*:
shows $\neg$ *contains-head-reduction* («x» $\circ$ u)
⟨*proof*⟩

lemma *hgt-resid-App-head-redex*:
assumes *Arr* ($t \circ u$) **and** $\neg$ *Ide* (*head-redex* ($t \circ u$))
shows *hgt* (($t \circ u$) $\backslash$ *head-redex* ($t \circ u$)) < *hgt* ($t \circ u$)
⟨*proof*⟩

3.5.3 Leftmost Reduction

Leftmost (or normal-order) reduction is the strategy that produces an elementary reduction path by contracting the leftmost redex at each step. It agrees with head reduction as long as there is a head redex, otherwise it continues on with the next subterm to the right.

fun *leftmost-strategy*
where *leftmost-strategy* «x» = «x»
| *leftmost-strategy* $\boldsymbol{\lambda}[t]$ = $\boldsymbol{\lambda}$[*leftmost-strategy* t]
| *leftmost-strategy* ($\boldsymbol{\lambda}[t] \circ u$) = $\boldsymbol{\lambda}[t] \bullet u$
| *leftmost-strategy* ($t \circ u$) =
(*if* $\neg$ *Ide* (*leftmost-strategy* t)
then *leftmost-strategy* $t \circ u$
else $t \circ$ *leftmost-strategy* u)
| *leftmost-strategy* ($\boldsymbol{\lambda}[t] \bullet u$) = $\boldsymbol{\lambda}[t] \bullet u$

| *leftmost-strategy* $\sharp = \sharp$

definition *is-leftmost-reduction*
where *is-leftmost-reduction* $t \longleftrightarrow$ *elementary-reduction* $t \wedge$ *leftmost-strategy* (*Src* t) $= t$

lemma *leftmost-strategy-is-reduction-strategy*:
shows *reduction-strategy leftmost-strategy*
⟨*proof*⟩

lemma *elementary-reduction-leftmost-strategy*:
shows *Ide* $t \Longrightarrow$ *elementary-reduction* (*leftmost-strategy* t) $\vee$ *Ide* (*leftmost-strategy* t)
⟨*proof*⟩

lemma (**in** *lambda-calculus*) *leftmost-strategy-selects-head-reduction*:
shows *is-head-reduction* $t \Longrightarrow t =$ *leftmost-strategy* (*Src* t)
⟨*proof*⟩

lemma *has-redex-iff-not-Ide-leftmost-strategy*:
shows *Arr* $t \Longrightarrow$ *has-redex* $t \longleftrightarrow \neg$ *Ide* (*leftmost-strategy* (*Src* t))
⟨*proof*⟩

lemma *leftmost-reduction-preservation*:
shows ⟦*is-leftmost-reduction* t; *elementary-reduction* u; $\neg$ *is-leftmost-reduction* u;
coinitial t u⟧ $\Longrightarrow$ *is-leftmost-reduction* ($t \setminus u$)
⟨*proof*⟩

end

3.6 Standard Reductions

In this section, we define the notion of a *standard reduction*, which is an elementary reduction path that performs reductions from left to right, possibly skipping some redexes that could be contracted. Once a redex has been skipped, neither that redex nor any redex to its left will subsequently be contracted. We then define and prove correct a function that transforms an arbitrary elementary reduction path into a congruent standard reduction path. Using this function, we prove the Standardization Theorem, which says that every elementary reduction path is congruent to a standard reduction path. We then show that a standard reduction path that reaches a normal form is in fact a leftmost reduction path. From this fact and the Standardization Theorem we prove the Leftmost Reduction Theorem: leftmost reduction is a normalizing strategy.

The Standardization Theorem was first proved by Curry and Feys [3], with subsequent proofs given by a number of authors. Formalized proofs have also been given; a recent one (using Agda) is presented in [2], with references to earlier work. The version of the theorem that we formalize here is a “strong” version, which asserts the existence of a standard reduction path congruent to a a given elementary reduction path. At the core of the proof is a function that directly transforms a given reduction path into a standard one, using an algorithm roughly analogous to insertion sort. The Finite Development Theorem is used in the proof of termination. The proof of correctness is long, due to the number of cases that have to be considered, but the use of a proof assistant makes this manageable.

3.6.1 Standard Reduction Paths

'Standardly Sequential' Reductions

We first need to define the notion of a "standard reduction". In contrast to what is typically done by other authors, we define this notion by direct comparison of adjacent terms in an elementary reduction path, rather than by using devices such as a numbering of subterms from left to right.

The following function decides when two terms t and u are elementary reductions that are "standardly sequential". This means that t and u are sequential, but in addition no marked redex in u is the residual of an (unmarked) redex "to the left of" any marked redex in t. Some care is required to make sure that the recursive definition captures what we intend. Most of the clauses are readily understandable. One clause that perhaps could use some explanation is the one for *sseq* $((\boldsymbol{\lambda}[t] \bullet u) \circ v)\ w$. Referring to the previously proved fact *seq-cases*, which classifies the way in which two terms t and u can be sequential, we see that one case that must be covered is when t has the form $\boldsymbol{\lambda}[t] \bullet v) \circ w$ and the top-level constructor of u is *Beta*. In this case, it is the reduction of t that creates the top-level redex contracted in u, so it is impossible for u to be a residual of a redex that already exists in *Src t*.

context *lambda-calculus*
begin

fun *sseq*
where *sseq* - ♯ = *False*
| *sseq* «-» «-» = *False*
| *sseq* $\boldsymbol{\lambda}[t]$ $\boldsymbol{\lambda}[t']$ = *sseq t t'*
| *sseq* $(t \circ u)$ $(t' \circ u')$ =
((*sseq t t'* ∧ *Ide u* ∧ $u = u'$) ∨
(*Ide t* ∧ $t = t'$ ∧ *sseq u u'*) ∨
(*elementary-reduction t* ∧ *Trg t* = t' ∧
(u = *Src u'* ∧ *elementary-reduction u'*)))
| *sseq* $(\boldsymbol{\lambda}[t] \circ u)$ $(\boldsymbol{\lambda}[t'] \bullet u')$ = *False*
| *sseq* $((\boldsymbol{\lambda}[t] \bullet u) \circ v)$ w =
(*Ide t* ∧ *Ide u* ∧ *Ide v* ∧ *elementary-reduction w* ∧ *seq* $((\boldsymbol{\lambda}[t] \bullet u) \circ v)$ w)
| *sseq* $(\boldsymbol{\lambda}[t] \bullet u)$ v = (*Ide t* ∧ *Ide u* ∧ *elementary-reduction v* ∧ *seq* $(\boldsymbol{\lambda}[t] \bullet u)$ v)
| *sseq* - - = *False*

lemma *sseq-imp-seq*:
shows *sseq t u* ⟹ *seq t u*
⟨*proof*⟩

lemma *sseq-imp-elementary-reduction1*:

shows *sseq t u* $\Longrightarrow$ *elementary-reduction t*
⟨*proof*⟩

lemma *sseq-imp-elementary-reduction2*:
shows *sseq t u* $\Longrightarrow$ *elementary-reduction u*
⟨*proof*⟩

lemma *sseq-Beta*:
shows *sseq* $(\boldsymbol{\lambda}[t] \bullet u)$ *v* $\longleftrightarrow$ *Ide t* $\wedge$ *Ide u* $\wedge$ *elementary-reduction v* $\wedge$ *seq* $(\boldsymbol{\lambda}[t] \bullet u)$ *v*
⟨*proof*⟩

lemma *sseq-BetaI* [*intro*]:
assumes *Ide t* **and** *Ide u* **and** *elementary-reduction v* **and** *seq* $(\boldsymbol{\lambda}[t] \bullet u)$ *v*
shows *sseq* $(\boldsymbol{\lambda}[t] \bullet u)$ *v*
⟨*proof*⟩

A head reduction is standardly sequential with any elementary reduction that can be performed after it.

lemma *sseq-head-reductionI*:
shows ⟦*is-head-reduction t*; *elementary-reduction u*; *seq t u*⟧ $\Longrightarrow$ *sseq t u*
⟨*proof*⟩

Once a head reduction is skipped in an application, then all terms that follow it in a standard reduction path are also applications that do not contain head reductions.

lemma *sseq-preserves-App-and-no-head-reduction*:
shows ⟦*sseq t u*; *is-App t* $\wedge$ $\neg$ *contains-head-reduction t*⟧
$\Longrightarrow$ *is-App u* $\wedge$ $\neg$ *contains-head-reduction u*
⟨*proof*⟩

end

Standard Reduction Paths

context *reduction-paths*
begin

A *standard reduction path* is an elementary reduction path in which successive reductions are standardly sequential.

fun *Std*
where *Std* [] = *True*
| *Std* [*t*] = Λ.*elementary-reduction t*
| *Std* (*t* # *U*) = (Λ.*sseq t* (*hd U*) $\wedge$ *Std U*)

lemma *Std-consE* [*elim*]:
assumes *Std* (*t* # *U*)
and ⟦Λ.*Arr t*; *U* $\neq$ [] $\Longrightarrow$ Λ.*sseq t* (*hd U*); *Std U*⟧ $\Longrightarrow$ *thesis*
shows *thesis*
⟨*proof*⟩

lemma *Std-imp-Arr* [*simp*]:
shows $[\![Std\ T;\ T \neq [\,]]\!] \Longrightarrow Arr\ T$
⟨*proof*⟩

lemma *Std-imp-sseq-last-hd*:
shows $[\![Std\ (T\ @\ U);\ T \neq [\,];\ U \neq [\,]]\!] \Longrightarrow \Lambda.sseq\ (last\ T)\ (hd\ U)$
⟨*proof*⟩

lemma *Std-implies-set-subset-elementary-reduction*:
shows $Std\ U \Longrightarrow set\ U \subseteq Collect\ \Lambda.elementary\text{-}reduction$
⟨*proof*⟩

lemma *Std-map-Lam*:
shows $Std\ T \Longrightarrow Std\ (map\ \Lambda.Lam\ T)$
⟨*proof*⟩

lemma *Std-map-App1*:
shows $[\![\Lambda.Ide\ b;\ Std\ T]\!] \Longrightarrow Std\ (map\ (\lambda X.\ X \circ b)\ T)$
⟨*proof*⟩

lemma *Std-map-App2*:
shows $[\![\Lambda.Ide\ a;\ Std\ T]\!] \Longrightarrow Std\ (map\ (\lambda u.\ a \circ u)\ T)$
⟨*proof*⟩

lemma *Std-map-un-Lam*:
shows $[\![Std\ T;\ set\ T \subseteq Collect\ \Lambda.is\text{-}Lam]\!] \Longrightarrow Std\ (map\ \Lambda.un\text{-}Lam\ T)$
⟨*proof*⟩

lemma *Std-append-single*:
shows $[\![Std\ T;\ T \neq [\,];\ \Lambda.sseq\ (last\ T)\ u]\!] \Longrightarrow Std\ (T\ @\ [u])$
⟨*proof*⟩

lemma *Std-append*:
shows $[\![Std\ T;\ Std\ U;\ T = [\,] \lor U = [\,] \lor \Lambda.sseq\ (last\ T)\ (hd\ U)]\!] \Longrightarrow Std\ (T\ @\ U)$
⟨*proof*⟩

Projections of Standard 'App Paths'

Given a standard reduction path, all of whose transitions have App as their top-level constructor, we can apply *un-App1* or *un-App2* to each transition to project the path onto paths formed from the "rator" and the "rand" of each application. These projected paths are not standard, since the projection operation will introduce identities, in general. However, in this section we show that if we remove the identities, then in fact we do obtain standard reduction paths.

abbreviation *notIde*
where $notIde \equiv \lambda u.\ \neg\ \Lambda.Ide\ u$

lemma *filter-notIde-Ide*:

shows *Ide U* $\Longrightarrow$ *filter notIde U* = []
  ⟨*proof*⟩

lemma *cong-filter-notIde*:
shows ⟦*Arr U*; ¬ *Ide U*⟧ $\Longrightarrow$ *filter notIde U* $^*\!\sim^*$ *U*
⟨*proof*⟩

lemma *Std-filter-map-un-App1*:
shows ⟦*Std U*; *set U* ⊆ *Collect* Λ.*is-App*⟧ $\Longrightarrow$ *Std* (*filter notIde* (*map* Λ.*un-App1 U*))
⟨*proof*⟩

lemma *Std-filter-map-un-App2*:
shows ⟦*Std U*; *set U* ⊆ *Collect* Λ.*is-App*⟧ $\Longrightarrow$ *Std* (*filter notIde* (*map* Λ.*un-App2 U*))
⟨*proof*⟩

If the first step in a standard reduction path contracts a redex that is not at the head position, then all subsequent terms have *App* as their top-level operator.

lemma *seq-App-Std-implies*:
shows ⟦*Std* (*t* # *U*); Λ.*is-App t* ∧ ¬ Λ.*contains-head-reduction t*⟧
  $\Longrightarrow$ *set U* ⊆ *Collect* Λ.*is-App*
⟨*proof*⟩

3.6.2 Standard Developments

The following function takes a term t (representing a parallel reduction) and produces a standard reduction path that is a complete development of t and is thus congruent to $[t]$. The proof of termination makes use of the Finite Development Theorem.

function (*sequential*) *standard-development*
where *standard-development* ♯ = []
  | *standard-development* «-» = []
  | *standard-development* **λ**[*t*] = *map* Λ.*Lam* (*standard-development t*)
  | *standard-development* (*t* ∘ *u*) =
    (*if* Λ.*Arr t* ∧ Λ.*Arr u then*
      map (λ*v*. *v* ∘ Λ.*Src u*) (*standard-development t*) @
      map (λ*v*. Λ.*Trg t* ∘ *v*) (*standard-development u*)
    else [])
  | *standard-development* (**λ**[*t*] • *u*) =
    (*if* Λ.*Arr t* ∧ Λ.*Arr u then*
      (**λ**[Λ.*Src t*] • Λ.*Src u*) # *standard-development* (Λ.*subst u t*)
    else [])
  ⟨*proof*⟩

abbreviation (**in** *lambda-calculus*) *stddev-term-rel*
where *stddev-term-rel* ≡ *mlex-prod hgt subterm-rel*

lemma (**in** *lambda-calculus*) *subst-lt-Beta*:
assumes *Arr t* **and** *Arr u*
shows (*subst u t*, **λ**[*t*] • *u*) ∈ *stddev-term-rel*
⟨*proof*⟩

termination *standard-development*
⟨*proof*⟩

lemma *Ide-iff-standard-development-empty*:
shows $\Lambda.Arr\ t \Longrightarrow \Lambda.Ide\ t \longleftrightarrow standard\text{-}development\ t = []$
⟨*proof*⟩

lemma *set-standard-development*:
shows $\Lambda.Arr\ t \longrightarrow set\ (standard\text{-}development\ t) \subseteq Collect\ \Lambda.elementary\text{-}reduction$
⟨*proof*⟩

lemma *cong-standard-development*:
shows $\Lambda.Arr\ t \wedge \neg\ \Lambda.Ide\ t \longrightarrow standard\text{-}development\ t\ ^*\!\sim^*\ [t]$
⟨*proof*⟩

lemma *Src-hd-standard-development*:
assumes $\Lambda.Arr\ t$ **and** $\neg\ \Lambda.Ide\ t$
shows $\Lambda.Src\ (hd\ (standard\text{-}development\ t)) = \Lambda.Src\ t$
⟨*proof*⟩

lemma *Trg-last-standard-development*:
assumes $\Lambda.Arr\ t$ **and** $\neg\ \Lambda.Ide\ t$
shows $\Lambda.Trg\ (last\ (standard\text{-}development\ t)) = \Lambda.Trg\ t$
⟨*proof*⟩

lemma *Srcs-standard-development*:
shows $[\![\Lambda.Arr\ t;\ standard\text{-}development\ t \neq []]\!]$
$\Longrightarrow Srcs\ (standard\text{-}development\ t) = \{\Lambda.Src\ t\}$
⟨*proof*⟩

lemma *Trgs-standard-development*:
shows $[\![\Lambda.Arr\ t;\ standard\text{-}development\ t \neq []]\!]$
$\Longrightarrow Trgs\ (standard\text{-}development\ t) = \{\Lambda.Trg\ t\}$
⟨*proof*⟩

lemma *development-standard-development*:
shows $\Lambda.Arr\ t \longrightarrow development\ t\ (standard\text{-}development\ t)$
⟨*proof*⟩

lemma *Std-standard-development*:
shows $Std\ (standard\text{-}development\ t)$
⟨*proof*⟩

3.6.3 Standardization

In this section, we define and prove correct a function that takes an arbitrary reduction path and produces a standard reduction path congruent to it. The method is roughly analogous to insertion sort: given a path, recursively standardize the tail and then "insert" the head into to the result. A complication is that in general the head may be a parallel reduction instead of an elementary reduction, and in any case elementary reductions are not preserved under residuation so we need to be able to handle the parallel reductions that arise from permuting elementary reductions. In general, this means that parallel reduction steps have to be decomposed into factors, and then each factor has to be inserted at its proper position. Another issue is that reductions don't all happen at the top level of a term, so we need to be able to descend recursively into terms during the insertion procedure. The key idea here is: in a standard reduction, once a step has occurred that is not a head reduction, then all subsequent terms will have *App* as their top-level constructor. So, once we have passed a step that is not a head reduction, we can recursively descend into the subsequent applications and treat the "rator" and the "rand" parts independently.

The following function performs the core insertion part of the standardization algorithm. It assumes that it is given an arbitrary parallel reduction t and an already-standard reduction path U, and it inserts t into U, producing a standard reduction path that is congruent to t # U. A somewhat elaborate case analysis is required to determine whether t needs to be factored and whether part of it might need to be permuted with the head of U. The recursion is complicated by the need to make sure that the second argument U is always a standard reduction path. This is so that it is possible to decide when the rest of the steps will be applications and it is therefore possible to recurse into them. This constrains what recursive calls we can make, since we are not able to make a recursive call in which an identity has been prepended to U. Also, if t # U consists completely of identities, then its standardization is the empty list [], which is not a path and cannot be congruent to t # U. So in order to be able to apply the induction hypotheses in the correctness proof, we need to make sure that we don't make recursive calls when U itself would consist entirely of identities. Finally, when we descend through an application, the step t and the path U are projected to their "rator" and "rand" components, which are treated separately and the results concatenated. However, the projection operations can introduce identities and therefore do not preserve elementary reductions. To handle this, we need to filter out identities after projection but before the recursive call.

Ensuring termination also involves some care: we make recursive calls in which the length of the second argument is increased, but the "height" of the first argument is decreased. So we use a lexicographic order that makes the height of the first argument more significant and the length of the second argument secondary. The base cases either discard paths that consist entirely of identities, or else they expand a single parallel reduction t into a standard development.

function (*sequential*) *stdz-insert*
where *stdz-insert t* [] = *standard-development t*
  | *stdz-insert «-» U* = *stdz-insert* (*hd U*) (*tl U*)
  | *stdz-insert* **λ**[*t*] *U* =
    (*if* Λ.*Ide t then*
      stdz-insert (*hd U*) (*tl U*)
    else

$map\ \Lambda.Lam\ (stdz\text{-}insert\ t\ (map\ \Lambda.un\text{-}Lam\ U)))$
| $stdz\text{-}insert\ (\boldsymbol{\lambda}[t] \circ u)\ ((\boldsymbol{\lambda}[\text{-}] \bullet \text{-}) \# U) = stdz\text{-}insert\ (\boldsymbol{\lambda}[t] \bullet u)\ U$
| $stdz\text{-}insert\ (t \circ u)\ U =$
(*if* $\Lambda.Ide\ (t \circ u)$ *then*
$stdz\text{-}insert\ (hd\ U)\ (tl\ U)$
else if $\Lambda.seq\ (t \circ u)\ (hd\ U)$ *then*
if $\Lambda.contains\text{-}head\text{-}reduction\ (t \circ u)$ *then*
if $\Lambda.Ide\ ((t \circ u) \setminus \Lambda.head\text{-}redex\ (t \circ u))$ *then*
$\Lambda.head\text{-}redex\ (t \circ u) \# stdz\text{-}insert\ (hd\ U)\ (tl\ U)$
else
$\Lambda.head\text{-}redex\ (t \circ u) \# stdz\text{-}insert\ ((t \circ u) \setminus \Lambda.head\text{-}redex\ (t \circ u))\ U$
else if $\Lambda.contains\text{-}head\text{-}reduction\ (hd\ U)$ *then*
if $\Lambda.Ide\ ((t \circ u) \setminus \Lambda.head\text{-}strategy\ (t \circ u))$ *then*
$stdz\text{-}insert\ (\Lambda.head\text{-}strategy\ (t \circ u))\ (tl\ U)$
else
$\Lambda.head\text{-}strategy\ (t \circ u) \# stdz\text{-}insert\ ((t \circ u) \setminus \Lambda.head\text{-}strategy\ (t \circ u))\ (tl\ U)$
else
$map\ (\lambda a.\ a \circ \Lambda.Src\ u)$
$(stdz\text{-}insert\ t\ (filter\ notIde\ (map\ \Lambda.un\text{-}App1\ U)))$ @
$map\ (\lambda b.\ \Lambda.Trg\ (\Lambda.un\text{-}App1\ (last\ U)) \circ b)$
$(stdz\text{-}insert\ u\ (filter\ notIde\ (map\ \Lambda.un\text{-}App2\ U)))$
else [])
| $stdz\text{-}insert\ (\boldsymbol{\lambda}[t] \bullet u)\ U =$
(*if* $\Lambda.Arr\ t \wedge \Lambda.Arr\ u$ *then*
$(\boldsymbol{\lambda}[\Lambda.Src\ t] \bullet \Lambda.Src\ u) \# stdz\text{-}insert\ (\Lambda.subst\ u\ t)\ U$
else [])
| $stdz\text{-}insert$ - - = []
⟨*proof*⟩

fun *standardize*
where *standardize* [] = []
| *standardize* U = $stdz\text{-}insert\ (hd\ U)\ (standardize\ (tl\ U))$

abbreviation *stdzins-rel*
where $stdzins\text{-}rel \equiv mlex\text{-}prod\ (length\ o\ snd)\ (inv\text{-}image\ (mlex\text{-}prod\ \Lambda.hgt\ \Lambda.subterm\text{-}rel)\ fst)$

termination *stdz-insert*
⟨*proof*⟩

lemma *stdz-insert-Ide*:
shows $Ide\ (t \# U) \Longrightarrow stdz\text{-}insert\ t\ U = []$
⟨*proof*⟩

lemma *stdz-insert-Ide-Std*:
shows $[\![\Lambda.Ide\ u;\ seq\ [u]\ U;\ Std\ U]\!] \Longrightarrow stdz\text{-}insert\ u\ U = stdz\text{-}insert\ (hd\ U)\ (tl\ U)$
⟨*proof*⟩

Insertion of a term with *Beta* as its top-level constructor always leaves such a term at the head of the result. Stated another way, *Beta* at the top-level must always come first in a standard reduction path.

lemma *stdz-insert-Beta-ind*:
shows $[\![\Lambda.\mathit{hgt}\ t + \mathit{length}\ U \leq n;\ \Lambda.\mathit{is\text{-}Beta}\ t;\ \mathit{seq}\ [t]\ U]\!]$
$\Longrightarrow \Lambda.\mathit{is\text{-}Beta}\ (\mathit{hd}\ (\mathit{stdz\text{-}insert}\ t\ U))$
⟨*proof*⟩

lemma *stdz-insert-Beta*:
assumes $\Lambda.\mathit{is\text{-}Beta}\ t$ **and** $\mathit{seq}\ [t]\ U$
shows $\Lambda.\mathit{is\text{-}Beta}\ (\mathit{hd}\ (\mathit{stdz\text{-}insert}\ t\ U))$
⟨*proof*⟩

This is the correctness lemma for insertion: Given a term t and standard reduction path U sequential with it, the result of insertion is a standard reduction path which is congruent to $t\ \#\ U$ unless $t\ \#\ U$ consists entirely of identities.

The proof is very long. Its structure parallels that of the definition of the function *stdz-insert*. For really understanding the details, I strongly suggest viewing the proof in Isabelle/JEdit and using the code folding feature to unfold the proof a little bit at a time.

lemma *stdz-insert-correctness*:
shows $\mathit{seq}\ [t]\ U \wedge \mathit{Std}\ U \longrightarrow$
$\mathit{Std}\ (\mathit{stdz\text{-}insert}\ t\ U) \wedge (\neg\ \mathit{Ide}\ (t\ \#\ U) \longrightarrow \mathit{cong}\ (\mathit{stdz\text{-}insert}\ t\ U)\ (t\ \#\ U))$
(**is** $?P\ t\ U$)
⟨*proof*⟩

The Standardization Theorem

Using the function *standardize*, we can now prove the Standardization Theorem. There is still a little bit more work to do, because we have to deal with various cases in which the reduction path to be standardized is empty or consists entirely of identities.

theorem *standardization-theorem*:
shows $\mathit{Arr}\ T \Longrightarrow \mathit{Std}\ (\mathit{standardize}\ T) \wedge (\mathit{Ide}\ T \longrightarrow \mathit{standardize}\ T = []) \wedge$
$(\neg\ \mathit{Ide}\ T \longrightarrow \mathit{cong}\ (\mathit{standardize}\ T)\ T)$
⟨*proof*⟩

The Leftmost Reduction Theorem

In this section we prove the Leftmost Reduction Theorem, which states that leftmost reduction is a normalizing strategy.

We first show that if a standard reduction path reaches a normal form, then the path must be the one produced by following the leftmost reduction strategy. This is because, in a standard reduction path, once a leftmost redex is skipped, all subsequent reductions occur "to the right of it", hence they are all non-leftmost reductions that do not contract the skipped redex, which remains in the leftmost position.

The Leftmost Reduction Theorem then follows from the Standardization Theorem. If a term is normalizable, there is a reduction path from that term to a normal form.

By the Standardization Theorem we may as well assume that path is standard. But a standard reduction path to a normal form is the path generated by following the leftmost reduction strategy, hence leftmost reduction reaches a normal form after a finite number of steps.

lemma *sseq-reflects-leftmost-reduction*:
assumes $\Lambda.$*sseq t u* **and** $\Lambda.$*is-leftmost-reduction u*
shows $\Lambda.$*is-leftmost-reduction t*
$\langle proof \rangle$

lemma *elementary-reduction-to-NF-is-leftmost*:
shows $[\![\Lambda.\mathit{elementary\text{-}reduction}\ t;\ \Lambda.\mathit{NF}\ (\mathit{Trg}\ [t])]\!] \Longrightarrow \Lambda.\mathit{leftmost\text{-}strategy}\ (\Lambda.\mathit{Src}\ t) = t$
$\langle proof \rangle$

lemma *Std-path-to-NF-is-leftmost*:
shows $[\![\mathit{Std}\ T;\ \Lambda.\mathit{NF}\ (\mathit{Trg}\ T)]\!] \Longrightarrow \mathit{set}\ T \subseteq \mathit{Collect}\ \Lambda.\mathit{is\text{-}leftmost\text{-}reduction}$
$\langle proof \rangle$

theorem *leftmost-reduction-theorem*:
shows $\Lambda.$*normalizing-strategy* $\Lambda.$*leftmost-strategy*
$\langle proof \rangle$

end

end

Bibliography

[1] H. Barendregt. *The Lambda-calculus: Its Syntax and Semantics*. North-Holland, 1984.

[2] M. Copes. A machine-checked proof of the standardization theorem in lambda calculus using multiple substitution. Master's thesis, Universidad ORT Uruguay, 2018. https://dspace.ort.edu.uy/bitstream/handle/20.500.11968/3725/Material%20completo.pdf.

[3] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

[4] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 34(5):381–392, 1972.

[5] R. de Vrijer. A direct proof of the finite developments theorem. *The Journal of Symbolic Logic*, 50(2):339–343, June 1985.

[6] R. Hindley. Reductions of residuals are finite. *Transactions of the American Mathematical Society*, 240:345–361, June 1978.

[7] G. Huet. Residual theory in λ-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.

[8] J.-J. Lévy. *Réductions correctes et optimales dans le λ-calcul*. PhD thesis, U. Paris VII, 1978. Thèse d'Etat.

[9] D. E. Schroer. *The Church-Rosser Theorem*. PhD thesis, Cornell University, 1965.

[10] E. W. Stark. Concurrent transition systems. *Theoretical Computer Science*, 64:221–269, July 1989.

[11] E. W. Stark. Category theory with adjunctions and limits. *Archive of Formal Proofs*, June 2016. http://isa-afp.org/entries/Category3.shtml, Formal proof development.