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 $n = null$
using *assms null-def ex-un-null the1-equality* [*of* $\lambda n.\ \forall t.\ OP\ n\ t = n \wedge OP\ t\ n = n$]
by *auto*

lemma *null-is-zero* [*simp*]:
shows $OP\ null\ t = null$ **and** $OP\ t\ null = null$
using *null-def ex-un-null theI'* [*of* $\lambda n.\ \forall t.\ OP\ n\ t = n \wedge OP\ t\ n = n$]
by *auto*

end

2.1.2 Residuation

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

type-synonym *'a resid* = *'a* ⇒ *'a* ⇒ *'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 $(v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$
using *cube-ax* **by** *metis*

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

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

lemma *conI* [*intro*]:
assumes $t \setminus u \neq null$
shows $t \frown u$
using *assms con-def* **by** *blast*

lemma *conE* [*elim*]:
assumes $t \frown u$
and $t \setminus u \neq null \Longrightarrow T$
shows *T*
using *assms con-def* **by** *simp*

lemma *con-sym*:
assumes $t \frown u$
shows $u \frown t$
using *assms con-def con-sym-ax* **by** *blast*

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

definition *arr*

where *arr t* ≡ *t* ⌢ *t*

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

lemma *arrE* [*elim*]:
assumes *arr t*
and *t* ⌢ *t* ⟹ *T*
shows *T*
  using *assms arr-def* **by** *simp*

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

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

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

lemma *arr-resid-iff-con*:
shows *arr* (*t* \ *u*) ⟷ *t* ⌢ *u*
  by *auto*

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

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

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

definition *trg*
where *trg t* ≡ *t* \ *t*

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

lemma *arr-trg-iff-arr*:
shows *arr* (*trg t*) ⟷ *arr t*
by (*metis arrI arrE arr-resid-iff-con resid-arr-self*)

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

definition *ide*
where *ide a* ≡ *a* ⌢ *a* ∧ *a* \ *a* = *a*

lemma *ideI* [*intro*]:
assumes *a* ⌢ *a* **and** *a* \ *a* = *a*
shows *ide a*
using *assms ide-def* **by** *auto*

lemma *ideE* [*elim*]:
assumes *ide a*
and ⟦*a* ⌢ *a*; *a* \ *a* = *a*⟧ ⟹ *T*
shows *T*
using *assms ide-def* **by** *blast*

lemma *ide-implies-arr* [*simp*]:
assumes *ide a*
shows *arr a*
using *assms* **by** *blast*

lemma *not-ide-null* [*simp*]:
shows *ide null* = *False*
by *auto*

end

2.1.3 Residuated Transition System

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

locale *rts* = *residuation* +
assumes *ide-trg* [*simp*]: *arr t* ⟹ *ide* (*trg t*)

and *resid-arr-ide*: $[\![ide\ a;\ t \frown a]\!] \Longrightarrow t \setminus a = t$
and *resid-ide-arr* [*simp*]: $[\![ide\ a;\ a \frown t]\!] \Longrightarrow ide\ (a \setminus t)$
and *con-imp-coinitial-ax*: $t \frown u \Longrightarrow \exists a.\ ide\ a \wedge a \frown t \wedge a \frown u$
and *con-target*: $[\![ide\ (t \setminus u);\ u \frown v]\!] \Longrightarrow t \setminus u \frown v \setminus u$
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 $a \in sources\ t$
using *assms sources-def* **by** *simp*

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

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

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

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

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

lemma *target-is-ide*:

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

Consistent arrows have a common source.

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

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

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

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

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

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

Residuation along an identity reflects identities.

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

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

lemma *con-transitive-on-ide*:
assumes $\mathit{ide}\ a$ **and** $\mathit{ide}\ b$ **and** $\mathit{ide}\ c$
shows $[\![a \frown b;\ b \frown c]\!] \Longrightarrow a \frown c$
using *assms*
by (*metis resid-arr-ide con-target con-sym*)

lemma *sources-are-con*:

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

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

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

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

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

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

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

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

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

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

lemma *coinitialE* [*elim*]:
assumes $\mathit{coinitial}\ t\ u$
and $[\![\mathit{arr}\ t;\ \mathit{arr}\ u;\ \mathit{sources}\ t = \mathit{sources}\ u]\!] \Longrightarrow T$

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

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

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

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

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

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

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

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

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

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

definition *seq*
where *seq t u* ≡ *arr t* ∧ *arr u* ∧ *targets t* = *sources u*

lemma *seqI* [*intro*]:
shows ⟦*arr t*; *targets t* = *sources u*⟧ ⟹ *seq t u*
and ⟦*arr u*; *targets t* = *sources u*⟧ ⟹ *seq t u*
using *seq-def arr-iff-has-source arr-iff-has-target* **by** *metis+*

lemma *seqE* [*elim*]:
assumes *seq t u*
and ⟦*arr t*; *arr u*; *targets t* = *sources u*⟧ ⟹ *T*
shows *T*
using *assms seq-def* **by** *blast*

Congruence of Transitions

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

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

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

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

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

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

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

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

abbreviation *cong* (**infix** ‹∼› *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
  using *assms prfx-implies-con* **by** *blast*

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

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

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

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

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

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

lemma *cong-implies-coterminal*:

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

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

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

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

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

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

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

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

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

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

Chosen Sources

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

definition *src*
where *src t* ≡ *if arr t then SOME a. a* ∈ *sources t else null*

lemma *src-in-sources*:
assumes *arr t*
shows *src t* ∈ *sources t*
using *assms someI-ex* [*of* λ*a. a* ∈ *sources t*] *arr-iff-has-source src-def*
by *auto*

lemma *src-congI*:
assumes *ide a* **and** *a* ⌢ *t*
shows *src t* ∼ *a*
using *assms src-in-sources sources-are-cong*
by (*metis arr-iff-has-source con-sym emptyE in-sourcesI*)

lemma *arr-src-iff-arr*:
shows *arr* (*src t*) ⟷ *arr t*
by (*metis arrI conE null-is-zero*(*2*) *sources-are-con arrE src-def src-in-sources*)

lemma *arr-src-if-arr* [*simp*]:
assumes *arr t*
shows *arr* (*src t*)
using *assms arr-src-iff-arr* **by** *blast*

lemma *sources-char*$_{CS}$:
shows *sources t* = {*a. arr t* ∧ *src t* ∼ *a*}
unfolding *sources-def*
by (*meson con-implies-arr*(*1*) *con-sym in-sourcesE sources-cong-closed*
src-congI src-in-sources)

lemma *targets-char*′:
shows *targets t* = {*b. arr t* ∧ *trg t* ∼ *b*}
unfolding *targets-def*
using *targets-char targets-def trg-def* **by** *presburger*

lemma *con-imp-cong-src*:
assumes *t* ⌢ *u*
shows *src t* ∼ *src u*
using *assms con-imp-coinitial-ax cong-transitive src-congI* **by** *blast*

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

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

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

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

lemma $seqE_{CS}$ [*elim*]:
assumes *seq t u*
and $[\![arr\ u;\ arr\ t;\ trg\ t \sim src\ u]\!] \Longrightarrow T$
shows *T*
  using *assms*
  by (*metis seq-def sources-are-cong src-in-sources trg-in-targets*)

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

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

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

lemma *coinitialE′* [*elim*]:
assumes *coinitial t u*
and ⟦*arr t*; *arr u*; *src t* ∼ *src u*⟧ ⟹ *T*
shows *T*
using *assms coinitial-iff′* **by** *blast*

lemma *coterminalI′* [*intro*]:
assumes *arr t* **and** *trg t* ∼ *trg u*
shows *coterminal t u*
by (*simp add*: *assms*(*2*) *prfx-implies-con coterminal-iff-con-trg*)

lemma *coterminalE′* [*elim*]:
assumes *coterminal t u*
and ⟦*arr t*; *arr u*; *trg t* ∼ *trg u*⟧ ⟹ *T*
shows *T*
using *assms coterminal-iff′* **by** *blast*

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

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

lemma *ide-iff-src-cong-self*:
assumes *arr a*
shows *ide a* ⟷ *src a* ∼ *a*
by (*metis assms ide-backward-stable in-sourcesE src-cong-ide src-in-sources*)

lemma *ide-iff-trg-cong-self*:
assumes *arr a*
shows *ide a* ⟷ *trg a* ∼ *a*
by (*metis assms ideE ide-backward-stable ide-trg trg-def*)

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

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

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

shows *src* (*trg t*) ∼ *trg t*
using *assms ide-trg src-cong-ide* **by** *blast*

lemma *trg-src-cong-src*:
assumes *arr t*
shows *trg* (*src t*) ∼ *src t*
using *assms*
by (*metis in-sourcesE resid-arr-self trg-ide src-in-sources*)

lemma *resid-ide-cong*:
assumes *ide a* **and** *coinitial a t*
shows *t* \ *a* ∼ *t* **and** *a* \ *t* ∼ *trg t*
using *assms*
apply (*metis coinitialE cong-reflexive ideE in-sourcesE in-sourcesI resid-arr-ide*)
by (*metis apex-arr-prfx′*(*2*) *assms coinitialE ideE in-sourcesI resid-arr-self source-is-prfx*)

lemma *con-arr-src* [*simp*]:
assumes *arr f*
shows *f* ⌢ *src f* **and** *src f* ⌢ *f*
using *assms src-in-sources con-sym* **by** *blast+*

lemma *resid-src-arr-cong*:
assumes *arr f*
shows *src f* \ *f* ∼ *trg f*
using *assms*
by (*meson resid-source-in-targets src-in-sources trg-in-targets targets-are-cong*)

lemma *resid-arr-src-cong*:
assumes *arr f*
shows *f* \ *src f* ∼ *f*
using *assms*
by (*metis cong-reflexive in-sourcesE resid-arr-ide src-in-sources*)

end

2.1.4 Weakly Extensional RTS

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

locale *weakly-extensional-rts* = *rts* +
assumes *weak-extensionality*: ⟦*t* ∼ *u*; *ide t*; *ide u*⟧ ⟹ *t* = *u*
begin

lemma *con-ide-are-eq*:
assumes *ide a* **and** *ide a′* **and** *a* ⌢ *a′*
shows *a* = *a′*
using *assms ide-imp-con-iff-cong weak-extensionality* **by** *blast*

lemma *coinitial-ide-are-eq*:
assumes *ide a* **and** *ide a'* **and** *coinitial a a'*
shows *a = a'*
using *assms coinitial-def con-ide-are-eq* **by** *blast*

lemma *arr-has-un-source*:
assumes *arr t*
shows $\exists!a.\ a \in sources\ t$
using *assms*
by (*meson arr-iff-has-source con-ide-are-eq ex-in-conv in-sourcesE sources-are-con*)

lemma *arr-has-un-target*:
assumes *arr t*
shows $\exists!b.\ b \in targets\ t$
using *assms*
by (*metis arrE arr-has-un-source arr-resid sources-resid*)

lemma *src-eqI*:
assumes *ide a* **and** *a ⌢ t*
shows *src t = a*
using *assms src-in-sources*
by (*metis arr-has-un-source resid-arr-ide in-sourcesI arr-resid-iff-con con-sym*)

lemma *sources-char*$_{WE}$:
shows $sources\ t = \{a.\ arr\ t \wedge src\ t = a\}$
using *arr-iff-has-source con-sym src-eqI* **by** *auto*

lemma *targets-char*$_{WE}$:
shows $targets\ t = \{b.\ arr\ t \wedge trg\ t = b\}$
using *trg-in-targets arr-has-un-target arr-iff-has-target* **by** *auto*

lemma *con-imp-eq-src*:
assumes *t ⌢ u*
shows *src t = src u*
using *assms*
by (*metis con-imp-coinitial-ax src-eqI*)

lemma *src-resid*$_{WE}$ [*simp*]:
assumes *t ⌢ u*
shows $src\ (t \setminus u) = trg\ u$
using *assms*
by (*metis arr-resid-iff-con con-implies-arr*(*2*) *arr-has-un-source trg-in-targets*
sources-resid src-in-sources)

lemma *apex-sym*:
shows $trg\ (t \setminus u) = trg\ (u \setminus t)$
by (*metis arr-has-un-target con-sym-ax arr-resid-iff-con conI targets-resid-sym*
trg-in-targets)

lemma *apex-arr-prfx*$_{W\,E}$:
assumes *prfx t u*
shows *trg* (*u* \ *t*) = *trg u*
and *trg* (*t* \ *u*) = *trg u*
using *assms*
apply (*metis apex-sym arr-resid-iff-con ideE src-resid*$_{W\,E}$)
by (*metis arr-resid-iff-con assms ideE src-resid*$_{W\,E}$)

lemma *seqI*$_{W\,E}$ [*intro*, *simp*]:
shows ⟦*arr t*; *trg t* = *src u*⟧ ⟹ *seq t u*
and ⟦*arr u*; *trg t* = *src u*⟧ ⟹ *seq t u*
by (*metis arrE arr-src-iff-arr arr-trg-iff-arr in-sourcesE resid-arr-ide*
seqI(*1*) *sources-resid src-in-sources trg-def*)+

lemma *seqE*$_{W\,E}$ [*elim*]:
assumes *seq t u*
and ⟦*arr u*; *arr t*; *trg t* = *src u*⟧ ⟹ *T*
shows *T*
using *assms*
by (*metis arr-has-un-source seq-def src-in-sources trg-in-targets*)

lemma *coinitial-iff*$_{W\,E}$:
shows *coinitial t u* ⟷ *arr t* ∧ *arr u* ∧ *src t* = *src u*
by (*metis arr-has-un-source coinitial-def coinitial-iff disjoint-iff-not-equal*
src-in-sources)

lemma *coterminal-iff*$_{W\,E}$:
shows *coterminal t u* ⟷ *arr t* ∧ *arr u* ∧ *trg t* = *trg u*
by (*metis arr-has-un-target coterminal-iff-con-trg coterminal-iff trg-in-targets*)

lemma *coinitialI*$_{W\,E}$ [*intro*]:
assumes *arr t* **and** *src t* = *src u*
shows *coinitial t u*
using *assms coinitial-iff*$_{W\,E}$ **by** (*metis arr-src-iff-arr*)

lemma *coinitialE*$_{W\,E}$ [*elim*]:
assumes *coinitial t u*
and ⟦*arr t*; *arr u*; *src t* = *src u*⟧ ⟹ *T*
shows *T*
using *assms coinitial-iff*$_{W\,E}$ **by** *blast*

lemma *coterminalI*$_{W\,E}$ [*intro*]:
assumes *arr t* **and** *trg t* = *trg u*
shows *coterminal t u*
using *assms coterminal-iff*$_{W\,E}$ **by** (*metis arr-trg-iff-arr*)

lemma *coterminalE*$_{W\,E}$ [*elim*]:
assumes *coterminal t u*

and $[\![arr\ t;\ arr\ u;\ trg\ t = trg\ u]\!] \Longrightarrow T$
shows T
using *assms coterminal-iff*$_{W\,E}$ **by** *blast*

lemma *src-ide* [*simp*]:
assumes *ide a*
shows *src a* = *a*
using *arrI assms src-eqI* **by** *blast*

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

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

lemma *src-src* [*simp*]:
shows *src* (*src t*) = *src t*
using *ide-src src-def src-ide* **by** *auto*

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

lemma *src-trg* [*simp*]:
shows *src* (*trg t*) = *trg t*
by (*metis con-def not-arr-null src-def src-resid*$_{W\,E}$ *trg-def*)

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

lemma *resid-ide*:
assumes *ide a* **and** *coinitial a t*
shows *t* \ *a* = *t* **and** *a* \ *t* = *trg t*
using *assms resid-arr-ide* **apply** *blast*
using *assms*
by (*metis con-def con-sym-ax ideE in-sourcesE in-sourcesI resid-ide-arr*
coinitialE src-ide src-resid$_{W\,E}$)

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

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

end

2.1.5 Extensional RTS

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

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

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

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

end

2.1.6 Composites of Transitions

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

context *rts*
begin

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

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

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

lemma *arr-composite-of*:

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

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

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

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

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

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

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

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

lemma *sources-composite-of*:

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

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

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

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

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

definition composable
where composable t u ≡ ∃v. composite-of t u v

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

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

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

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

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

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

end

RTS with Composites

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

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

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

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

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

end

2.1.7 Joins of Transitions

context *rts*
begin

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

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

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

lemma *join-ofE* [*elim*]:
assumes *join-of t u v*
and ⟦*composite-of t* (*u* \ *t*) *v*; *composite-of u* (*t* \ *u*) *v*⟧ ⟹ *T*
shows *T*
using *assms join-of-def* **by** *simp*

definition *joinable*
where *joinable t u* ≡ ∃ *v*. *join-of t u v*

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

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

lemma *joinable-iff-composable*:
shows *joinable t u* ⟷ *composable t* (*u* \ *t*)
proof
show *joinable t u* ⟹ *composable t* (*u* \ *t*)
unfolding *joinable-def join-of-def composable-def* **by** *auto*
show *composable t* (*u* \ *t*) ⟹ *joinable t u*
proof −
assume *1*: *composable t* (*u* \ *t*)
obtain *v* **where** *v*: *composite-of t* (*u* \ *t*) *v*
using *1 composable-def* **by** *blast*
have *composite-of u* (*t* \ *u*) *v*
proof
show *u* ≲ *v*
by (*metis v composite-ofE cube ide-backward-stable*)
show *v* \ *u* ∼ *t* \ *u*
by (*metis v* ‹*u* ≲ *v*› *coinitial-ide-are-cong composite-ofE*
con-imp-coinitial cube prfx-implies-con)
qed
thus *joinable t u*
using *v joinable-def* **by** *auto*
qed
qed

lemma *join-of-un-upto-cong*:
assumes *join-of t u v* **and** *join-of t u v*′
shows *v* ∼ *v*′
using *assms join-of-def composite-of-unq-upto-cong* **by** *auto*

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

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

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

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

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

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

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

lemma *join-of-respects-cong-left*:
assumes *join-of t u v* **and** *cong t t′*
shows *join-of t′ u v*
using *assms prfx-transitive cong-subst-left*(*2*) *cong-subst-right*(*2*)
apply (*elim join-ofE composite-ofE*, *intro join-ofI composite-ofI*)
by (*meson con-sym con-with-join-of-iff*(*2*) *prfx-implies-con*)+

lemma *join-of-respects-cong-right*:
assumes *join-of t u v* **and** *cong u u′*
shows *join-of t u′ v*
using *assms join-of-respects-cong-left join-of-symmetric*
by *meson*

end

RTS with Joins

locale *rts-with-joins* = *rts* +
assumes *has-joins*: *t* ⌢ *u* ⟹ *joinable t u*

2.1.8 Joins and Composites in a Weakly Extensional RTS

context *weakly-extensional-rts*
begin

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

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

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

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

end

2.1.9 Joins and Composites in an Extensional RTS

context *extensional-rts*
begin

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

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

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

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

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

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

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

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

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

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

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

lemma *arr-comp* [*intro*, *simp*]:
assumes *composable t u*
shows *arr* (*t · u*)
using *assms composable-iff-arr-comp* **by** *blast*

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

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

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

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

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

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

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

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

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

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

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

We note the following 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 $u = v$
using *assms*
by (*metis composable-def composable-iff-arr-comp composite-of-cancel-left extensionality comp-is-composite-of*(*2*))

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

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

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

definition *join* (**infix** ‹⊔› *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* $(t \sqcup u)$
using *assms joinable-def join-def join-of-unique the1I2* [*of join-of t u join-of t u*]
by *force*

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

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

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

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

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

lemma *resid-join*$_E$ [*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 $(t \sqcup u) \setminus v = (t \setminus v) \sqcup (u \setminus v)$
proof −
show *1*: $v \setminus (t \sqcup u) = (v \setminus u) \setminus (t \setminus u)$
by (*meson assms con-sym join-of-def resid-composite-of*(*3*) *extensionality*
join-is-join-of)
show $v \setminus (t \sqcup u) = (v \setminus t) \setminus (u \setminus t)$
by (*metis 1 cube*)
show $(t \sqcup u) \setminus v = (t \setminus v) \sqcup (u \setminus v)$
using *assms joinable-def join-of-resid join-is-join-of extensionality*
by (*meson join-of-unique*)
qed

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

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

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

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

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

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

lemma *con-prfx*:
shows $[\![t \frown u;\ v \lesssim u]\!] \Longrightarrow t \frown v$
and $[\![t \frown u;\ v \lesssim t]\!] \Longrightarrow v \frown u$
apply (*metis arr-resid con-arr-src*(*1*) *ide-iff-src-self prfx-implies-con resid-reflects-con src-resid*$_{W\,E}$)
by (*metis arr-resid-iff-con comp-eqI con-comp-iff con-implies-arr*(*1*) *con-sym*)

lemma *join-prfx*:
assumes $t \lesssim u$
shows $t \sqcup u = u$ **and** $u \sqcup t = u$
proof −
show $t \sqcup u = u$
using *assms*
by (*metis* (*no-types*, *lifting*) *join-eqI ide-iff-src-self ide-implies-arr resid-arr-self prfx-implies-con src-resid*$_{W\,E}$)
thus $u \sqcup t = u$
by (*metis join-sym*)
qed

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

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

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

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

lemma join-prfx-monotone:
assumes t ≲ u and u ⊔ v ⌢ t ⊔ v
shows t ⊔ v ≲ u ⊔ v
proof −
  have (t ⊔ v) \ (u ⊔ v) = (t \ u) \ (v \ u)
  proof −
    have (t ⊔ v) \ (u ⊔ v) = t \ (u ⊔ v) ⊔ v \ (u ⊔ v)
      using assms join-sym resid-join_E(3) [of t v join u v] joinable-iff-join-not-null
      by fastforce
    also have ... = (t \ u) \ (v \ u) ⊔ (v \ u) \ (v \ u)
      by (metis (full-types) assms(2) conE conI joinable-iff-join-not-null null-is-zero(1)
          resid-join_E(1−2) con-sym-ax)
    also have ... = (t \ u) \ (v \ u) ⊔ trg (v \ u)
      using trg-def by fastforce
    also have ... = (t \ u) \ (v \ u) ⊔ src ((t \ u) \ (v \ u))
      by (metis assms(1−2) con-implies-arr(1) con-target joinable-iff-arr-join
          joinable-implies-con src-resid_WE)
    also have ... = (t \ u) \ (v \ u)
      by (metis arr-resid-iff-con assms(2) con-implies-arr(1) con-sym join-def
          join-src join-sym not-arr-null resid-join_E(2))
    finally show ?thesis by blast
  qed
  moreover have ide ...
    by (metis arr-resid-iff-con assms(1−2) calculation con-sym resid-ide-arr)
  ultimately show ?thesis by presburger
qed

lemma join-eqI′:
assumes t ≲ v and u ≲ v and v \ u = t \ u and v \ t = u \ t
shows v = t ⊔ u
  using assms composite-of-def cube ideE join-of-def joinable-def join-of-unique
      join-is-join-of trg-def
  by metis
```

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

```
lemma join-expansion:
assumes joinable t u
shows t ⊔ u = t · (u \ t) and seq t (u \ t)
   apply (metis assms comp-is-composite-of(2) join-is-join-of join-of-def)
  using assms joinable-iff-composable by auto
```

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

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

end

Extensional RTS with Joins

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

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

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

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

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

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 $(t \sqcup u) \setminus v = (t \setminus v) \sqcup (u \setminus v)$
using *assms has-joins resid-join*$_E$ [*of* t u v] **by** *blast*+

lemma *join-assoc*:
shows $t \sqcup (u \sqcup v) = (t \sqcup u) \sqcup v$
proof −
  have ∗: $\bigwedge t\ u\ v.\ con\ (t \sqcup u)\ v \Longrightarrow t \sqcup (u \sqcup v) = (t \sqcup u) \sqcup v$
  proof −
    fix *t u v*
    assume *1*: *con* $(t \sqcup u)$ *v*
    have *vt-ut*: $v \setminus t \frown u \setminus t$
      using *1*
      by (*metis con-with-join-of-iff*(*2*) *join-def join-is-join-of not-con-null*(*1*))
    have *tv-uv*: $t \setminus v \frown u \setminus v$
      using *vt-ut cube con-sym*
      by (*metis arr-resid-iff-con*)
    have *2*: $(t \sqcup u) \sqcup v = (t \cdot (u \setminus t)) \cdot (v \setminus (t \cdot (u \setminus t)))$
      using *1*
      by (*metis comp-is-composite-of*(*2*) *con-implies-arr*(*1*) *has-joins join-is-join-of*
          join-of-def joinable-iff-arr-join)
    also have ... $= t \cdot ((u \setminus t) \cdot (v \setminus (t \cdot (u \setminus t))))$
      using *1*
      by (*metis calculation has-joins joinable-iff-join-not-null comp-assoc comp-def*)
    also have ... $= t \cdot ((u \setminus t) \cdot ((v \setminus t) \setminus (u \setminus t)))$
      using *1*
      by (*metis 2 comp-null*(*2*) *con-compI*(*2*) *con-comp-iff has-joins resid-comp*(*1*)
        conI joinable-iff-join-not-null)
    also have ... $= t \cdot ((v \setminus t) \sqcup (u \setminus t))$
      by (*metis vt-ut comp-is-composite-of*(*2*) *has-joins join-of-def join-is-join-of*)
    also have ... $= t \cdot ((u \setminus t) \sqcup (v \setminus t))$
      using *join-sym* **by** *metis*
    also have ... $= t \cdot ((u \sqcup v) \setminus t)$
      by (*metis tv-uv vt-ut con-implies-arr*(*2*) *con-sym con-with-join-of-iff*(*1*) *has-joins*
          join-is-join-of arr-resid-iff-con resid-join$_E$(*3*))
    also have ... $= t \sqcup (u \sqcup v)$
      by (*metis comp-is-composite-of*(*2*) *comp-null*(*2*) *conI has-joins join-is-join-of*
        join-of-def joinable-iff-join-not-null)
    finally show $t \sqcup (u \sqcup v) = (t \sqcup u) \sqcup v$
      by *simp*
  qed
  thus *?thesis*
    by (*metis* (*full-types*) *has-joins joinable-iff-join-not-null joinable-implies-con con-sym*)
qed

lemma *join-is-lub*:
assumes $t \lesssim v$ **and** $u \lesssim v$
shows $t \sqcup u \lesssim v$
proof −
  have $(t \sqcup u) \setminus v = (t \setminus v) \sqcup (u \setminus v)$
    using *assms resid-join*$_E$(*3*) [*of t u v*]
    by (*metis arr-prfx-join-self con-target con-sym join-assoc joinable-iff-con*
      joinable-iff-join-not-null prfx-implies-con resid-reflects-con)

```
  also have ... = trg v ⊔ trg v
    using assms
    by (metis ideE prfx-implies-con src-residW E trg-ide)
  also have ... = trg v
    by (metis assms(2) ide-iff-src-self ide-implies-arr join-arr-self prfx-implies-con
        src-residW E)
  finally have (t ⊔ u) \ v = trg v by blast
  moreover have ide (trg v)
    using assms
    by (metis con-implies-arr(2) prfx-implies-con conq-char trg-def)
  ultimately show ?thesis by simp
qed

end
```

Extensional RTS with Composites

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

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

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

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

  lemma arr-compEEC [elim]:
  assumes arr (t · u)
  and ⟦arr t; arr u; trg t = src u⟧ ⟹ T
  shows T
    using assms composable-iff-arr-comp composable-iff-seq by blast

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

lemma *src-comp*$_{EC}$ [*simp*]:
assumes *seq t u*
shows *src* $(t \cdot u) =$ *src t*
using *assms src-comp has-composites* **by** *simp*

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* $\wedge$ $u \frown w \setminus t$
using *composable-iff-seq con-comp-iff con-sym* **by** *meson*+

lemma *comp-assoc*$_{EC}$:
shows $t \cdot (u \cdot v) = (t \cdot u) \cdot v$
apply (*cases seq t u*)
apply (*metis arr-comp comp-assoc comp-def not-arr-null arr-compE*$_{EC}$ *arr-comp*$_{EC}$
seq-implies-arr-comp trg-comp$_{EC}$)
by (*metis comp-def composable-iff-arr-comp seqI*$_{WE}$*(1) src-comp arr-compE*$_{EC}$)

lemma *diamond-commutes*:
shows $t \cdot (u \setminus t) = u \cdot (t \setminus u)$
proof (*cases* $t \frown u$)
show $\neg\ t \frown u \Longrightarrow$ *?thesis*
by (*metis comp-null(2) conI con-sym*)
assume *con*: $t \frown u$
have $(t \cdot (u \setminus t)) \setminus u = (t \setminus u) \cdot ((u \setminus t) \setminus (u \setminus t))$
using *con*
by (*metis* (*no-types*, *lifting*) *arr-resid-iff-con con-compI(2) con-implies-arr(1)*
resid-comp(2) con-imp-arr-resid con-sym comp-def arr-comp$_{EC}$ *src-resid*$_{WE}$ *conI*)
moreover have $u \lesssim t \cdot (u \setminus t)$
by (*metis arr-resid-iff-con calculation con cong-reflexive comp-arr-trg*
resid-arr-self resid-comp(1) apex-sym)
ultimately show *?thesis*
by (*metis comp-eqI con comp-arr-trg resid-arr-self arr-resid apex-sym*)
qed

lemma *mediating-transition*:
assumes $t \cdot v = u \cdot w$
shows $v \setminus (u \setminus t) = w \setminus (t \setminus u)$
proof (*cases seq t v*)
assume *1*: *seq t v*
hence *2*: *arr* $(u \cdot w)$
using *assms* **by** (*metis arr-comp*$_{EC}$ *seqE*$_{WE}$)
have *3*: $v \setminus (u \setminus t) = ((t \cdot v) \setminus t) \setminus (u \setminus t)$
by (*metis 2 assms comp-resid-prfx*)
also have ... $= (t \cdot v) \setminus (t \cdot (u \setminus t))$
by (*metis* (*no-types*, *lifting*) *2 assms con-comp-iff*$_{EC}$*(2) con-imp-eq-src*
con-sym comp-resid-prfx prfx-comp resid-comp(1) arr-compE$_{EC}$ *arr-comp*$_{EC}$
prfx-implies-con)
also have ... $= (u \cdot w) \setminus (u \cdot (t \setminus u))$
using *assms diamond-commutes* **by** *presburger*

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

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 $(t' \setminus t) \cdot v = u \Longrightarrow v = u \setminus (t' \setminus t)$
apply (*metis assms comp-eqI arr-compE*$_{EC}$ *prfx-comp resid-comp*(*1*) *arr-resid-iff-con seq-implies-arr-comp*)
apply (*metis assms comp-resid-prfx arr-compE*$_{EC}$ *resid-comp*(*2*) *arr-resid-iff-con seq-implies-arr-comp*)
by (*metis assms*(*1*) *comp-resid-prfx seq-def*)

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

sublocale *extensional-rts-with-joins*
proof
fix *t u*
assume *con*: $t \frown u$
have *1*: *con u* $(t \cdot (u \setminus t))$
using *con-compI*(*1*) [*of t u* $\setminus$ *t u*]
by (*metis con con-implies-arr*(*1*) *con-sym diamond-commutes prfx-implies-con prfx-comp src-resid*$_{WE}$ *arr-comp*$_{EC}$)
have $t \sqcup u = t \cdot (u \setminus t)$
proof (*intro join-eqI*)
show $t \lesssim t \cdot (u \setminus t)$
by (*metis 1 composable-def comp-is-composite-of*(*2*) *composite-of-def con-comp-iff*)
moreover show *2*: $u \lesssim t \cdot (u \setminus t)$
using *1 arr-resid con con-sym prfx-reflexive resid-comp*(*1*) **by** *metis*
moreover show $(t \cdot (u \setminus t)) \setminus u = t \setminus u$
using *1 diamond-commutes induced-arrow*(*2*) *resid-comp*(*2*) **by** *force*
ultimately show $(t \cdot (u \setminus t)) \setminus t = u \setminus t$
by (*metis con-comp-iff*$_{EC}$(*1*) *con-sym prfx-implies-con resid-comp*(*2*) *induced-arrow*(*1*))

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

lemma *comp-join*$_{EC}$:
assumes *composable t u* **and** *joinable u u'*
shows *composable t* (*u* ⊔ *u'*)
and *t* · (*u* ⊔ *u'*) = *t* · *u* ⊔ *t* · *u'*
proof −
have *1*: *u* ⊔ *u'* = *u* · (*u'* \ *u*) ∧ *u* ⊔ *u'* = *u'* · (*u* \ *u'*)
using *assms joinable-implies-con diamond-commutes*
by (*metis comp-is-composite-of*(*2*) *join-is-join-of join-ofE*)
show *2*: *composable t* (*u* ⊔ *u'*)
using *assms 1 composable-iff-seq arr-comp src-join arr-compE*$_{EC}$
joinable-iff-arr-join seqI$_{WE}$(*1*)
by *metis*
have *con* (*t* · *u*) (*t* · *u'*)
using *1 2 arr-comp arr-compE*$_{EC}$ *assms*(*2*) *comp-assoc*$_{EC}$ *comp-resid-prfx*
con-comp-iff joinable-implies-con comp-def not-arr-null
by *metis*
thus *t* · (*u* ⊔ *u'*) = *t* · *u* ⊔ *t* · *u'*
using *assms comp-join*(*2*) *joinable-iff-con* **by** *blast*
qed

lemma *resid-common-prefix*:
assumes *t* · *u* ⌢ *t* · *v*
shows (*t* · *u*) \ (*t* · *v*) = *u* \ *v*
using *assms*
by (*metis con-comp-iff con-sym con-comp-iff*$_{EC}$(*2*) *con-implies-arr*(*2*)
induced-arrow(*1*) *resid-comp*(*1*−*2*) *arr-resid-iff-con*)

end

2.1.10 Confluence

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

locale *confluent-rts* = *rts* +
assumes *confluence*: *coinitial t u* ⟹ *con t u*

2.2 Simulations

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

locale *simulation* =
A: *rts A* +
B: *rts B*
for *A* :: *'a resid* (**infix** ‹\$_A$› *70*)

and $B :: 'b\ resid$ (**infix** $\langle \backslash_B \rangle$ *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** $\langle \frown_A \rangle$ *50*)
notation $A.prfx$ (**infix** $\langle \lesssim_A \rangle$ *50*)
notation $A.cong$ (**infix** $\langle \sim_A \rangle$ *50*)

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

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

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

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

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

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

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

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

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

lemma *preserves-joins*:
assumes *A.join-of t u v*
shows *B.join-of* (*F t*) (*F u*) (*F v*)
using *assms A.join-of-def B.join-of-def A.joinable-def*
by (*metis A.joinable-implies-con preserves-composites preserves-resid*)

lemma *preserves-prfx*:
assumes $t \lesssim_A u$
shows $F\ t \lesssim_B F\ u$
using *assms*
by (*metis A.prfx-implies-con preserves-ide preserves-resid*)

lemma *preserves-cong*:
assumes $t \sim_A u$
shows $F\ t \sim_B F\ u$
using *assms preserves-prfx* **by** *simp*

end

2.2.1 Identity Simulation

locale *identity-simulation* =
rts
begin

abbreviation *map*
where *map* ≡ λ*t*. *if arr t then t else null*

sublocale *simulation resid resid map*
using *con-implies-arr con-sym arr-resid-iff-con*
by *unfold-locales auto*

end

2.2.2 Composite of Simulations

lemma *simulation-comp* [*intro*]:
assumes *simulation A B F* **and** *simulation B C G*
shows *simulation A C* (*G o F*)
proof −
interpret *F*: *simulation A B F* **using** *assms*(*1*) **by** *auto*
interpret *G*: *simulation B C G* **using** *assms*(*2*) **by** *auto*
show *simulation A C* (*G o F*)
using *F.extensionality G.extensionality* **by** *unfold-locales auto*
qed

locale *composite-simulation* =

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

  abbreviation map
  where map ≡ G o F

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

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

end
```

2.2.3 Simulations into a Weakly Extensional RTS

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

  lemma preserves-src [simp]:
  shows a ∈ A.sources t ⟹ B.src (F t) = F a
    by (metis equals0D image-subset-iff B.arr-iff-has-source
        preserves-sources B.arr-has-un-source B.src-in-sources)

  lemma preserves-trg [simp]:
  shows b ∈ A.targets t ⟹ B.trg (F t) = F b
    by (metis equals0D image-subset-iff B.arr-iff-has-target
        preserves-targets B.arr-has-un-target B.trg-in-targets)

end
```

2.2.4 Simulations into an Extensional RTS

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

  notation B.comp  (infixr ‹·B› 55)
  notation B.join  (infixr ‹⊔B› 52)
```

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

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

end

2.2.5 Simulations between Weakly Extensional RTS's

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

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

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

end

2.2.6 Simulations between Extensional RTS's

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

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

notation *A.comp* (**infixr** ·*A* *55*)
notation *A.join* (**infixr** ⊔*A* *52*)

lemma *preserves-comp*:
assumes *A.composable t u*
shows *F* (*t ·A u*) = *F t ·B F u*
using *assms*

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

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

end

2.2.7 Transformations

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

The proper way to define a transformation when the domain and codomain are general RTS's is not yet clear to me. However, if the codomain is weakly extensional, then we have unique sources and targets, so there is no problem. The definition below is limited to that case. I do not make any attempt here to develop facts about transformations. My main reason for including this definition here is so that in the subsequent application to the 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* ⇒ *'b*
and *G* :: *'a* ⇒ *'b*
and τ :: *'a* ⇒ *'b* +
assumes *extensionality*: ¬ *A.arr f* ⟹ τ *f* = *B.null*
and *respects-cong-ide*: ⟦*A.ide a*; *A.cong a a'*⟧ ⟹ τ *a* = τ *a'*
and *preserves-src*: *A.ide f* ⟹ *B.src* (τ *f*) = *F f*
and *preserves-trg*: *A.ide f* ⟹ *B.trg* (τ *f*) = *G f*
and *naturality1-ax*: *a* ∈ *A.sources f* ⟹ τ *a* $\backslash_B$ *F f* = τ (*a* $\backslash_A$ *f*)
and *naturality2-ax*: *a* ∈ *A.sources f* ⟹ *F f* $\backslash_B$ τ *a* = *G f*
and *naturality3*: *a* ∈ *A.sources f* ⟹ *B.join-of* (τ *a*) (*F f*) (τ *f*)
begin

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

```
  notation B.con     (infix ‹⌢B› 50)
  notation B.prfx    (infix ‹≲B› 50)

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

  lemma naturality2:
  shows F f \B τ (A.src f) = G f
    by (metis A.src-in-sources B.null-is-zero(1) F.extensionality G.extensionality
        naturality2-ax)

  lemma respects-cong:
  assumes A.cong u u'
  shows B.cong (τ u) (τ u')
  proof −
    obtain a where a: a ∈ A.sources u ∩ A.sources u'
      using assms
      by (meson A.con-imp-common-source A.prfx-implies-con ex-in-conv)
    have B.cong (F u) (F u')
      using assms F.preserves-cong by blast
    thus ?thesis
      using a naturality3 B.join-of-respects-cong-right
      by (metis A.con-imp-common-source A.prfx-implies-con A.sources-eqI
          B.join-of-un-upto-cong assms inf.idem)
  qed

end
```

2.3 Normal Sub-RTS's and Congruence

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

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

2.3.1 Normal Sub-RTS's

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

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

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

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

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

end

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

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

  lemma *factor-closed*$_{EC}$:
  assumes $t \cdot u \in \mathfrak{N}$
  shows $t \in \mathfrak{N}$ **and** $u \in \mathfrak{N}$
    using *assms factor-closed*
    by (*metis R.arrE R.composable-def R.comp-is-composite-of*(*2*) *R.con-comp-iff*
        elements-are-arr)+

  lemma *comp-in-normal-iff*:
  shows $t \cdot u \in \mathfrak{N} \longleftrightarrow t \in \mathfrak{N} \wedge u \in \mathfrak{N} \wedge R.seq\ t\ u$
    by (*metis R.comp-is-composite-of*(*2*) *composite-closed elements-are-arr*
      factor-closed(*1*−*2*) *R.composable-def R.has-composites R.rts-with-composites-axioms*
      R.extensional-rts-axioms extensional-rts-with-composites.arr-compE$_{EC}$
      extensional-rts-with-composites-def R.seqI$_{WE}$(*1*))

end

2.3.2 Semi-Congruence

context *normal-sub-rts*
begin

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

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

  lemma *Cong*$_0$*-reflexive*:
  assumes *R.arr t*
  shows $t \approx_0 t$
    using *assms R.cong-reflexive ide-closed* **by** *simp*

  lemma *Cong*$_0$*-symmetric*:
  assumes $t \approx_0 t'$
  shows $t' \approx_0 t$
    using *assms* **by** *simp*

  lemma *Cong*$_0$*-transitive* [*trans*]:

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

lemma *Cong$_0$-imp-con*:
assumes $t \approx_0 t'$
shows *R.con t t'*
using *assms R.arr-resid-iff-con elements-are-arr* **by** *blast*

lemma *Cong$_0$-imp-coinitial*:
assumes $t \approx_0 t'$
shows *R.sources t* = *R.sources t'*
using *assms* **by** (*meson Cong$_0$-imp-con R.coinitial-iff R.con-imp-coinitial*)

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

lemma *Resid-along-normal-preserves-Cong$_0$*:
assumes $t \approx_0 t'$ **and** $u \in \mathfrak{N}$ **and** *R.sources t* = *R.sources u*
shows $t \setminus u \approx_0 t' \setminus u$
by (*metis Cong$_0$-imp-coinitial R.arr-resid-iff-con R.coinitialI R.coinitial-def R.cube R.sources-resid assms elements-are-arr forward-stable*)

lemma *Resid-along-normal-reflects-Cong$_0$*:
assumes $t \setminus u \approx_0 t' \setminus u$ **and** $u \in \mathfrak{N}$
shows $t \approx_0 t'$
using *assms*
by (*metis backward-stable R.con-imp-coinitial R.cube R.null-is-zero*(*2*) *forward-stable R.conI*)

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

lemma *Cong$_0$-subst-left*:
assumes $t \approx_0 t'$ **and** $t \frown u$
shows $t' \frown u$ **and** $t \setminus u \approx_0 t' \setminus u$
proof −
have *1*: $t \frown u \wedge t \frown t' \wedge u \setminus t \frown t' \setminus t$
using *assms*
by (*metis Resid-along-normal-preserves-Cong$_0$ Cong$_0$-imp-con Cong$_0$-reflexive R.con-sym R.null-is-zero*(*2*) *R.arr-resid-iff-con R.sources-resid R.conI*)
hence *2*: $t' \frown u \wedge u \setminus t \frown t' \setminus t \wedge$
$(t \setminus u) \setminus (t' \setminus u) = (t \setminus t') \setminus (u \setminus t') \wedge$
$(t' \setminus u) \setminus (t \setminus u) = (t' \setminus t) \setminus (u \setminus t)$
by (*meson R.con-sym R.cube R.resid-reflects-con*)
show $t' \frown u$
using *2* **by** *simp*
show $t \setminus u \approx_0 t' \setminus u$
using *assms 1 2*
by (*metis R.arr-resid-iff-con R.con-imp-coinitial R.cube forward-stable*)
qed

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

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

lemma *Cong$_0$-subst-right*:
assumes $u \approx_0 u'$ **and** $t \frown u$
shows $t \frown u'$ **and** $(t \setminus u) \setminus (u' \setminus u) \approx_0 (t \setminus u') \setminus (u \setminus u')$
using *assms*
apply (*meson Cong$_0$-subst-left(1) R.con-sym*)
using *assms*
by (*metis R.sources-resid Cong$_0$-imp-con Cong$_0$-reflexive Resid-along-normal-preserves-Cong$_0$ R.arr-resid-iff-con residuation.cube R.residuation-axioms*)

lemma *Cong$_0$-subst-Con*:
assumes $t \approx_0 t'$ **and** $u \approx_0 u'$
shows $t \frown u \longleftrightarrow t' \frown u'$
using *assms*
by (*meson Cong$_0$-subst-left(1) Cong$_0$-subst-right(1)*)

lemma *Cong$_0$-cancel-left*:
assumes *R.composite-of t u v* **and** *R.composite-of t u' v'* **and** $v \approx_0 v'$
shows $u \approx_0 u'$
proof −
have $u \approx_0 v \setminus t$
using *assms(1) ide-closed* **by** *blast*
also have $v \setminus t \approx_0 v' \setminus t$
by (*meson assms(1,3) Cong$_0$-subst-left(2) R.composite-of-def R.con-sym R.prfx-implies-con*)
also have $v' \setminus t \approx_0 u'$
using *assms(2) ide-closed* **by** *blast*
finally show *?thesis* **by** *auto*
qed

lemma *Cong$_0$-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-of t u v $\wedge$ *R.composite-of t' u' v'*)
proof (*intro iffI*)
show $\exists u\ u'\ v\ v'.\ u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge v \approx_0 v' \wedge$
R.composite-of t u v $\wedge$ *R.composite-of t' u' v'*
$\Longrightarrow t \approx_0 t'$
by (*meson Cong$_0$-transitive R.composite-of-def ide-closed prfx-closed*)
show $t \approx_0 t' \Longrightarrow \exists u\ u'\ v\ v'.\ u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge v \approx_0 v' \wedge$
R.composite-of t u v $\wedge$ *R.composite-of t' u' v'*
by (*metis Cong$_0$-imp-con Cong$_0$-transitive R.composite-of-def R.prfx-reflexive R.arrI R.ideE*)
qed

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

```
  have v \ v ≈₀ v' \ v ∧ v' \ v' ≈₀ v \ v'
  proof−
    have 1: (v \ t) \ (u \ t) ≈₀ (v' \ u) \ (t \ u)
      using assms(2−3) R.cube [of v t u]
      by (metis R.con-target R.composite-ofE R.ide-imp-con-iff-cong ide-closed
          R.conI)
    have 2: v \ v ≈₀ v' \ v
    proof −
      have v \ v ≈₀ (v \ t) \ (u \ t)
        using assms R.composite-of-def ide-closed
        by (meson R.composite-of-unq-upto-cong R.prfx-implies-con R.resid-composite-of(3))
      also have (v \ t) \ (u \ t) ≈₀ (v' \ u) \ (t \ u)
        using 1 by simp
      also have (v' \ u) \ (t \ u) ≈₀ (v' \ t) \ (u \ t)
        by (metis 1 Cong₀-transitive R.cube)
      also have (v' \ t) \ (u \ t) ≈₀ v' \ v
        using assms R.composite-of-def ide-closed
        by (metis 1 R.conI R.con-sym-ax R.cube R.null-is-zero(2) R.resid-composite-of(3))
      finally show ?thesis by auto
    qed
    moreover have v' \ v' ≈₀ v \ v'
    proof −
      have v' \ v' ≈₀ (v' \ u) \ (t \ u)
        using assms R.composite-of-def ide-closed
        by (meson R.composite-of-unq-upto-cong R.prfx-implies-con R.resid-composite-of(3))
      also have (v' \ u) \ (t \ u) ≈₀ (v \ t) \ (u \ t)
        using 1 by simp
      also have (v \ t) \ (u \ t) ≈₀ (v \ u) \ (t \ u)
        using R.cube [of v t u] ide-closed
        by (metis Cong₀-reflexive R.arr-resid-iff-con assms(2) R.composite-of-def
              R.prfx-implies-con)
      also have (v \ u) \ (t \ u) ≈₀ v \ v'
        using assms R.composite-of-def ide-closed
        by (metis 2 R.conI elements-are-arr R.not-arr-null R.null-is-zero(2)
            R.resid-composite-of(3))
      finally show ?thesis by auto
    qed
    ultimately show ?thesis by blast
  qed
  thus ?thesis
    by (metis assms(2−3) R.composite-of-unq-upto-cong R.resid-arr-ide Cong₀-imp-con)
qed
```

2.3.3 Congruence

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

```
definition Cong  (infix ‹≈› 50)
where Cong t t' ≡ ∃u u'. u ∈ 𝔑 ∧ u' ∈ 𝔑 ∧ t \ u ≈₀ t' \ u'
```

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

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

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

lemma *Cong-reflexive*:
assumes *R.arr t*
shows $t \approx t$
  by (*metis CongI Cong*$_0$*-reflexive assms R.con-imp-coinitial-ax ide-closed*
      R.resid-arr-ide R.arrE R.con-sym)

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

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

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

```
    have (v'' \ u'') \ (t'' \ u'') ∈ 𝔑
      by (metis Cong₀-transitive R.con-imp-coinitial forward-stable Cong₀-imp-con
          resid-along-elem-preserves-con R.arrI R.arr-resid-iff-con u''v'' uu'' v'v'')
    thus ?thesis
      by (metis Cong₀-subst-left(2) R.con-sym R.null-is-zero(1) uu'' w'' R.conI)
  qed
  have 2: ?w'' \ ?w ∈ 𝔑
    by (metis 1 Cong₀-subst-left(2) uu'' w'' R.conI)
  have 3: R.seq u (v'' \ u'')
    by (metis (full-types) 2 Cong₀-imp-coinitial R.sources-resid
        Cong₀-imp-con R.arr-resid-iff-con R.con-implies-arr(2) R.seqI(1) uu'' R.conI)
  have 4: R.seq v' (u'' \ v'')
    by (metis 1 Cong₀-imp-coinitial Cong₀-imp-con R.arr-resid-iff-con
        R.con-implies-arr(2) R.seq-def R.sources-resid v'v'' R.conI)
  obtain x where x: R.composite-of u (v'' \ u'') x
    using 3 composite-closed-left uu'' by blast
  obtain x' where x': R.composite-of v' (u'' \ v'') x'
    using 4 composite-closed-left v'v'' by presburger
  have ?w ≈₀ ?w'
  proof −
    have ?w ≈₀ ?w'' ∧ ?w' ≈₀ ?w''
      using 1 2
      by (metis Cong₀-subst-left(2) R.null-is-zero(2) v'v'' R.conI)
    thus ?thesis
      using Cong₀-transitive by blast
  qed
  moreover have x ∈ 𝔑 ∧ ?w ≈₀ t \ x
    apply (intro conjI)
      apply (meson composite-closed forward-stable u''v'' uu'' v'v'' x)
     apply (metis (full-types) R.arr-resid-iff-con R.con-implies-arr(2) R.con-sym
        ide-closed forward-stable R.composite-of-def R.resid-composite-of(3)
        Cong₀-subst-right(1) prfx-closed u''v'' uu'' v'v'' x R.conI)
    by (metis (no-types, lifting) 1 R.con-composite-of-iff ide-closed
        R.resid-composite-of(3) R.arr-resid-iff-con R.con-implies-arr(1) R.con-sym x R.conI)
  moreover have x' ∈ 𝔑 ∧ ?w' ≈₀ t' \ x'
    apply (intro conjI)
      apply (meson composite-closed forward-stable uu'' v''u'' v'v'' x')
     apply (metis (full-types) Cong₀-subst-right(1) R.composite-ofE R.con-sym
        ide-closed forward-stable R.con-imp-coinitial prfx-closed
        R.resid-composite-of(3) R.arr-resid-iff-con R.con-implies-arr(1) uu'' v'v'' x' R.conI)
    by (metis (full-types) Cong₀-subst-left(1) R.composite-ofE R.con-sym ide-closed
        forward-stable R.con-imp-coinitial prfx-closed R.resid-composite-of(3)
        R.arr-resid-iff-con R.con-implies-arr(1) uu'' v'v'' x' R.conI)
  ultimately show t ≈ t'
    using Cong-def Cong₀-transitive by metis
qed

lemma Cong-closure-props:
shows t ≈ u ⟹ u ≈ t
```

and $[\![t \approx u;\ u \approx v]\!] \Longrightarrow t \approx v$
and $t \approx_0 u \Longrightarrow t \approx u$
and $[\![u \in \mathfrak{N};\ R.sources\ t = R.sources\ u]\!] \Longrightarrow t \approx t \setminus u$
proof −
  show $t \approx u \Longrightarrow u \approx t$
    using *Cong-symmetric* **by** *blast*
  show $[\![t \approx u;\ u \approx v]\!] \Longrightarrow t \approx v$
    using *Cong-transitive* **by** *blast*
  show $t \approx_0 u \Longrightarrow t \approx u$
    by (*metis Cong*$_0$*-subst-left*(*2*) *Cong-def Cong-reflexive R.con-implies-arr*(*1*)
        R.null-is-zero(*2*) *R.conI*)
  show $[\![u \in \mathfrak{N};\ R.sources\ t = R.sources\ u]\!] \Longrightarrow t \approx t \setminus u$
  proof −
    assume *u*: $u \in \mathfrak{N}$ **and** *coinitial*: $R.sources\ t = R.sources\ u$
    obtain *a* **where** *a*: $a \in R.targets\ u$
      by (*meson elements-are-arr empty-subsetI R.arr-iff-has-target subsetI subset-antisym u*)
    have $t \setminus u \approx_0 (t \setminus u) \setminus a$
    proof −
      have $R.arr\ t$
        using *R.arr-iff-has-source coinitial elements-are-arr u* **by** *presburger*
      thus *?thesis*
        by (*meson u a R.arr-resid-iff-con coinitial ide-closed forward-stable*
            elements-are-arr R.coinitial-iff R.composite-of-arr-target R.resid-composite-of(*3*))
    qed
    thus *?thesis*
      using *Cong-def*
      by (*metis a R.composite-of-arr-target elements-are-arr factor-closed*(*2*) *u*)
  qed
qed

lemma *Cong*$_0$*-implies-Cong*:
assumes $t \approx_0 t'$
shows $t \approx t'$
  using *assms Cong-closure-props*(*3*) **by** *simp*

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

```
    have a \ u ∈ R.targets u
      by (metis Cong₀-imp-con R.arr-resid-iff-con assms(2) R.con-imp-common-source
          R.con-implies-arr(1) R.resid-source-in-targets R.sources-eqI uu')
    moreover have a' \ u' ∈ R.targets u'
      by (metis Cong₀-imp-con R.arr-resid-iff-con assms(3) R.con-imp-common-source
          R.resid-source-in-targets R.con-implies-arr(1) R.sources-eqI uu')
    moreover have R.targets u = R.targets u'
      by (metis Cong₀-imp-coinitial Cong₀-imp-con R.arr-resid-iff-con
          R.con-implies-arr(1) R.sources-resid uu')
    ultimately show ?thesis
      using ide-closed R.targets-are-cong by presburger
  qed
 qed
qed

lemma in-targets-respects-Cong:
assumes t ≈ t' and b ∈ R.targets t and b' ∈ R.targets t'
shows b ≈ b'
proof −
 obtain u u' where uu': u ∈ 𝔑 ∧ u' ∈ 𝔑 ∧ t \ u ≈₀ t' \ u'
   using assms Cong-def by blast
 have seq: R.seq (u \ t) ((t' \ u') \ (t \ u)) ∧ R.seq (u' \ t') ((t \ u) \ (t' \ u'))
   by (metis R.arr-iff-has-source R.arr-iff-has-target R.conI elements-are-arr R.not-arr-null
       R.seqI(2) R.sources-resid R.targets-resid-sym uu')
 obtain v where v: R.composite-of (u \ t) ((t' \ u') \ (t \ u)) v
   using seq composite-closed-right uu' by presburger
 obtain v' where v': R.composite-of (u' \ t') ((t \ u) \ (t' \ u')) v'
   using seq composite-closed-right uu' by presburger
 show b ≈ b'
 proof
   show v-in-𝔑: v ∈ 𝔑
     by (metis composite-closed R.con-imp-coinitial R.con-implies-arr(1) forward-stable
         R.composite-of-def R.prfx-implies-con R.arr-resid-iff-con R.con-sym uu' v)
   show v'-in-𝔑: v' ∈ 𝔑
     by (metis backward-stable R.composite-of-def R.con-imp-coinitial forward-stable
         R.null-is-zero(2) prfx-closed uu' v' R.conI)
   show b \ v ≈₀ b' \ v'
     using assms uu' v v'
     by (metis R.arr-resid-iff-con ide-closed R.seq-def R.sources-resid R.targets-resid-sym
         R.resid-source-in-targets seq R.sources-composite-of R.targets-are-cong
         R.targets-composite-of)
 qed
qed

lemma sources-are-Cong:
assumes a ∈ R.sources t and a' ∈ R.sources t
shows a ≈ a'
 using assms
 by (simp add: ide-closed R.sources-are-cong Cong-closure-props(3))
```

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

It is *not* the case that sources and targets are $\approx$-closed; *i.e.* $t \approx t' \Longrightarrow sources\ t = sources\ t'$ 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 $t \setminus u \frown t' \setminus u \longleftrightarrow t \frown t'$
by (*metis R.arr-resid-iff-con assms R.con-implies-arr*(*1*−*2*) *elements-are-arr R.coinitial-iff R.resid-reflects-con resid-along-elem-preserves-con*)

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

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

lemma *Cong'-if*:
shows $[\![u \in \mathfrak{N};\ u' \in \mathfrak{N};\ t \setminus u \approx_0 t' \setminus u']\!] \Longrightarrow Cong'\ t\ t'$
proof −
assume *u*: $u \in \mathfrak{N}$ **and** *u'*: $u' \in \mathfrak{N}$ **and** *1*: $t \setminus u \approx_0 t' \setminus u'$
show $Cong'\ t\ t'$
using *u u' 1*
by (*metis* (*no-types*, *lifting*) *Cong'.simps Cong*$_0$*-imp-con R.arr-resid-iff-con R.coinitial-iff R.con-imp-coinitial*)
qed

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

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

2.3.4 Congruence Classes

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

definition *Cong-class* (‹⦇-⦈›)
where *Cong-class* $t \equiv \{t'.\ t \approx t'\}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end

2.3.5 Coherent Normal Sub-RTS's

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

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

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

$\longrightarrow t \setminus w \approx_0 t' \setminus w'$
show $\forall t\ u\ u'.\ R.arr\ t \wedge u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge R.sources\ t = R.sources\ u \wedge$
$R.sources\ u = R.sources\ u' \wedge R.targets\ u = R.targets\ u'$
$\longrightarrow t \setminus u \approx_0 t \setminus u'$
proof (*intro allI impI*, *elim conjE*)
fix $t\ u\ u'$
assume *t*: $R.arr\ t$ **and** *u*: $u \in \mathfrak{N}$ **and** *u'*: $u' \in \mathfrak{N}$
and *tu*: $R.sources\ t = R.sources\ u$ **and** *sources*: $R.sources\ u = R.sources\ u'$
and *targets*: $R.targets\ u = R.targets\ u'$
show $t \setminus u \approx_0 t \setminus u'$
by (*metis 1 Cong*$_0$*-reflexive Resid-along-normal-preserves-Cong*$_0$ *sources t targets tu u u'*)
qed
next
assume *1*: $\forall t\ u\ u'.\ R.arr\ t \wedge u \in \mathfrak{N} \wedge u' \in \mathfrak{N} \wedge R.sources\ t = R.sources\ u \wedge$
$R.sources\ u = R.sources\ u' \wedge R.targets\ u = R.targets\ u'$
$\longrightarrow t \setminus u \approx_0 t \setminus u'$
show $\forall t\ t'\ v\ v'\ w\ w'.\ v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge$
$R.sources\ v = R.sources\ w \wedge R.sources\ v' = R.sources\ w' \wedge$
$R.targets\ w = R.targets\ w' \wedge t \setminus v \approx_0 t' \setminus v'$
$\longrightarrow t \setminus w \approx_0 t' \setminus w'$
proof (*intro allI impI*, *elim conjE*)
fix $t\ t'\ v\ v'\ w\ w'$
assume *v*: $v \in \mathfrak{N}$ **and** *v'*: $v' \in \mathfrak{N}$ **and** *w*: $w \in \mathfrak{N}$ **and** *w'*: $w' \in \mathfrak{N}$
and *vw*: $R.sources\ v = R.sources\ w$ **and** *v'w'*: $R.sources\ v' = R.sources\ w'$
and *ww'*: $R.targets\ w = R.targets\ w'$
and *tvt'v'*: $(t \setminus v) \setminus (t' \setminus v') \in \mathfrak{N}$ **and** *t'v'tv*: $(t' \setminus v') \setminus (t \setminus v) \in \mathfrak{N}$
show $t \setminus w \approx_0 t' \setminus w'$
proof −
have *3*: $R.sources\ t = R.sources\ v \wedge R.sources\ t' = R.sources\ v'$
using *R.con-imp-coinitial*
by (*meson Cong*$_0$*-imp-con tvt'v' t'v'tv R.coinitial-iff R.arr-resid-iff-con*)
have *2*: $t \setminus w \approx t' \setminus w'$
using *Cong-closure-props*
by (*metis tvt'v' t'v'tv 3 vw v'w' v v' w w'*)
obtain *z z'* **where** *zz'*: $z \in \mathfrak{N} \wedge z' \in \mathfrak{N} \wedge (t \setminus w) \setminus z \approx_0 (t' \setminus w') \setminus z'$
using *2* **by** *auto*
have $(t \setminus w) \setminus z \approx_0 (t \setminus w) \setminus z'$
proof −
have *R.coinitial* $((t \setminus w) \setminus z)\ ((t \setminus w) \setminus z')$
proof −
have $R.targets\ z = R.targets\ z'$
using *ww' zz'*
by (*metis Cong*$_0$*-imp-coinitial Cong*$_0$*-imp-con R.con-sym-ax R.null-is-zero*(*2*) *R.sources-resid R.conI*)
moreover have $R.sources\ ((t \setminus w) \setminus z) = R.targets\ z$
using *ww' zz'*
by (*metis R.con-def R.not-arr-null R.null-is-zero*(*2*)

```
            R.sources-resid elements-are-arr)
        moreover have R.sources ((t \ w) \ z') = R.targets z'
          using ww' zz'
          by (metis Cong-closure-props(4) Cong-imp-arr(2) R.arr-resid-iff-con
              R.coinitial-iff R.con-imp-coinitial R.sources-resid)
        ultimately show ?thesis
          using ww' zz'
          apply (intro R.coinitialI)
           apply auto
          by (meson R.arr-resid-iff-con R.con-implies-arr(2) elements-are-arr)
      qed
      thus ?thesis
        apply (intro conjI)
        by (metis 1 R.coinitial-iff R.con-imp-coinitial R.arr-resid-iff-con
              R.sources-resid zz')+
    qed
    hence (t \ w) \ z' ≈₀ (t' \ w') \ z'
      using zz' Cong₀-transitive Cong₀-symmetric by blast
    thus ?thesis
      using zz' Resid-along-normal-reflects-Cong₀ by metis
  qed
 qed
qed

end

context coherent-normal-sub-rts
begin
```

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

```
lemma coherent':
assumes v ∈ 𝔑 and v' ∈ 𝔑 and w ∈ 𝔑 and w' ∈ 𝔑
and R.sources v = R.sources w and R.sources v' = R.sources w'
and R.targets w = R.targets w' and t \ v ≈₀ t' \ v'
shows t \ w ≈₀ t' \ w'
proof
  show (t \ w) \ (t' \ w') ∈ 𝔑
    using assms coherent coherent-iff by meson
  show (t' \ w') \ (t \ w) ∈ 𝔑
    using assms coherent coherent-iff by meson
qed
```

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

```
lemma Cong-subst:
assumes t ≈ t' and u ≈ u' and t ⌢ u and R.sources t' = R.sources u'
shows t' ⌢ u' and t \ u ≈ t' \ u'
proof −
  obtain v v' where vv': v ∈ 𝔑 ∧ v' ∈ 𝔑 ∧ t \ v ≈₀ t' \ v'
```

using *assms* **by** *auto*
obtain $w\ w'$ **where** *ww'*: $w \in \mathfrak{R} \wedge w' \in \mathfrak{R} \wedge u \setminus w \approx_0 u' \setminus w'$
using *assms* **by** *auto*
let *?x* $= t \setminus v$ **and** *?x'* $= t' \setminus v'$
let *?y* $= u \setminus w$ **and** *?y'* $= u' \setminus w'$
have *xx'*: *?x* $\approx_0$ *?x'*
using *assms vv'* **by** *blast*
have *yy'*: *?y* $\approx_0$ *?y'*
using *assms ww'* **by** *blast*
have *1*: $t \setminus w \approx_0 t' \setminus w'$
proof −
have *R.sources v* = *R.sources w*
by (*metis* (*no-types*, *lifting*) *Cong*$_0$*-imp-con R.arr-resid-iff-con assms*(*3*)
R.con-imp-common-source R.con-implies-arr(*2*) *R.sources-eqI ww' xx'*)
moreover have *R.sources v'* = *R.sources w'*
by (*metis* (*no-types*, *lifting*) *assms*(*4*) *R.coinitial-iff R.con-imp-coinitial*
Cong$_0$*-imp-con R.arr-resid-iff-con ww' xx'*)
moreover have *R.targets w* = *R.targets w'*
by (*metis Cong*$_0$*-implies-Cong Cong*$_0$*-imp-coinitial Cong-imp-arr*(*1*)
R.arr-resid-iff-con R.sources-resid ww')
ultimately show *?thesis*
using *assms vv' ww'*
by (*intro coherent'* [*of v v' w w' t*]) *auto*
qed
have *2*: $t' \setminus w' \frown u' \setminus w'$
using *assms 1 ww'*
by (*metis Cong*$_0$*-subst-left*(*1*) *Cong*$_0$*-subst-right*(*1*)
Resid-along-normal-preserves-reflects-con R.arr-resid-iff-con
R.coinitial-iff R.con-imp-coinitial elements-are-arr)
thus *3*: $t' \frown u'$
using *ww' R.cube* **by** *force*
have $t \setminus u \approx ((t \setminus u) \setminus (w \setminus u)) \setminus (?y' \setminus ?y)$
proof −
have $t \setminus u \approx (t \setminus u) \setminus (w \setminus u)$
by (*metis Cong-closure-props*(*4*) *assms*(*3*) *R.con-imp-coinitial*
elements-are-arr forward-stable R.arr-resid-iff-con R.con-implies-arr(*1*)
R.sources-resid ww')
also have ... $\approx ((t \setminus u) \setminus (w \setminus u)) \setminus (?y' \setminus ?y)$
by (*metis Cong*$_0$*-imp-con Cong-closure-props*(*4*) *Cong-imp-arr*(*2*)
R.arr-resid-iff-con calculation R.con-implies-arr(*2*) *R.targets-resid-sym*
R.sources-resid ww')
finally show *?thesis* **by** *simp*
qed
also have ... $\approx (((t \setminus w) \setminus ?y) \setminus (?y' \setminus ?y))$
using *ww'*
by (*metis Cong-imp-arr*(*2*) *Cong-reflexive calculation R.cube*)
also have ... $\approx (((t' \setminus w') \setminus ?y) \setminus (?y' \setminus ?y))$
using *1 Cong*$_0$*-subst-left*(*2*) [*of t* $\setminus$ *w* (*t'* $\setminus$ *w'*) *?y*]
Cong$_0$*-subst-left*(*2*) [*of* (*t* $\setminus$ *w*) $\setminus$ *?y* (*t'* $\setminus$ *w'*) $\setminus$ *?y ?y'* $\setminus$ *?y*]

by (*meson 2 Cong$_0$-implies-Cong Cong$_0$-subst-Con Cong-imp-arr(2) R.arr-resid-iff-con calculation ww'*)
also have ... $\approx ((t' \setminus w') \setminus ?y') \setminus (?y \setminus ?y')$
using *2 Cong$_0$-implies-Cong Cong$_0$-subst-right(2) ww'* **by** *presburger*
also have *4*: ... $\approx (t' \setminus u') \setminus (w' \setminus u')$
using *2 ww'*
by (*metis Cong$_0$-imp-con Cong-closure-props(4) Cong-symmetric R.cube R.sources-resid*)
also have ... $\approx t' \setminus u'$
using *ww' 3 4*
by (*metis Cong-closure-props(4) Cong-imp-arr(2) Cong-symmetric R.con-imp-coinitial R.con-implies-arr(2) forward-stable R.sources-resid R.arr-resid-iff-con*)
finally show $t \setminus u \approx t' \setminus u'$ **by** *simp*
qed

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

lemma *Cong$_0$-composite-of-arr-normal*:
assumes *R.composite-of t u t'* **and** $u \in \mathfrak{N}$
shows $t' \approx_0 t$
using *assms backward-stable R.composite-of-def ide-closed* **by** *blast*

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

end

context *normal-sub-rts*
begin

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

lemma *Cong-subst-iff-coherent'*:
shows $(\forall\, t\ t'\ u\ u'.\ t \approx t' \wedge u \approx u' \wedge t \frown u \wedge \textit{R.sources}\ t' = \textit{R.sources}\ u' \longrightarrow t' \frown u' \wedge t \setminus u \approx t' \setminus u')$
$\longleftrightarrow$
$(\forall\, t\ t'\ v\ v'\ w\ w'.\ v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge$
R.sources v = *R.sources w* $\wedge$ *R.sources v'* = *R.sources w'* $\wedge$
R.targets w = *R.targets w'* $\wedge$ $t \setminus v \approx_0 t' \setminus v'$
$\longrightarrow t \setminus w \approx_0 t' \setminus w')$
proof
assume *1*: $\forall\, t\ t'\ u\ u'.\ t \approx t' \wedge u \approx u' \wedge t \frown u \wedge \textit{R.sources}\ t' = \textit{R.sources}\ u'$

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

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

proof (*intro allI impI*, *elim conjE*)
fix $t\ t'\ v\ v'\ w\ w'$
assume *v*: $v \in \mathfrak{R}$ **and** *v'*: $v' \in \mathfrak{R}$ **and** *w*: $w \in \mathfrak{R}$ **and** *w'*: $w' \in \mathfrak{R}$
and *sources-vw*: $R.sources\ v = R.sources\ w$
and *sources-v'w'*: $R.sources\ v' = R.sources\ w'$
and *targets-ww'*: $R.targets\ w = R.targets\ w'$
and *tt'*: $(t \setminus v) \setminus (t' \setminus v') \in \mathfrak{R}$ **and** *t't*: $(t' \setminus v') \setminus (t \setminus v) \in \mathfrak{R}$
show $t \setminus w \approx_0 t' \setminus w'$
proof −
have *2*: $\bigwedge t\ t'\ u\ u'.\ [\![t \approx t';\ u \approx u';\ t \frown u;\ R.sources\ t' = R.sources\ u']\!]$
$\Longrightarrow t' \frown u' \wedge t \setminus u \approx t' \setminus u'$
using *1* **by** *blast*
have *3*: $t \setminus w \approx t \setminus v \wedge t' \setminus w' \approx t' \setminus v'$
by (*metis tt' t't sources-vw sources-v'w' Cong$_0$-subst-right*(*2*)
Cong-closure-props(*4*) *Cong-def R.arr-resid-iff-con Cong-closure-props*(*3*)
Cong-imp-arr(*1*) *normal-is-Cong-closed v w v' w'*)
have $(t \setminus w) \setminus (t' \setminus w') \approx (t \setminus v) \setminus (t' \setminus v')$
using *2* [*of* $t \setminus w\ t \setminus v\ t' \setminus w'\ t' \setminus v'$] *3*
by (*metis tt' t't targets-ww' 1 Cong$_0$-imp-con Cong-imp-arr*(*1*) *Cong-symmetric*
R.arr-resid-iff-con R.sources-resid)
moreover have $(t' \setminus w') \setminus (t \setminus w) \approx (t' \setminus v') \setminus (t \setminus v)$
using *2 3*
by (*metis tt' t't targets-ww' Cong$_0$-imp-con Cong-symmetric*
Cong-imp-arr(*1*) *R.arr-resid-iff-con R.sources-resid*)
ultimately show *?thesis*
by (*meson tt' t't normal-is-Cong-closed Cong-symmetric*)
qed
qed
next
assume *1*: $\forall t\ t'\ v\ v'\ w\ w'.\ v \in \mathfrak{R} \wedge v' \in \mathfrak{R} \wedge w \in \mathfrak{R} \wedge w' \in \mathfrak{R} \wedge$
$R.sources\ v = R.sources\ w \wedge R.sources\ v' = R.sources\ w' \wedge$
$R.targets\ w = R.targets\ w' \wedge t \setminus v \approx_0 t' \setminus v'$
$\longrightarrow t \setminus w \approx_0 t' \setminus w'$
show $\forall t\ t'\ u\ u'.\ t \approx t' \wedge u \approx u' \wedge t \frown u \wedge R.sources\ t' = R.sources\ u'$
$\longrightarrow t' \frown u' \wedge t \setminus u \approx t' \setminus u'$
proof (*intro allI impI*, *elim conjE*, *intro conjI*)
have $*$: $\bigwedge t\ t'\ v\ v'\ w\ w'.\ [\![v \in \mathfrak{R};\ v' \in \mathfrak{R};\ w \in \mathfrak{R};\ w' \in \mathfrak{R};$
$R.sources\ v = R.sources\ w;\ R.sources\ v' = R.sources\ w';$
$R.targets\ v = R.targets\ v';\ R.targets\ w = R.targets\ w';$
$t \setminus v \approx_0 t' \setminus v']\!]$
$\Longrightarrow t \setminus w \approx_0 t' \setminus w'$
using *1* **by** *metis*
fix $t\ t'\ u\ u'$
assume *tt'*: $t \approx t'$ **and** *uu'*: $u \approx u'$ **and** *con*: $t \frown u$

and $t'u'$: *R.sources* $t' = R.sources\ u'$
obtain $v\ v'$ **where** vv': $v \in \mathfrak{N} \wedge v' \in \mathfrak{N} \wedge t \setminus v \approx_0 t' \setminus v'$
using tt' **by** *auto*
obtain $w\ w'$ **where** ww': $w \in \mathfrak{N} \wedge w' \in \mathfrak{N} \wedge u \setminus w \approx_0 u' \setminus w'$
using uu' **by** *auto*
let $?x = t \setminus v$ **and** $?x' = t' \setminus v'$
let $?y = u \setminus w$ **and** $?y' = u' \setminus w'$
have xx': $?x \approx_0 ?x'$
using $tt'\ vv'$ **by** *blast*
have yy': $?y \approx_0 ?y'$
using $uu'\ ww'$ **by** *blast*
have *1*: $t \setminus w \approx_0 t' \setminus w'$
proof −
have *R.sources* $v = R.sources\ w \wedge R.sources\ v' = R.sources\ w'$
proof
show *R.sources* $v' = R.sources\ w'$
using *Cong*$_0$*-imp-con R.arr-resid-iff-con R.coinitial-iff R.con-imp-coinitial*
$t'u'\ vv'\ ww'$
by *metis*
show *R.sources* $v = R.sources\ w$
by (*metis con elements-are-arr R.not-arr-null R.null-is-zero*(*2*) *R.conI*
R.con-imp-common-source rts.sources-eqI R.rts-axioms $vv'\ ww'$)
qed
moreover have *R.targets* $v = R.targets\ v' \wedge R.targets\ w = R.targets\ w'$
by (*metis Cong*$_0$*-imp-coinitial Cong*$_0$*-imp-con R.arr-resid-iff-con*
R.con-implies-arr(*2*) *R.sources-resid* $vv'\ ww'$)
ultimately show *?thesis*
using $vv'\ ww'\ xx'$
by (*intro* $*$ [*of* $v\ v'\ w\ w'\ t\ t'$]) *auto*
qed
have *2*: $t' \setminus w' \frown u' \setminus w'$
using *1* $tt'\ ww'$
by (*meson Cong*$_0$*-imp-con Cong*$_0$*-subst-Con R.arr-resid-iff-con con R.con-imp-coinitial*
R.con-implies-arr(*2*) *resid-along-elem-preserves-con*)
thus *3*: $t' \frown u'$
using ww' *R.cube* **by** *force*
have $t \setminus u \approx (t \setminus u) \setminus (w \setminus u)$
by (*metis Cong-closure-props*(*4*) *R.arr-resid-iff-con con R.con-imp-coinitial*
elements-are-arr forward-stable R.con-implies-arr(*2*) *R.sources-resid* ww')
also have $(t \setminus u) \setminus (w \setminus u) \approx ((t \setminus u) \setminus (w \setminus u)) \setminus (?y' \setminus ?y)$
using yy'
by (*metis Cong*$_0$*-imp-con Cong-closure-props*(*4*) *Cong-imp-arr*(*2*)
R.arr-resid-iff-con calculation R.con-implies-arr(*2*) *R.sources-resid R.targets-resid-sym*)
also have ... $\approx (((t \setminus w) \setminus ?y) \setminus (?y' \setminus ?y))$
using ww'
by (*metis Cong-imp-arr*(*2*) *Cong-reflexive calculation R.cube*)
also have ... $\approx (((t' \setminus w') \setminus ?y) \setminus (?y' \setminus ?y))$
proof −
have $((t \setminus w) \setminus ?y) \setminus (?y' \setminus ?y) \approx_0 ((t' \setminus w') \setminus ?y) \setminus (?y' \setminus ?y)$

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

end
```

2.3.6 Quotient by Coherent Normal Sub-RTS

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

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

  definition Resid  (infix ‹⦃\⦄› 70)
  where 𝒯 ⦃\⦄ 𝒰 ≡
        if N.is-Cong-class 𝒯 ∧ N.is-Cong-class 𝒰 ∧ (∃ t u. t ∈ 𝒯 ∧ u ∈ 𝒰 ∧ t ⌣ u)
        then N.Cong-class
               (fst (SOME tu. fst tu ∈ 𝒯 ∧ snd tu ∈ 𝒰 ∧ fst tu ⌣ snd tu) \
                snd (SOME tu. fst tu ∈ 𝒯 ∧ snd tu ∈ 𝒰 ∧ fst tu ⌣ snd tu))
        else {}

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

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

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

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

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

abbreviation *Con* (**infix** $\langle\{\!|\frown|\!\}\rangle$ *50*)
where $\mathcal{T}\ \{\!|\frown|\!\}\ \mathcal{U} \equiv \mathcal{T}\ \{\!|\backslash|\!\}\ \mathcal{U} \neq \{\}$

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

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

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

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

```
  using N.composite-closed-left ‹R.seq v (w \ v)› vv' by blast
obtain x' where x': R.composite-of v' (w \ v) x'
  using x vv' N.composite-closed-left
  by (metis N.Cong₀-implies-Cong N.Cong₀-imp-coinitial N.Cong-imp-arr(1)
      R.composable-def R.composable-imp-seq R.con-implies-arr(2)
      R.seq-def R.sources-resid R.arr-resid-iff-con)
have ∗: t' \ x ≈₀ t \ x'
  by (metis N.coherent' N.composite-closed N.forward-stable R.con-imp-coinitial
      R.targets-composite-of 3 R.con-sym R.sources-composite-of vv' ww' x x')
obtain y where y: R.composite-of w (v \ w) y
  using x vv' ww'
  by (metis R.arr-resid-iff-con R.composable-def R.composable-imp-seq
      R.con-imp-coinitial R.seq-def R.sources-resid N.elements-are-arr
      N.forward-stable N.composite-closed-left)
obtain y' where y': R.composite-of w' (v \ w) y'
  using y ww'
  by (metis N.Cong₀-imp-coinitial N.Cong-closure-props(3) N.Cong-imp-arr(1)
      R.composable-def R.composable-imp-seq R.con-implies-arr(2) R.seq-def
      R.sources-resid N.composite-closed-left R.arr-resid-iff-con)
have ∗∗: u' \ y ≈₀ u \ y'
  by (metis N.composite-closed N.forward-stable R.con-imp-coinitial R.targets-composite-of
      ‹w ⌢ v› N.coherent' R.sources-composite-of vv' ww' y y')
have 4: x ∈ ℜ ∧ y ∈ ℜ
  using x y vv' ww' ∗ ∗∗
  by (metis 3 N.composite-closed N.forward-stable R.con-imp-coinitial R.con-sym)
have t \ x' ⌢ u \ y'
proof −
  have t \ x' ≈₀ t' \ x
    using ∗ by simp
  moreover have t' \ x ⌢ u' \ y
  proof −
    have t' \ x ⌢ u' \ x
      using t'u' vv' ww' 4 ∗
      by (metis N.Resid-along-normal-preserves-reflects-con N.elements-are-arr
          R.coinitial-iff R.con-imp-coinitial R.arr-resid-iff-con)
    moreover have u' \ x ≈₀ u' \ y
      using ww' x y
      by (metis 4 N.Cong₀-imp-coinitial N.Cong₀-imp-con N.Cong₀-transitive
          N.coherent' N.factor-closed(2) R.sources-composite-of
          R.targets-composite-of R.targets-resid-sym)
    ultimately show ?thesis
      using N.Cong₀-subst-right by blast
  qed
  moreover have u' \ y ≈₀ u \ y'
    using ∗∗ R.con-sym by simp
  ultimately show ?thesis
    using N.Cong₀-subst-Con by auto
qed
moreover have x' ∈ ℜ ∧ y' ∈ ℜ
```

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

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

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

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

let *?w-u* = *w* \ *u* **and** *?w'-v* = *w'* \ *v*
have *w-tx-in-𝔑*: *?w-tx* ∈ 𝔑
using *tx ww' xx' R.con-composite-of-iff* [*of t x tx w*]
by (*metis* (*full-types*) *N.Cong₀-composite-of-arr-normal N.Cong₀-subst-left*(*1*)
N.forward-stable R.null-is-zero(*2*) *R.con-imp-coinitial R.conI R.con-sym*)
have *w'-t'x'-in-𝔑*: *?w'-t'x'* ∈ 𝔑
using *t'x' ww' xx' R.con-composite-of-iff* [*of t' x' t'x' w'*]
by (*metis* (*full-types*) *N.Cong₀-composite-of-arr-normal N.Cong₀-subst-left*(*1*)
R.con-sym N.forward-stable R.null-is-zero(*2*) *R.con-imp-coinitial R.conI*)
have *2*: *?tx-w* ≈₀ *?t'x'-w'*
proof −
have *?tx-w* ≈₀ *t* \ *w*
using *t'x' tx ww' xx' N.Cong₀-composite-of-arr-normal* [*of t x tx*] *N.Cong₀-subst-left*(*2*)
by (*metis N.Cong₀-transitive R.conI*)
also have *t* \ *w* ≈₀ *t'* \ *w'*
using *ww'* **by** *blast*
also have *t'* \ *w'* ≈₀ *?t'x'-w'*
using *t'x' tx ww' xx' N.Cong₀-composite-of-arr-normal* [*of t' x' t'x'*] *N.Cong₀-subst-left*(*2*)
by (*metis N.Cong₀-transitive R.conI*)
finally show *?thesis* **by** *blast*
qed
obtain *z* **where** *z*: *R.composite-of ?tx-w* (*?t'x'-w'* \ *?tx-w*) *z*
by (*metis 2 R.arr-resid-iff-con R.con-implies-arr*(*2*) *N.elements-are-arr*
N.composite-closed-right R.seqI(*1*) *R.sources-resid*)
obtain *z'* **where** *z'*: *R.composite-of ?t'x'-w'* (*?tx-w* \ *?t'x'-w'*) *z'*
by (*metis 2 R.arr-resid-iff-con R.con-implies-arr*(*2*) *N.elements-are-arr*
N.composite-closed-right R.seqI(*1*) *R.sources-resid*)
have *3*: *z* ≈₀ *z'*
using *2 N.diamond-commutes-upto-Cong₀ N.Cong₀-imp-con z z'* **by** *blast*
have *R.targets z* = *R.targets z'*
by (*metis R.targets-resid-sym z z' R.targets-composite-of R.conI*)
have *Con-z-uw*: *z* ⌢ *?u-w*
proof −
have *?tx-w* ⌢ *?u-w*
by (*meson 3 N.Cong₀-composite-of-arr-normal N.Cong₀-subst-left*(*1*)
R.bounded-imp-con R.con-implies-arr(*1*) *R.con-imp-coinitial*
N.resid-along-elem-preserves-con tu tx ww' xx' z z' R.arr-resid-iff-con)
thus *?thesis*
using *2 N.Cong₀-composite-of-arr-normal N.Cong₀-subst-left*(*1*) *z* **by** *blast*
qed
moreover have *Con-z'-vw'*: *z'* ⌢ *?v-w'*
proof −
have *?t'x'-w'* ⌢ *?v-w'*
by (*meson 3 N.Cong₀-composite-of-arr-normal N.Cong₀-subst-left*(*1*)
R.bounded-imp-con t'v t'x' ww' xx' z z' R.con-imp-coinitial
N.resid-along-elem-preserves-con R.arr-resid-iff-con R.con-implies-arr(*1*))
thus *?thesis*
by (*meson 2 N.Cong₀-composite-of-arr-normal N.Cong₀-subst-left*(*1*) *z'*)
qed

moreover have *Con-z-vw'*: $z \frown ?v\text{-}w'$
using *3 Con-z'-vw' N.Cong$_0$-subst-left(1)* **by** *blast*
moreover have $*$: $?u\text{-}w \setminus z \frown ?v\text{-}w' \setminus z$
proof −
obtain y **where** y: *R.composite-of* $(w \setminus tx)$ $(?t'x'\text{-}w' \setminus ?tx\text{-}w)$ y
by (*metis 2 R.arr-resid-iff-con R.composable-def R.composable-imp-seq R.con-imp-coinitial N.elements-are-arr N.composite-closed-right R.seq-def R.targets-resid-sym ww' z N.forward-stable*)
obtain y' **where** y': *R.composite-of* $(w' \setminus t'x')$ $(?tx\text{-}w \setminus ?t'x'\text{-}w')$ y'
by (*metis 2 R.arr-resid-iff-con R.composable-def R.composable-imp-seq R.con-imp-coinitial N.elements-are-arr N.composite-closed-right R.targets-resid-sym ww' z' R.seq-def N.forward-stable*)
have *y-comp*: *R.composite-of* $(w \setminus tx)$ $((t'x' \setminus w') \setminus (tx \setminus w))$ y
using y **by** *simp*
have *y-in-normal*: $y \in \mathfrak{N}$
by (*metis 2 Con-z-uw R.arr-iff-has-source R.arr-resid-iff-con N.composite-closed R.con-imp-coinitial R.con-implies-arr(1) N.forward-stable R.sources-composite-of ww' y-comp z*)
have *y-coinitial*: *R.coinitial* y $(u \setminus tx)$
using y *R.arr-composite-of R.sources-composite-of*
apply (*intro R.coinitialI*)
apply *auto*
apply (*metis N.Cong$_0$-composite-of-arr-normal N.Cong$_0$-subst-right(1) R.composite-of-cancel-left R.con-sym R.not-ide-null R.null-is-zero(2) R.sources-resid R.conI tu tx xx'*)
by (*metis R.arr-iff-has-source R.not-arr-null R.sources-resid empty-iff R.conI*)
have *y-con*: $y \frown u \setminus tx$
using *y-in-normal y-coinitial*
by (*metis R.coinitial-iff N.elements-are-arr N.forward-stable R.arr-resid-iff-con*)
have *A*: $?u\text{-}w \setminus z \sim (u \setminus tx) \setminus y$
proof −
have $(u \setminus tx) \setminus y \sim ((u \setminus tx) \setminus (w \setminus tx)) \setminus (?t'x'\text{-}w' \setminus ?tx\text{-}w)$
using *y-comp y-con R.resid-composite-of(3) [of w \ tx ?t'x'-w' \ ?tx-w y u \ tx]*
by *simp*
also have $((u \setminus tx) \setminus (w \setminus tx)) \setminus (?t'x'\text{-}w' \setminus ?tx\text{-}w) \sim ?u\text{-}w \setminus z$
by (*metis Con-z-uw R.resid-composite-of(3) z R.cube*)
finally show *?thesis* **by** *blast*
qed
have *y'-comp*: *R.composite-of* $(w' \setminus t'x')$ $(?tx\text{-}w \setminus ?t'x'\text{-}w')$ y'
using y' **by** *simp*
have *y'-in-normal*: $y' \in \mathfrak{N}$
by (*metis 2 Con-z'-vw' R.arr-iff-has-source R.arr-resid-iff-con N.composite-closed R.con-imp-coinitial R.con-implies-arr(1) N.forward-stable R.sources-composite-of ww' y'-comp z'*)
have *y'-coinitial*: *R.coinitial* y' $(v \setminus t'x')$
using y' *R.coinitial-def*
by (*metis Con-z'-vw' R.arr-resid-iff-con R.composite-ofE R.con-imp-coinitial*

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

```
      using par-y-y' N.coherent R.coinitial-iff y'-coinitial y'-in-normal y-in-normal
      by presburger
    ultimately show ?thesis
      using N.Cong₀-subst-right(1) by blast
  qed
  hence ?u-w \ z ⌣ ?v-w' \ z'
  proof −
    have (u \ tx) \ y ∼ ?u-w \ z
      using A by simp
    moreover have (u \ tx) \ y ⌣ (v \ t'x') \ y'
      using E by blast
    moreover have (v \ t'x') \ y' ∼ ?v-w' \ z'
      using B R.cong-symmetric by blast
    moreover have R.sources ((u \ w) \ z) = R.sources ((v \ w') \ z')
      by (simp add: Con-z'-vw' Con-z-uw R.con-sym ‹R.targets z = R.targets z'›)
    ultimately show ?thesis
      by (meson N.Cong₀-subst-Con N.ide-closed)
  qed
  moreover have ?v-w' \ z' ≈ ?v-w' \ z
    by (meson 3 Con-z-vw' N.CongI N.Cong₀-subst-right(2) R.con-sym)
  moreover have R.sources ((v \ w') \ z) = R.sources ((u \ w) \ z)
    by (metis R.con-implies-arr(1) R.sources-resid calculation(1) calculation(2)
          N.Cong-imp-arr(2) R.arr-resid-iff-con)
  ultimately show ?thesis
    by (metis N.Cong-reflexive N.Cong-subst(1) R.con-implies-arr(1))
qed
ultimately have **: ?v-w' \ z ⌣ ?u-w \ z ∧
                    (?v-w' \ z) \ (?u-w \ z) = (?v-w' \ ?u-w) \ (z \ ?u-w)
  by (meson R.con-sym R.cube)
have Cong-t-z: t ≈ z
  by (metis 2 N.Cong₀-composite-of-arr-normal N.Cong-closure-props(2−3)
      N.Cong-closure-props(4) N.Cong-imp-arr(2) R.coinitial-iff R.con-imp-coinitial
      tx ww' xx' z R.arr-resid-iff-con)
have Cong-u-uw: u ≈ ?u-w
  by (meson Con-z-uw N.Cong-closure-props(4) R.coinitial-iff R.con-imp-coinitial
      ww' R.arr-resid-iff-con)
have Cong-v-vw': v ≈ ?v-w'
  by (meson Con-z-vw' N.Cong-closure-props(4) R.coinitial-iff ww' R.con-imp-coinitial
      R.arr-resid-iff-con)
have T: N.is-Cong-class T ∧ z ∈ T
  by (metis (no-types, lifting) Cong-t-z N.Cong-class-eqI N.Cong-class-is-nonempty
    N.Cong-class-memb-Cong-rep N.Cong-class-rep N.Cong-imp-arr(2) N.arr-in-Cong-class
      tu assms Con-char)
have U: N.is-Cong-class U ∧ ?u-w ∈ U
  by (metis Con-char Con-z-uw Cong-u-uw Int-iff N.Cong-class-eqI' N.Cong-class-eqI
      N.arr-in-Cong-class R.con-implies-arr(2) N.is-Cong-classI tu assms empty-iff)
have V: N.is-Cong-class V ∧ ?v-w' ∈ V
  by (metis Con-char Con-z-vw' Cong-v-vw' Int-iff N.Cong-class-eqI' N.Cong-class-eqI
      N.arr-in-Cong-class R.con-implies-arr(2) N.is-Cong-classI t'v assms empty-iff)
```

```
  show (𝒱 ⫽ 𝒯) ⫽ (𝒰 ⫽ 𝒯) = (𝒱 ⫽ 𝒰) ⫽ (𝒯 ⫽ 𝒰)
  proof −
    have (𝒱 ⫽ 𝒯) ⫽ (𝒰 ⫽ 𝒯) = ⫽(?v-w' \ z) \ (?u-w \ z)⫽
      using 𝒯 𝒰 𝒱 ∗ Resid-by-members
     by (metis ∗∗ Con-char N.arr-in-Cong-class R.arr-resid-iff-con assms R.con-implies-arr(2))
    moreover have (𝒱 ⫽ 𝒰) ⫽ (𝒯 ⫽ 𝒰) = ⫽(?v-w' \ ?u-w) \ (z \ ?u-w)⫽
      using Resid-by-members [of 𝒱 𝒰 ?v-w' ?u-w] Resid-by-members [of 𝒯 𝒰 z ?u-w]
            Resid-by-members [of 𝒱 ⫽ 𝒰 𝒯 ⫽ 𝒰 ?v-w' \ ?u-w z \ ?u-w]
      by (metis 𝒯 𝒰 𝒱 ∗ ∗∗ N.arr-in-Cong-class R.con-implies-arr(2) N.is-Cong-classI
          R.resid-reflects-con R.arr-resid-iff-con)
    ultimately show ?thesis
      using ∗∗ by simp
  qed
qed

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

lemma is-residuation:
shows residuation Resid
  ..

lemma arr-char:
shows arr 𝒯 ⟷ N.is-Cong-class 𝒯
  by (metis N.is-Cong-class-def arrI not-arr-null null-char N.Cong-class-memb-is-arr
      Con-char R.arrE arrE arr-resid conI)

lemma ide-char:
shows ide 𝒰 ⟷ arr 𝒰 ∧ 𝒰 ∩ 𝔑 ≠ {}
proof
  show ide 𝒰 ⟹ arr 𝒰 ∧ 𝒰 ∩ 𝔑 ≠ {}
    apply (elim ideE)
    by (metis Con-char N.Cong₀-reflexive Resid-by-members disjoint-iff null-char
        N.arr-in-Cong-class R.arrE R.arr-resid arr-resid conE)
  show arr 𝒰 ∧ 𝒰 ∩ 𝔑 ≠ {} ⟹ ide 𝒰
  proof −
    assume 𝒰: arr 𝒰 ∧ 𝒰 ∩ 𝔑 ≠ {}
    obtain u where u: R.arr u ∧ u ∈ 𝒰 ∩ 𝔑
      using 𝒰 arr-char
      by (metis IntI N.Cong-class-memb-is-arr disjoint-iff)
    show ?thesis
      by (metis IntD1 IntD2 N.Cong-class-eqI N.Cong-closure-props(4) N.arr-in-Cong-class
          N.is-Cong-classI Resid-by-members 𝒰 arrE arr-char disjoint-iff ideI
          N.Cong-class-eqI' R.arrE u)
  qed
qed

lemma ide-char':
```

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

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

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 $t \frown u$
by (*meson assms N.Cong-subst*(*1*) *N.is-Cong-classE con-char*$_{QCN}$)

sublocale *rts Resid*
proof
show *1*: $\bigwedge \mathcal{A}\ \mathcal{T}.\ [\![ide\ \mathcal{A};\ con\ \mathcal{T}\ \mathcal{A}]\!] \Longrightarrow \mathcal{T}\ \{\!|\backslash|\!\}\ \mathcal{A} = \mathcal{T}$
proof −
fix $\mathcal{A}\ \mathcal{T}$
assume $\mathcal{A}$: *ide* $\mathcal{A}$ **and** *con*: *con* $\mathcal{T}$ $\mathcal{A}$
obtain $t\ a$ **where** *ta*: $t \in \mathcal{T} \wedge a \in \mathcal{A} \wedge R.con\ t\ a \wedge \mathcal{T}\ \{\!|\backslash|\!\}\ \mathcal{A} = \{\!|t \setminus a|\!\}$
using *con con-char*$_{QCN}$ *Resid-by-members* **by** *auto*
have $a \in \mathfrak{N}$
using $\mathcal{A}$ *ta ide-char*′ **by** *auto*
hence $t \setminus a \approx t$
by (*meson N.Cong-closure-props*(*4*) *N.Cong-symmetric R.coinitialE R.con-imp-coinitial ta*)
thus $\mathcal{T}\ \{\!|\backslash|\!\}\ \mathcal{A} = \mathcal{T}$
using *ta*
by (*metis N.Cong-class-eqI N.Cong-class-memb-Cong-rep N.Cong-class-rep con con-char*$_{QCN}$)
qed
show $\bigwedge \mathcal{T}.$ *arr* $\mathcal{T} \Longrightarrow$ *ide* (*trg* $\mathcal{T}$)
by (*metis N.Cong*$_0$*-reflexive Resid-by-members disjoint-iff ide-char N.Cong-class-memb-is-arr N.arr-in-Cong-class N.is-Cong-class-def arr-char R.arrE R.arr-resid resid-arr-self*)
show $\bigwedge \mathcal{A}\ \mathcal{T}.\ [\![ide\ \mathcal{A};\ con\ \mathcal{A}\ \mathcal{T}]\!] \Longrightarrow ide\ (\mathcal{A}\ \{\!|\backslash|\!\}\ \mathcal{T})$
by (*metis 1 arrE arr-resid con-sym ideE ideI cube*)
show $\bigwedge \mathcal{T}\ \mathcal{U}.$ *con* $\mathcal{T}$ $\mathcal{U} \Longrightarrow \exists \mathcal{A}.$ *ide* $\mathcal{A} \wedge$ *con* $\mathcal{A}$ $\mathcal{T} \wedge$ *con* $\mathcal{A}$ $\mathcal{U}$
proof −
fix $\mathcal{T}\ \mathcal{U}$
assume $\mathcal{TU}$: *con* $\mathcal{T}$ $\mathcal{U}$
obtain $t\ u$ **where** *tu*: $\mathcal{T} = \{\!|t|\!\} \wedge \mathcal{U} = \{\!|u|\!\} \wedge t \frown u$
using $\mathcal{TU}$ *con-char*$_{QCN}$ *arr-char*
by (*metis N.Cong-class-memb-Cong-rep N.Cong-class-eqI N.Cong-class-rep*)
obtain a **where** a: $a \in$ *R.sources* t
using $\mathcal{TU}$ *tu R.con-implies-arr*(*1*) *R.arr-iff-has-source* **by** *blast*
have *ide* $\{\!|a|\!\} \wedge$ *con* $\{\!|a|\!\}$ $\mathcal{T} \wedge$ *con* $\{\!|a|\!\}$ $\mathcal{U}$
proof (*intro conjI*)

```
    have 2: a ∈ 𝔑
      using 𝒯𝒰 tu a arr-char N.ide-closed R.sources-def by force
    show 3: ide ⦇a⦈
      using 𝒯𝒰 tu 2 a ide-char arr-char con-char_QCN
      by (metis IntI N.arr-in-Cong-class N.is-Cong-classI empty-iff N.elements-are-arr)
    show con ⦇a⦈ 𝒯
      using 𝒯𝒰 tu 2 3 a ide-char arr-char con-char_QCN
      by (metis N.arr-in-Cong-class R.composite-of-source-arr
          R.composite-of-def R.prfx-implies-con R.con-implies-arr(1))
    show con ⦇a⦈ 𝒰
      using 𝒯𝒰 tu a ide-char arr-char con-char_QCN
      by (metis N.arr-in-Cong-class R.composite-of-source-arr R.con-prfx-composite-of
          N.is-Cong-classI R.con-implies-arr(1) R.con-implies-arr(2))
  qed
  thus ∃𝒜. ide 𝒜 ∧ con 𝒜 𝒯 ∧ con 𝒜 𝒰 by auto
qed
show ⋀𝒯 𝒰 𝒱. ⟦ide (𝒯 ⦇\⦈ 𝒰); con 𝒰 𝒱⟧ ⟹ con (𝒯 ⦇\⦈ 𝒰) (𝒱 ⦇\⦈ 𝒰)
proof −
  fix 𝒯 𝒰 𝒱
  assume 𝒯𝒰: ide (𝒯 ⦇\⦈ 𝒰)
  assume 𝒰𝒱: con 𝒰 𝒱
  obtain t u where tu: t ∈ 𝒯 ∧ u ∈ 𝒰 ∧ t ⌢ u ∧ 𝒯 ⦇\⦈ 𝒰 = ⦇t \ u⦈
    using 𝒯𝒰
    by (meson Resid-by-members ide-implies-arr con-char_QCN arr-resid-iff-con)
  obtain v u' where vu': v ∈ 𝒱 ∧ u' ∈ 𝒰 ∧ v ⌢ u' ∧ 𝒱 ⦇\⦈ 𝒰 = ⦇v \ u'⦈
    by (meson R.con-sym Resid-by-members 𝒰𝒱 con-char_QCN)
  have 1: u ≈ u'
    using 𝒰𝒱 tu vu'
    by (meson N.Cong-class-membs-are-Cong con-char_QCN)
  obtain w w' where ww': w ∈ 𝔑 ∧ w' ∈ 𝔑 ∧ u \ w ≈₀ u' \ w'
    using 1 by auto
  have 2: ((t \ u) \ (w \ u)) \ ((u' \ w') \ (u \ w)) ⌢
          ((v \ u') \ (w' \ u')) \ ((u \ w) \ (u' \ w'))
  proof −
    have ((t \ u) \ (w \ u)) \ ((u' \ w') \ (u \ w)) ∈ 𝔑
    proof −
      have t \ u ∈ 𝔑
        using tu N.arr-in-Cong-class R.arr-resid-iff-con 𝒯𝒰 ide-char' by blast
      hence (t \ u) \ (w \ u) ∈ 𝔑
        by (metis N.Cong-closure-props(4) N.forward-stable R.null-is-zero(2)
            R.con-imp-coinitial R.sources-resid N.Cong-imp-arr(2) R.arr-resid-iff-con
            tu ww' R.conI)
      thus ?thesis
        by (metis N.Cong-closure-props(4) N.normal-is-Cong-closed R.sources-resid
            R.targets-resid-sym N.elements-are-arr R.arr-resid-iff-con ww' R.conI)
    qed
    moreover have R.sources (((t \ u) \ (w \ u)) \ ((u' \ w') \ (u \ w))) =
                  R.sources (((v \ u') \ (w' \ u')) \ ((u \ w) \ (u' \ w')))
    proof −
```

```
      have R.sources (((t \ u) \ (w \ u)) \ ((u' \ w') \ (u \ w))) =
           R.targets ((u' \ w') \ (u \ w))
        using R.arr-resid-iff-con N.elements-are-arr R.sources-resid calculation by blast
      also have ... = R.targets ((u \ w) \ (u' \ w'))
        by (metis R.targets-resid-sym R.conI)
      also have ... = R.sources (((v \ u') \ (w' \ u')) \ ((u \ w) \ (u' \ w')))
        using R.arr-resid-iff-con N.elements-are-arr R.sources-resid
        by (metis N.Cong-closure-props(4) N.Cong-imp-arr(2) R.con-implies-arr(1)
            R.con-imp-coinitial N.forward-stable R.targets-resid-sym vu' ww')
      finally show ?thesis by simp
    qed
    ultimately show ?thesis
      by (metis (no-types, lifting) N.Cong₀-imp-con N.Cong-closure-props(4)
          N.Cong-imp-arr(2) R.arr-resid-iff-con R.con-imp-coinitial N.forward-stable
          R.null-is-zero(2) R.conI)
  qed
  moreover have t \ u ≈ ((t \ u) \ (w \ u)) \ ((u' \ w') \ (u \ w))
    by (metis (no-types, opaque-lifting) N.Cong-closure-props(4) N.Cong-transitive
        N.forward-stable R.arr-resid-iff-con R.con-imp-coinitial R.rts-axioms calculation
        rts.coinitial-iff ww')
  moreover have v \ u' ≈ ((v \ u') \ (w' \ u')) \ ((u \ w) \ (u' \ w'))
  proof -
    have w' \ u' ∈ 𝔑
      by (meson R.con-implies-arr(2) R.con-imp-coinitial N.forward-stable
          ww' N.Cong₀-imp-con R.arr-resid-iff-con)
    moreover have (u \ w) \ (u' \ w') ∈ 𝔑
      using ww' by blast
    ultimately show ?thesis
      by (meson 2 N.Cong-closure-props(2) N.Cong-closure-props(4) R.arr-resid-iff-con
          R.coinitial-iff R.con-imp-coinitial)
  qed
  ultimately show con (𝒯 ⟦\⟧ 𝒰) (𝒱 ⟦\⟧ 𝒰)
    using con-char_QCN N.Cong-class-def N.is-Cong-classI tu vu' R.arr-resid-iff-con
    by auto
 qed
qed

lemma is-rts:
shows rts Resid
 ..

sublocale extensional-rts Resid
proof
 fix 𝒯 𝒰
 assume 𝒯𝒰: cong 𝒯 𝒰
 show 𝒯 = 𝒰
 proof -
   obtain t u where tu: 𝒯 = ⟦t⟧ ∧ 𝒰 = ⟦u⟧ ∧ t ⌣ u
     by (metis Con-char N.Cong-class-eqI N.Cong-class-memb-Cong-rep N.Cong-class-rep
```

```
        TU ide-char not-arr-null null-char)
    have t ≈₀ u
    proof
      show t \ u ∈ 𝔑
        using tu TU Resid-by-members [of T U t u]
        by (metis (full-types) N.arr-in-Cong-class R.con-implies-arr(1-2)
            N.is-Cong-classI ide-char' R.arr-resid-iff-con subset-iff)
      show u \ t ∈ 𝔑
        using tu TU Resid-by-members [of U T u t] R.con-sym
        by (metis (full-types) N.arr-in-Cong-class R.con-implies-arr(1-2)
            N.is-Cong-classI ide-char' R.arr-resid-iff-con subset-iff)
    qed
    hence t ≈ u
      using N.Cong₀-implies-Cong by simp
    thus T = U
      by (simp add: N.Cong-class-eqI tu)
  qed
qed

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

lemma sources-char_QCN:
shows sources T = {A. arr T ∧ A = {a. ∃t a'. t ∈ T ∧ a' ∈ R.sources t ∧ a' ≈ a}}
proof -
  let ?A = {a. ∃t a'. t ∈ T ∧ a' ∈ R.sources t ∧ a' ≈ a}
  have 1: arr T ⟹ ide ?A
  proof (unfold ide-char', intro conjI)
    assume T: arr T
    show ?A ⊆ 𝔑
      using N.ide-closed N.normal-is-Cong-closed by blast
    show arr ?A
    proof -
      have N.is-Cong-class ?A
      proof
        show ?A ≠ {}
         by (metis (mono-tags, lifting) Collect-empty-eq N.Cong-class-def N.Cong-imp-arr(1)
             N.is-Cong-class-def N.sources-are-Cong R.arr-iff-has-source R.sources-def
             T arr-char mem-Collect-eq)
        show ⋀a a'. ⟦a ∈ ?A; a' ≈ a⟧ ⟹ a' ∈ ?A
          using N.Cong-transitive by blast
        show ⋀a a'. ⟦a ∈ ?A; a' ∈ ?A⟧ ⟹ a ≈ a'
        proof -
          fix a a'
          assume a: a ∈ ?A and a': a' ∈ ?A
          obtain t b where b: t ∈ T ∧ b ∈ R.sources t ∧ b ≈ a
            using a by blast
          obtain t' b' where b': t' ∈ T ∧ b' ∈ R.sources t' ∧ b' ≈ a'
```

```
        using a' by blast
      have b ≈ b'
        using T arr-char b b'
        by (meson IntD1 N.Cong-class-membs-are-Cong N.in-sources-respects-Cong)
      thus a ≈ a'
        by (meson N.Cong-symmetric N.Cong-transitive b b')
    qed
  qed
  thus ?thesis
    using arr-char by auto
 qed
 moreover have arr T ⟹ con T ?A
 proof −
  assume T: arr T
  obtain t a where a: t ∈ T ∧ a ∈ R.sources t
    using T arr-char
    by (metis N.Cong-class-is-nonempty R.arr-iff-has-source empty-subsetI
          N.Cong-class-memb-is-arr subsetI subset-antisym)
  have t ∈ T ∧ a ∈ {a. ∃ t a'. t ∈ T ∧ a' ∈ R.sources t ∧ a' ≈ a} ∧ t ⌣ a
    using a N.Cong-reflexive R.sources-def R.con-implies-arr(2) by fast
  thus ?thesis
    using T 1 arr-char con-char_QCN [of T ?A] by auto
 qed
 ultimately have arr T ⟹ ?A ∈ sources T
  using sources-def by blast
 thus ?thesis
  using 1 ide-char sources-char_WE by auto
qed

lemma targets-char_QCN:
shows targets T = {B. arr T ∧ B = T ⟦\⟧ T}
proof −
 have targets T = {B. ide B ∧ con (T ⟦\⟧ T) B}
  by (simp add: targets-def trg-def)
 also have ... = {B. arr T ∧ ide B ∧ (∃ t u. t ∈ T ⟦\⟧ T ∧ u ∈ B ∧ t ⌣ u)}
  using arr-resid-iff-con con-char_QCN arr-char arr-def by auto
 also have ... = {B. arr T ∧ ide B ∧
                  (∃ t t' b u. t ∈ T ∧ t' ∈ T ∧ t ⌣ t' ∧ b ∈ ⟦t \ t'⟧ ∧ u ∈ B ∧ b ⌣ u)}
  apply auto
   apply (metis (full-types) Resid-by-members cong-char not-ide-null null-char Con-char)
  by (metis Resid-by-members arr-char)
 also have ... = {B. arr T ∧ ide B ∧
                  (∃ t t' b. t ∈ T ∧ t' ∈ T ∧ t ⌣ t' ∧ b ∈ ⟦t \ t'⟧ ∧ b ∈ B)}
 proof −
  have ⋀B t t' b. ⟦arr T; ide B; t ∈ T; t' ∈ T; t ⌣ t'; b ∈ ⟦t \ t'⟧⟧
                  ⟹ (∃ u. u ∈ B ∧ b ⌣ u) ⟷ b ∈ B
  proof −
   fix B t t' b
```

assume $\mathcal{T}$: *arr* $\mathcal{T}$ **and** $\mathcal{B}$: *ide* $\mathcal{B}$ **and** t: $t \in \mathcal{T}$ **and** t': $t' \in \mathcal{T}$
and tt': $t \frown t'$ **and** b: $b \in \{\!|t \setminus t'|\!\}$
have *0*: $b \in \mathfrak{N}$
by (*metis Resid-by-members* $\mathcal{T}$ *b ide-char' ide-trg arr-char subsetD t t' trg-def tt'*)
show $(\exists u.\ u \in \mathcal{B} \wedge b \frown u) \longleftrightarrow b \in \mathcal{B}$
using *0*
by (*meson N.Cong-closure-props*(*3*) *N.forward-stable N.elements-are-arr*
$\mathcal{B}$ *arr-char R.con-imp-coinitial N.is-Cong-classE ide-char' R.arrE*
R.con-sym subsetD)
qed
thus *?thesis*
using *ide-char arr-char*
by (*metis* (*no-types*, *lifting*))
qed
also have ... = $\{\mathcal{B}.\ arr\ \mathcal{T} \wedge ide\ \mathcal{B} \wedge (\exists t\ t'.\ t \in \mathcal{T} \wedge t' \in \mathcal{T} \wedge t \frown t' \wedge \{\!|t \setminus t'|\!\} \subseteq \mathcal{B})\}$
proof −
have $\bigwedge \mathcal{B}\ t\ t'\ b.\ [\![arr\ \mathcal{T};\ ide\ \mathcal{B};\ t \in \mathcal{T};\ t' \in \mathcal{T};\ t \frown t']\!]$
$\Longrightarrow (\exists b.\ b \in \{\!|t \setminus t'|\!\} \wedge b \in \mathcal{B}) \longleftrightarrow \{\!|t \setminus t'|\!\} \subseteq \mathcal{B}$
using *ide-char arr-char*
apply (*intro iffI*)
apply (*metis IntI N.Cong-class-eqI' R.arr-resid-iff-con N.is-Cong-classI empty-iff*
set-eq-subset)
by (*meson N.arr-in-Cong-class R.arr-resid-iff-con subsetD*)
thus *?thesis*
using *ide-char arr-char*
by (*metis* (*no-types*, *lifting*))
qed
also have ... = $\{\mathcal{B}.\ arr\ \mathcal{T} \wedge ide\ \mathcal{B} \wedge \mathcal{T}\ \{\!|\setminus|\!\}\ \mathcal{T} \subseteq \mathcal{B}\}$
using *arr-char ide-char Resid-by-members* [*of* $\mathcal{T}$ $\mathcal{T}$]
by (*metis* (*no-types*, *opaque-lifting*) *arrE con-char*$_{QCN}$)
also have ... = $\{\mathcal{B}.\ arr\ \mathcal{T} \wedge \mathcal{B} = \mathcal{T}\ \{\!|\setminus|\!\}\ \mathcal{T}\}$
by (*metis* (*no-types*, *lifting*) *arr-has-un-target calculation con-ide-are-eq*
cong-reflexive mem-Collect-eq targets-def trg-def)
finally show *?thesis* **by** *blast*
qed

lemma *src-char*$_{QCN}$:
shows *src* $\mathcal{T} = \{a.\ arr\ \mathcal{T} \wedge (\exists t\ a'.\ t \in \mathcal{T} \wedge a' \in R.sources\ t \wedge a' \approx a)\}$
using *sources-char*$_{QCN}$ [*of* $\mathcal{T}$]
by (*simp add*: *null-char src-def*)

lemma *trg-char*$_{QCN}$:
shows *trg* $\mathcal{T} = \mathcal{T}\ \{\!|\setminus|\!\}\ \mathcal{T}$
unfolding *trg-def* **by** *blast*

Quotient Map

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

sublocale *quot*: *simulation-to-extensional-rts resid Resid quot*
proof
show $\bigwedge t.\ \neg\ R.arr\ t \Longrightarrow \lfloor\!\lfloor t \rfloor\!\rfloor = null$
using *N.Cong-class-def N.Cong-imp-arr*(*1*) *null-char* **by** *force*
show $\bigwedge t\ u.\ t \frown u \Longrightarrow con\ \lfloor\!\lfloor t \rfloor\!\rfloor\ \lfloor\!\lfloor u \rfloor\!\rfloor$
by (*meson N.arr-in-Cong-class N.is-Cong-classI R.con-implies-arr*(*1*−*2*) *con-char*$_{QCN}$)
show $\bigwedge t\ u.\ t \frown u \Longrightarrow \lfloor\!\lfloor t \setminus u \rfloor\!\rfloor = \lfloor\!\lfloor t \rfloor\!\rfloor\ \lfloor\!\lfloor \setminus \rfloor\!\rfloor\ \lfloor\!\lfloor u \rfloor\!\rfloor$
by (*metis N.arr-in-Cong-class N.is-Cong-classI R.con-implies-arr*(*1*−*2*) *Resid-by-members*)
qed

lemma *quotient-is-simulation*:
shows *simulation resid Resid quot*
..

lemma *ide-quot-normal*:
assumes $t \in \mathfrak{N}$
shows *ide* (*quot t*)
using *assms*
by (*metis IntI N.arr-in-Cong-class N.elements-are-arr empty-iff*
quot.preserves-reflects-arr ide-char)

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

lemma *is-couniversal*:
assumes *extensional-rts B*
and *simulation resid B F*
and $\bigwedge t.\ t \in \mathfrak{N} \Longrightarrow$ *residuation.ide B* (*F t*)
shows $\exists!F'.$ *simulation Resid B F'* $\wedge$ $F' \circ$ *quot* $= F$
proof −
interpret *B*: *extensional-rts B*
using *assms*(*1*) *simulation.axioms*(*2*) **by** *blast*
interpret *F*: *simulation resid B F*
using *assms* **by** *blast*
have *1*: $\bigwedge t\ u.\ t \approx u \Longrightarrow F\ t = F\ u$
proof −
fix *t u*
assume *Cong*: $t \approx u$
obtain *v w* **where** *vw*: $v \in \mathfrak{N} \wedge w \in \mathfrak{N} \wedge t \setminus v \approx_0 u \setminus w$
using *Cong* **by** *blast*
have *B.cong* (*F t*) (*F u*)
by (*metis assms*(*3*) *vw B.cong-char F.preserves-reflects-arr F.preserves-resid*
N.elements-are-arr R.arr-resid-iff-con B.arr-resid-iff-con B.resid-arr-ide)
thus $F\ t = F\ u$
using *B.extensionality* **by** *blast*
qed
let $?F' = \lambda\mathcal{T}.$ *if arr* $\mathcal{T}$ *then F* (*N.Cong-class-rep* $\mathcal{T}$) *else B.null*
interpret F': *simulation Resid B* $?F'$
proof

```
    show ⋀𝒯. ¬ arr 𝒯 ⟹ ?F' 𝒯 = B.null
      by argo
    fix 𝒯 𝒰
    assume con: con 𝒯 𝒰
    show B.con (?F' 𝒯) (?F' 𝒰)
      using con
      by (metis (full-types) 1 F.preserves-con N.Cong-class-memb-Cong-rep arr-char
          con-char_QCN)
    show ?F' (𝒯 ⦇\⦈ 𝒰) = B (?F' 𝒯) (?F' 𝒰)
    proof −
      have 2: N.is-Cong-class 𝒯 ∧ N.is-Cong-class 𝒰
        using con con-char_QCN by auto
      obtain t u where tu: t ∈ 𝒯 ∧ u ∈ 𝒰 ∧ t ⌢ u
        using con con-char_QCN by force
      have ?F' (𝒯 ⦇\⦈ 𝒰) = ?F' ⦇t \ u⦈
        using tu 2 Resid-by-members by force
      also have ... = F (t \ u)
        by (metis tu N.Cong-class-memb-Cong-rep N.arr-in-Cong-class N.is-Cong-classI
            R.arr-resid ‹⋀y x. x ≈ y ⟹ F x = F y› quot.preserves-reflects-arr)
      also have ... = B (F t) (F u)
        by (simp add: tu)
      also have ... = B (?F' 𝒯) (?F' 𝒰)
        by (metis (full-types) tu 1 2 N.Cong-class-memb-Cong-rep con
            con-implies-arr(1−2))
      finally show ?thesis by blast
    qed
  qed
  have simulation Resid B ?F' ∧ ?F' ∘ quot = F
  proof −
    have ?F' ∘ quot = F
    proof
      fix t
      have ?F' (quot t) = F t
        by (metis 1 F.extensionality N.Cong-class-memb-Cong-rep N.arr-in-Cong-class
            arr-char quot.preserves-reflects-arr)
      thus (?F' ∘ quot) t = F t
        by auto
    qed
    thus ?thesis
      using F'.simulation-axioms by blast
  qed
  moreover have ⋀F''. ⟦simulation Resid B F''; F'' ∘ quot = F⟧ ⟹ F'' = ?F'
    using simulation.extensionality arr-char by force
  ultimately show ?thesis by blast
qed

definition ext-to-quotient
where ext-to-quotient B F ≡ THE F'. simulation Resid B F' ∧ F' ∘ 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 −
have *simulation Resid B* (*ext-to-quotient B F*) $\wedge$ *ext-to-quotient B F* $\circ$ *quot* = *F*
unfolding *ext-to-quotient-def*
using *assms is-couniversal* [*of B F*]
theI' [*of* $\lambda F'.$ *simulation* ($\{\!|\backslash|\!\}$) *B F'* $\wedge$ *F'* $\circ$ *quot* = *F*]
by *fastforce*
thus *simulation Resid B* (*ext-to-quotient B F*) **and** *ext-to-quotient B F* $\circ$ *quot* = *F*
by *auto*
qed

end

2.3.7 Identities form a Coherent Normal Sub-RTS

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

context *rts*
begin

interpretation *normal-sub-rts resid* ‹*Collect ide*›
proof
show $\bigwedge t.\ t \in$ *Collect ide* $\Longrightarrow$ *arr t*
by *blast*
show *1*: $\bigwedge a.$ *ide a* $\Longrightarrow a \in$ *Collect ide*
by *blast*
show $\bigwedge u\ t.$ ⟦*u* ∈ *Collect ide*; *coinitial t u*⟧ ⟹ *u* \ *t* ∈ *Collect ide*
by (*metis 1 CollectD arr-def coinitial-iff*
con-sym in-sourcesE in-sourcesI resid-ide-arr)
show $\bigwedge u\ t.$ ⟦*u* ∈ *Collect ide*; *t* \ *u* ∈ *Collect ide*⟧ ⟹ *t* ∈ *Collect ide*
using *ide-backward-stable* **by** *blast*
show $\bigwedge u\ t.$ ⟦*u* ∈ *Collect ide*; *seq u t*⟧ ⟹ ∃ *v*. *composite-of u t v*
by (*metis composite-of-source-arr ide-def in-sourcesI mem-Collect-eq seq-def*
resid-source-in-targets)
show $\bigwedge u\ t.$ ⟦*u* ∈ *Collect ide*; *seq t u*⟧ ⟹ ∃ *v*. *composite-of t u v*
by (*metis arrE composite-of-arr-target in-sourcesI seqE mem-Collect-eq*)
qed

lemma *identities-form-normal-sub-rts*:
shows *normal-sub-rts resid* (*Collect ide*)
..

interpretation *coherent-normal-sub-rts resid* ‹*Collect ide*›
apply *unfold-locales*
by (*metis CollectD Cong*$_0$*-reflexive Cong-closure-props*(*4*) *Cong-imp-arr*(*2*)
arr-resid-iff-con resid-arr-ide)

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

lemma *Cong-iff-cong*:
shows *Cong t u* ⟷ *t* ∼ *u*
by (*metis CollectD Cong-def ide-closed resid-arr-ide*
Cong-closure-props(*3*) *Cong-imp-arr*(*2*) *arr-resid-iff-con*)

end

2.4 Paths

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

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

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

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

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

fun *Arr*
where *Arr* [] = *False*
| *Arr* [*t*] = *R.arr t*
| *Arr* (*t* # *T*) = (*R.arr t* ∧ *Arr T* ∧ *R.targets t* ⊆ *Srcs T*)

fun *Ide*
where *Ide* [] = *False*
| *Ide* [*t*] = *R.ide t*
| *Ide* (*t* # *T*) = (*R.ide t* ∧ *Ide T* ∧ *R.targets t* ⊆ *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 (*induct T*)
  show $Arr\ [] \Longrightarrow P\ []$
    using *Arr.simps*(*1*) **by** *blast*
  show $\bigwedge t\ U.\ [\![Arr\ U \Longrightarrow P\ U;\ Arr\ (t\ \#\ U)]\!] \Longrightarrow P\ (t\ \#\ U)$
    by (*metis assms Arr.simps*(*2*−*3*) *list.exhaust*)
qed

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 (*induct T*)
  show $Ide\ [] \Longrightarrow P\ []$
    using *Ide.simps*(*1*) **by** *blast*
  show $\bigwedge t\ T.\ [\![Ide\ T \Longrightarrow P\ T;\ Ide\ (t\ \#\ T)]\!] \Longrightarrow P\ (t\ \#\ T)$
    by (*metis assms Ide.simps*(*2*−*3*) *list.exhaust*)
qed

lemma *set-Arr-subset-arr*:
assumes *Arr T*
shows *set T* ⊆ *Collect R.arr*
  apply (*induct T rule*: *Arr-induct*)
  using *assms Arr.elims*(*2*) **by** *auto*

lemma *Arr-imp-arr-hd* [*simp*]:
assumes *Arr T*
shows *R.arr* (*hd T*)
  using *assms*
  by (*metis Arr.simps*(*1*) *CollectD hd-in-set set-Arr-subset-arr subset-code*(*1*))

lemma *Arr-imp-arr-last* [*simp*]:
assumes *Arr T*
shows *R.arr* (*last T*)
  using *assms*
  by (*metis Arr.simps*(*1*) *CollectD in-mono last-in-set set-Arr-subset-arr*)

lemma *Arr-imp-Arr-tl* [*simp*]:
assumes *Arr T* **and** *tl T* ≠ []
shows *Arr* (*tl T*)
  using *assms*
  by (*metis Arr.simps*(*3*) *list.exhaust-sel list.sel*(*2*))

lemma *set-Ide-subset-ide*:
assumes *Ide T*

shows *set T* ⊆ *Collect R.ide*
apply (*induct T rule*: *Ide-induct*)
using *assms* **by** *auto*

lemma *Ide-imp-Ide-hd* [*simp*]:
assumes *Ide T*
shows *R.ide* (*hd T*)
using *assms*
by (*metis Ide.simps*(*1*) *CollectD hd-in-set set-Ide-subset-ide subset-code*(*1*))

lemma *Ide-imp-Ide-last* [*simp*]:
assumes *Ide T*
shows *R.ide* (*last T*)
using *assms*
by (*metis Ide.simps*(*1*) *CollectD in-mono last-in-set set-Ide-subset-ide*)

lemma *Ide-imp-Ide-tl* [*simp*]:
assumes *Ide T* **and** *tl T* ≠ []
shows *Ide* (*tl T*)
using *assms*
by (*metis Ide.simps*(*3*) *list.exhaust-sel list.sel*(*2*))

lemma *Ide-implies-Arr*:
assumes *Ide T*
shows *Arr T*
apply (*induct T rule*: *Ide-induct*)
using *assms*
apply *auto*[*3*]
by (*metis Arr.elims*(*2*) *Arr.simps*(*3*) *R.ide-implies-arr*)

lemma *const-ide-is-Ide*:
shows ⟦*T* ≠ []; *R.ide* (*hd T*); *set T* ⊆ {*hd T*}⟧ ⟹ *Ide T*
apply (*induct T*)
apply *auto*[*2*]
by (*metis Ide.simps*(*2*−*3*) *R.ideE R.sources-resid Srcs.simps*(*2*−*3*) *empty-iff insert-iff*
list.exhaust-sel list.set-sel(*1*) *order-refl subset-singletonD*)

lemma *Ide-char*:
shows *Ide T* ⟷ *Arr T* ∧ *set T* ⊆ *Collect R.ide*
apply (*induct T*)
apply *auto*[*1*]
by (*metis Arr.simps*(*3*) *Ide.simps*(*2*−*3*) *Ide-implies-Arr empty-subsetI*
insert-subset list.simps(*15*) *mem-Collect-eq neq-Nil-conv set-empty*)

lemma *IdeI* [*intro*]:
assumes *Arr T* **and** *set T* ⊆ *Collect R.ide*
shows *Ide T*
using *assms Ide-char* **by** *force*

lemma *Arr-has-Src*:
shows *Arr T* $\Longrightarrow$ *Srcs T* $\neq$ {}
apply (*cases T*)
apply *simp*
by (*metis R.arr-iff-has-source Srcs.elims Arr.elims*(*2*) *list.distinct*(*1*) *list.sel*(*1*))

lemma *Arr-has-Trg*:
shows *Arr T* $\Longrightarrow$ *Trgs T* $\neq$ {}
using *R.arr-iff-has-target*
apply (*induct T*)
apply *simp*
by (*metis Arr.simps*(*2*) *Arr.simps*(*3*) *Trgs.simps*(*2*−*3*) *list.exhaust-sel*)

lemma *Srcs-are-ide*:
shows *Srcs T* $\subseteq$ *Collect R.ide*
apply (*cases T*)
apply *simp*
by (*metis* (*no-types*, *lifting*) *Srcs.elims list.distinct*(*1*) *mem-Collect-eq*
R.sources-def subsetI)

lemma *Trgs-are-ide*:
shows *Trgs T* $\subseteq$ *Collect R.ide*
apply (*induct T*)
apply *simp*
by (*metis R.arr-iff-has-target R.sources-resid Srcs.simps*(*2*) *Trgs.simps*(*2*−*3*)
Srcs-are-ide empty-subsetI list.exhaust R.arrE)

lemma *Srcs-are-con*:
assumes *a* $\in$ *Srcs T* **and** *a′* $\in$ *Srcs T*
shows *a* $\frown$ *a′*
using *assms*
by (*metis Srcs.elims empty-iff R.sources-are-con*)

lemma *Srcs-con-closed*:
assumes *a* $\in$ *Srcs T* **and** *R.ide a′* **and** *a* $\frown$ *a′*
shows *a′* $\in$ *Srcs T*
using *assms R.sources-con-closed*
apply (*cases T*, *auto*)
by (*metis Srcs.simps*(*2*−*3*) *neq-Nil-conv*)

lemma *Srcs-eqI*:
assumes *Srcs T* $\cap$ *Srcs T′* $\neq$ {}
shows *Srcs T* = *Srcs T′*
using *assms R.sources-eqI*
apply (*cases T*; *cases T′*)
apply *auto*
apply (*metis IntI Srcs.simps*(*2*−*3*) *empty-iff neq-Nil-conv*)
by (*metis Srcs.simps*(*2*−*3*) *assms neq-Nil-conv*)

lemma *Trgs-are-con*:
shows $[\![b \in \mathit{Trgs}\ T;\ b' \in \mathit{Trgs}\ T]\!] \Longrightarrow b \frown b'$
apply (*induct T*)
apply *auto*
by (*metis R.targets-are-con Trgs.simps*(2−3) *list.exhaust-sel*)

lemma *Trgs-con-closed*:
shows $[\![b \in \mathit{Trgs}\ T;\ \mathit{R.ide}\ b';\ b \frown b']\!] \Longrightarrow b' \in \mathit{Trgs}\ T$
apply (*induct T*)
apply *auto*
by (*metis R.targets-con-closed Trgs.simps*(2−3) *neq-Nil-conv*)

lemma *Trgs-eqI*:
assumes $\mathit{Trgs}\ T \cap \mathit{Trgs}\ T' \neq \{\}$
shows $\mathit{Trgs}\ T = \mathit{Trgs}\ T'$
using *assms Trgs-are-ide Trgs-are-con Trgs-con-closed* **by** *blast*

lemma *Srcs-simp*$_P$:
assumes *Arr T*
shows *Srcs T* = *R.sources* (*hd T*)
using *assms*
by (*metis Arr-has-Src Srcs.simps*(1) *Srcs.simps*(2) *Srcs.simps*(3) *list.exhaust-sel*)

lemma *Trgs-simp*$_P$:
shows *Arr T* $\Longrightarrow$ *Trgs T* = *R.targets* (*last T*)
apply (*induct T*)
apply *simp*
by (*metis Arr.simps*(3) *Trgs.simps*(2) *Trgs.simps*(3) *last-ConsL last-ConsR neq-Nil-conv*)

2.4.1 Residuation on Paths

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

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

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

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

fun *Residx1* (**infix** ‹$^*\backslash^1$› 70)

```
where [] *\^1 u = []
  | [t] *\^1 u = (if t ⌢ u then [t \ u] else [])
  | (t # T) *\^1 u =
      (if t ⌢ u ∧ T *\^1 (u \ t) ≠ [] then (t \ u) # T *\^1 (u \ t) else [])
```

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

```
function (sequential) Resid (infix ‹*\*› 70)
where [] *\* - = []
  | - *\* [] = []
  | [t] *\* [u] = (if t ⌢ u then [t \ u] else [])
  | [t] *\* (u # U) =
      (if t ⌢ u ∧ (t \ u) ^1\* U ≠ R.null then [(t \ u) ^1\* U] else [])
  | (t # T) *\* [u] =
      (if t ⌢ u ∧ T *\^1 (u \ t) ≠ [] then (t \ u) # (T *\^1 (u \ t)) else [])
  | (t # T) *\* (u # U) =
      (if t ⌢ u ∧ (t \ u) ^1\* U ≠ R.null ∧
          (T *\^1 (u \ t)) *\* (U *\^1 (t \ u)) ≠ []
       then (t \ u) ^1\* U # (T *\^1 (u \ t)) *\* (U *\^1 (t \ u))
       else [])
 by pat-completeness auto
```

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

```
lemma length-Residx1:
shows length (T *\^1 u) ≤ length T
proof (induct T arbitrary: u)
  show ⋀u. length ([] *\^1 u) ≤ length []
    by simp
  fix t T u
  assume ind: ⋀u. length (T *\^1 u) ≤ length T
  show length ((t # T) *\^1 u) ≤ length (t # T)
    using ind
    by (cases T, cases t ⌢ u, cases T *\^1 (u \ t)) auto
qed

termination Resid
proof (relation measure (λ(T, U). length T + length U))
  show wf (measure (λ(T, U). length T + length U))
    by simp
  fix t t' T u U
  have length ((t' # T) *\^1 (u \ t)) + length (U *\^1 (t \ u))
        < length (t # t' # T) + length (u # U)
    using length-Residx1
    by (metis add-less-le-mono impossible-Cons le-neq-implies-less list.size(4) trans-le-add1)
  thus 1: (((t' # T) *\^1 (u \ t), U *\^1 (t \ u)), t # t' # T, u # U)
          ∈ measure (λ(T, U). length T + length U)
    by simp
  show (((t' # T) *\^1 (u \ t), U *\^1 (t \ u)), t # t' # T, u # U)
```

```
        ∈ measure (λ(T, U). length T + length U)
    using 1 length-Residx1 by blast
  have length (T *\^1 (u \ t)) + length (U *\^1 (t \ u)) ≤ length T + length U
    using length-Residx1 by (simp add: add-mono)
  thus 2: ((T *\^1 (u \ t), U *\^1 (t \ u)), t # T, u # U)
          ∈ measure (λ(T, U). length T + length U)
    by simp
  show ((T *\^1 (u \ t), U *\^1 (t \ u)), t # T, u # U)
        ∈ measure (λ(T, U). length T + length U)
    using 2 length-Residx1 by blast
qed

lemma Resid1x-null:
shows R.null 1\* T = R.null
  apply (induct T)
   apply auto
  by (metis R.null-is-zero(1) Resid1x.simps(2-3) list.exhaust)

lemma Resid1x-ide:
shows ⟦R.ide a; a 1\* T ≠ R.null⟧ ⟹ R.ide (a 1\* T)
proof (induct T arbitrary: a)
  show ⋀a. a 1\* [] ≠ R.null ⟹ R.ide (a 1\* [])
    by simp
  fix a t T
  assume a: R.ide a
  assume ind: ⋀a. ⟦R.ide a; a 1\* T ≠ R.null⟧ ⟹ R.ide (a 1\* T)
  assume con: a 1\* (t # T) ≠ R.null
  have 1: a ⌢ t
    using con
    by (metis R.con-def Resid1x.simps(2-3) Resid1x-null list.exhaust)
  show R.ide (a 1\* (t # T))
    using a 1 con ind R.resid-ide-arr
    by (metis Resid1x.simps(2-3) list.exhaust)
qed


abbreviation Con  (infix ‹*⌢*› 50)
where T *⌢* U ≡ T *\* U ≠ []

lemma Con-sym1:
shows T *\^1 u ≠ [] ⟷ u 1\* T ≠ R.null
proof (induct T arbitrary: u)
  show ⋀u. [] *\^1 u ≠ [] ⟷ u 1\* [] ≠ R.null
    by simp
  show ⋀t T u. (⋀u. T *\^1 u ≠ [] ⟷ u 1\* T ≠ R.null)
                ⟹ (t # T) *\^1 u ≠ [] ⟷ u 1\* (t # T) ≠ R.null
  proof −
    fix t T u
    assume ind: ⋀u. T *\^1 u ≠ [] ⟷ u 1\* T ≠ R.null
```

```
    show (t # T) *\^1 u ≠ [] ⟷ u ^1\* (t # T) ≠ R.null
    proof
      show (t # T) *\^1 u ≠ [] ⟹ u ^1\* (t # T) ≠ R.null
        by (metis R.con-sym Resid1x.simps(2−3) Residx1.simps(2−3)
            ind neq-Nil-conv R.conE)
      show u ^1\* (t # T) ≠ R.null ⟹ (t # T) *\^1 u ≠ []
        using ind R.con-sym
        apply (cases T)
         apply auto
        by (metis R.conI Resid1x-null)
    qed
  qed
qed

lemma Con-sym-ind:
shows length T + length U ≤ n ⟹ T *⌢* U ⟷ U *⌢* T
proof (induct n arbitrary: T U)
  show ⋀T U. length T + length U ≤ 0 ⟹ T *⌢* U ⟷ U *⌢* T
    by simp
  fix n and T U :: 'a list
  assume ind: ⋀T U. length T + length U ≤ n ⟹ T *⌢* U ⟷ U *⌢* T
  assume 1: length T + length U ≤ Suc n
  show T *⌢* U ⟷ U *⌢* T
    using R.con-sym Con-sym1
      apply (cases T; cases U)
       apply auto[3]
  proof −
    fix t u T' U'
    assume T: T = t # T' and U: U = u # U'
    show T *⌢* U ⟷ U *⌢* T
    proof (cases T' = [])
      show T' = [] ⟹ T *⌢* U ⟷ U *⌢* T
        using T U Con-sym1 R.con-sym
        by (cases U') auto
      show T' ≠ [] ⟹ T *⌢* U ⟷ U *⌢* T
      proof (cases U' = [])
        show ⟦T' ≠ []; U' = []⟧ ⟹ T *⌢* U ⟷ U *⌢* T
          using T U R.con-sym Con-sym1
          by (cases T') auto
        show ⟦T' ≠ []; U' ≠ []⟧ ⟹ T *⌢* U ⟷ U *⌢* T
        proof −
          assume T': T' ≠ [] and U': U' ≠ []
          have 2: length (U' *\^1 (t \ u)) + length (T' *\^1 (u \ t)) ≤ n
          proof −
            have length (U' *\^1 (t \ u)) + length (T' *\^1 (u \ t))
                   ≤ length U' + length T'
              by (simp add: add-le-mono length-Residx1)
            also have ... ≤ length T' + length U'
              using T' add.commute not-less-eq-eq by auto
```

```
      also have ... ≤ n
        using 1 T U by simp
      finally show ?thesis by blast
    qed
    show T *⌢* U ⟷ U *⌢* T
    proof
      assume Con: T *⌢* U
      have 3: t ⌢ u ∧ T' *\¹ (u \ t) ≠ [] ∧ (t \ u) ¹\* U' ≠ R.null ∧
            (T' *\¹ (u \ t)) *\* (U' *\¹ (t \ u)) ≠ []
        using Con T U T' U' Con-sym1
        apply (cases T', cases U')
          apply simp-all
        by (metis Resid.simps(1) Resid.simps(6) neq-Nil-conv)
      hence u ⌢ t ∧ U' *\¹ (t \ u) ≠ [] ∧ (u \ t) ¹\* T' ≠ R.null
        using T' U' R.con-sym Con-sym1 by simp
      moreover have (U' *\¹ (t \ u)) *\* (T' *\¹ (u \ t)) ≠ []
        using 2 3 ind by simp
      ultimately show U *⌢* T
        using T U T' U'
        by (cases T'; cases U') auto
      next
      assume Con: U *⌢* T
      have 3: u ⌢ t ∧ U' *\¹ (t \ u) ≠ [] ∧ (u \ t) ¹\* T' ≠ R.null ∧
            (U' *\¹ (t \ u)) *\* (T' *\¹ (u \ t)) ≠ []
        using Con T U T' U' Con-sym1
        apply (cases T'; cases U')
           apply auto
         apply argo
        by force
      hence t ⌢ u ∧ T' *\¹ (u \ t) ≠ [] ∧ (t \ u) ¹\* U' ≠ R.null
        using T' U' R.con-sym Con-sym1 by simp
      moreover have (T' *\¹ (u \ t)) *\* (U' *\¹ (t \ u)) ≠ []
        using 2 3 ind by simp
      ultimately show T *⌢* U
        using T U T' U'
        by (cases T'; cases U') auto
    qed
  qed
 qed
qed
qed

lemma Con-sym:
shows T *⌢* U ⟷ U *⌢* T
  using Con-sym-ind by blast

lemma Residx1-as-Resid:
shows T *\¹ u = T *\* [u]
```

proof (*induct T*)
show [] $*\backslash^1$ *u* = [] $*\backslash^*$ [*u*] **by** *simp*
fix *t T*
assume *ind*: *T* $*\backslash^1$ *u* = *T* $*\backslash^*$ [*u*]
show (*t* # *T*) $*\backslash^1$ *u* = (*t* # *T*) $*\backslash^*$ [*u*]
by (*cases T*) *auto*
qed

lemma *Resid1x-as-Resid′*:
shows *t* $^1\backslash^*$ *U* = (*if* [*t*] $*\backslash^*$ *U* ≠ [] *then hd* ([*t*] $*\backslash^*$ *U*) *else R.null*)
proof (*induct U*)
show *t* $^1\backslash^*$ [] = (*if* [*t*] $*\backslash^*$ [] ≠ [] *then hd* ([*t*] $*\backslash^*$ []) *else R.null*) **by** *simp*
fix *u U*
assume *ind*: *t* $^1\backslash^*$ *U* = (*if* [*t*] $*\backslash^*$ *U* ≠ [] *then hd* ([*t*] $*\backslash^*$ *U*) *else R.null*)
show *t* $^1\backslash^*$ (*u* # *U*) = (*if* [*t*] $*\backslash^*$ (*u* # *U*) ≠ [] *then hd* ([*t*] $*\backslash^*$ (*u* # *U*)) *else R.null*)
using *Resid1x-null*
by (*cases U*) *auto*
qed

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

lemma *Con-rec*:
shows [*t*] $*\frown^*$ [*u*] ⟷ *t* ⌢ *u*
and *T* ≠ [] ⟹ *t* # *T* $*\frown^*$ [*u*] ⟷ *t* ⌢ *u* ∧ *T* $*\frown^*$ [*u* \ *t*]
and *U* ≠ [] ⟹ [*t*] $*\frown^*$ (*u* # *U*) ⟷ *t* ⌢ *u* ∧ [*t* \ *u*] $*\frown^*$ *U*
and ⟦*T* ≠ []; *U* ≠ []⟧ ⟹
t # *T* $*\frown^*$ *u* # *U* ⟷ *t* ⌢ *u* ∧ *T* $*\frown^*$ [*u* \ *t*] ∧ [*t* \ *u*] $*\frown^*$ *U* ∧
T $*\backslash^*$ [*u* \ *t*] $*\frown^*$ *U* $*\backslash^*$ [*t* \ *u*]
proof −
show [*t*] $*\frown^*$ [*u*] ⟷ *t* ⌢ *u*
by *simp*
show *T* ≠ [] ⟹ *t* # *T* $*\frown^*$ [*u*] ⟷ *t* ⌢ *u* ∧ *T* $*\frown^*$ [*u* \ *t*]
using *Residx1-as-Resid*
by (*cases T*) *auto*
show *U* ≠ [] ⟹ [*t*] $*\frown^*$ (*u* # *U*) ⟷ *t* ⌢ *u* ∧ [*t* \ *u*] $*\frown^*$ *U*
using *Resid1x-as-Resid′ Con-sym Con-sym1 Resid1x.simps*(*3*) *Residx1-as-Resid*
by (*cases U*) *auto*
show ⟦*T* ≠ []; *U* ≠ []⟧ ⟹
t # *T* $*\frown^*$ *u* # *U* ⟷ *t* ⌢ *u* ∧ *T* $*\frown^*$ [*u* \ *t*] ∧ [*t* \ *u*] $*\frown^*$ *U* ∧
T $*\backslash^*$ [*u* \ *t*] $*\frown^*$ *U* $*\backslash^*$ [*t* \ *u*]
using *Residx1-as-Resid Resid1x-as-Resid′ Con-sym1 Con-sym R.con-sym*
by (*cases T*; *cases U*) *auto*
qed

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

lemma *Resid1x-as-Resid*:
assumes [*t*] $*\backslash^*$ *U* ≠ []
shows [*t*] $*\backslash^*$ *U* = [*t* $^1\backslash^*$ *U*]
using *assms Con-rec*(*2*,*4*)

```
  apply (cases U; cases tl U)
    apply auto
  by argo+
```

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

```
lemma Resid-rec:
 shows [simp]: [t] *⌢* [u] ⟹ [t] *\* [u] = [t \ u]
 and ⟦T ≠ []; t # T *⌢* [u]⟧ ⟹ (t # T) *\* [u] = (t \ u) # (T *\* [u \ t])
 and ⟦U ≠ []; Con [t] (u # U)⟧ ⟹ [t] *\* (u # U) = [t \ u] *\* U
 and ⟦T ≠ []; U ≠ []; Con (t # T) (u # U)⟧ ⟹
     (t # T) *\* (u # U) = ([t \ u] *\* U) @ ((T *\* [u \ t]) *\* (U *\* [t \ u]))
 proof −
  show [t] *⌢* [u] ⟹ Resid [t] [u] = [t \ u]
    by (meson Resid.simps(3))
  show ⟦T ≠ []; t # T *⌢* [u]⟧ ⟹ (t # T) *\* [u] = (t \ u) # (T *\* [u \ t])
    using Residx1-as-Resid
    by (metis Residx1.simps(3) list.exhaust-sel)
  show 1: ⟦U ≠ []; [t] *⌢* u # U⟧ ⟹ [t] *\* (u # U) = [t \ u] *\* U
    by (metis Con-rec(3) Resid1x.simps(3) Resid1x-as-Resid list.exhaust)
  show ⟦T ≠ []; U ≠ []; t # T *⌢* u # U⟧ ⟹
       (t # T) *\* (u # U) = ([t \ u] *\* U) @ ((T *\* [u \ t]) *\* (U *\* [t \ u]))
  proof −
    assume T: T ≠ [] and U: U ≠ [] and Con: Con (t # T) (u # U)
    have tu: t ⌢ u
      using Con Con-rec by metis
    have (t # T) *\* (u # U) = ((t \ u) ¹\* U) # ((T *\¹ (u \ t)) *\* (U *\¹ (t \ u)))
      using T U Con tu
      by (cases T; cases U) auto
    also have ... = ([t \ u] *\* U) @ ((T *\* [u \ t]) *\* (U *\* [t \ u]))
      using T U Con tu Con-rec(4) Resid1x-as-Resid Residx1-as-Resid by force
    finally show ?thesis by simp
  qed
 qed
```

For consistent paths, residuation is length-preserving.

```
lemma length-Resid-ind:
shows ⟦length T + length U ≤ n; T *⌢* U⟧ ⟹ length (T *\* U) = length T
  apply (induct n arbitrary: T U)
   apply simp
 proof −
  fix n T U
  assume ind: ⋀T U. ⟦length T + length U ≤ n; T *⌢* U⟧
                    ⟹ length (T *\* U) = length T
  assume Con: T *⌢* U
  assume len: length T + length U ≤ Suc n
  show length (T *\* U) = length T
    using Con len ind Resid1x-as-Resid length-Cons Con-rec(2) Resid-rec(2)
    apply (cases T; cases U)
```

```
      apply auto
    apply (cases tl T = []; cases tl U = [])
      apply auto
     apply metis
    apply fastforce
  proof −
    fix t T' u U'
    assume T: T = t # T' and U: U = u # U'
    assume T': T' ≠ [] and U': U' ≠ []
    show length ((t # T') *\* (u # U')) = Suc (length T')
      using Con Con-rec(4) Con-sym Resid-rec(4) T T' U U' ind len by auto
  qed
qed

lemma length-Resid:
assumes T *⌢* U
shows length (T *\* U) = length T
  using assms length-Resid-ind by auto

lemma Con-initial-left:
shows t # T *⌢* U ⟹ [t] *⌢* U
  apply (induct U)
   apply simp
  by (metis Con-rec(1−4))

lemma Con-initial-right:
shows T *⌢* u # U ⟹ T *⌢* [u]
  apply (induct T)
    apply simp
  by (metis Con-rec(1−4))

lemma Resid-cons-ind:
shows ⟦T ≠ []; U ≠ []; length T + length U ≤ n⟧ ⟹
        (∀t. t # T *⌢* U ⟷ [t] *⌢* U ∧ T *⌢* U *\* [t]) ∧
        (∀u. T *⌢* u # U ⟷ T *⌢* [u] ∧ T *\* [u] *⌢* U) ∧
        (∀t. t # T *⌢* U ⟶ (t # T) *\* U = [t] *\* U @ T *\* (U *\* [t])) ∧
        (∀u. T *⌢* u # U ⟶ T *\* (u # U) = (T *\* [u]) *\* U)
proof (induct n arbitrary: T U)
  show ⋀T U. ⟦T ≠ []; U ≠ []; length T + length U ≤ 0⟧ ⟹
             (∀t. t # T *⌢* U ⟷ [t] *⌢* U ∧ T *⌢* U *\* [t]) ∧
             (∀u. T *⌢* u # U ⟷ T *⌢* [u] ∧ T *\* [u] *⌢* U) ∧
             (∀t. t # T *⌢* U ⟶ (t # T) *\* U = [t] *\* U @ T *\* (U *\* [t])) ∧
             (∀u. T *⌢* u # U ⟶ T *\* (u # U) = (T *\* [u]) *\* U)
    by simp
  fix n and T U :: 'a list
  assume ind: ⋀T U. ⟦T ≠ []; U ≠ []; length T + length U ≤ n⟧ ⟹
             (∀t. t # T *⌢* U ⟷ [t] *⌢* U ∧ T *⌢* U *\* [t]) ∧
             (∀u. T *⌢* u # U ⟷ T *⌢* [u] ∧ T *\* [u] *⌢* U) ∧
             (∀t. t # T *⌢* U ⟶ (t # T) *\* U = [t] *\* U @ T *\* (U *\* [t])) ∧
```

$(\forall u.\ T\ {}^*\!\frown^*\ u\ \#\ U \longrightarrow T\ {}^*\backslash^*\ (u\ \#\ U) = (T\ {}^*\backslash^*\ [u])\ {}^*\backslash^*\ U)$
assume T: $T \neq []$ **and** U: $U \neq []$
assume *len*: *length* T + *length* $U \leq$ *Suc* n
show $(\forall t.\ t\ \#\ T\ {}^*\!\frown^*\ U \longleftrightarrow [t]\ {}^*\!\frown^*\ U \wedge T\ {}^*\!\frown^*\ U\ {}^*\backslash^*\ [t]) \wedge$
$(\forall u.\ T\ {}^*\!\frown^*\ u\ \#\ U \longleftrightarrow T\ {}^*\!\frown^*\ [u] \wedge T\ {}^*\backslash^*\ [u]\ {}^*\!\frown^*\ U) \wedge$
$(\forall t.\ t\ \#\ T\ {}^*\!\frown^*\ U \longrightarrow (t\ \#\ T)\ {}^*\backslash^*\ U = [t]\ {}^*\backslash^*\ U\ @\ T\ {}^*\backslash^*\ (U\ {}^*\backslash^*\ [t])) \wedge$
$(\forall u.\ T\ {}^*\!\frown^*\ u\ \#\ U \longrightarrow T\ {}^*\backslash^*\ (u\ \#\ U) = (T\ {}^*\backslash^*\ [u])\ {}^*\backslash^*\ U)$
proof (*intro allI conjI iffI impI*)
fix t
show *1*: $t\ \#\ T\ {}^*\!\frown^*\ U \Longrightarrow (t\ \#\ T)\ {}^*\backslash^*\ U = [t]\ {}^*\backslash^*\ U\ @\ T\ {}^*\backslash^*\ (U\ {}^*\backslash^*\ [t])$
proof (*cases* U)
show $U = [] \Longrightarrow$ *?thesis*
using U **by** *simp*
fix $u\ U'$
assume U: $U = u\ \#\ U'$
assume *Con*: $t\ \#\ T\ {}^*\!\frown^*\ U$
show *?thesis*
proof (*cases* $U' = []$)
show $U' = [] \Longrightarrow$ *?thesis*
using T U *Con R.con-sym Con-rec(2) Resid-rec(2)* **by** *auto*
assume U': $U' \neq []$
have $(t\ \#\ T)\ {}^*\backslash^*\ U = [t \setminus u]\ {}^*\backslash^*\ U'\ @\ (T\ {}^*\backslash^*\ [u \setminus t])\ {}^*\backslash^*\ (U'\ {}^*\backslash^*\ [t \setminus u])$
using T U U' *Con Resid-rec(4)* **by** *fastforce*
also have *1*: $\ldots = [t]\ {}^*\backslash^*\ U\ @\ (T\ {}^*\backslash^*\ [u \setminus t])\ {}^*\backslash^*\ (U'\ {}^*\backslash^*\ [t \setminus u])$
using T U U' *Con Con-rec(3–4) Resid-rec(3)* **by** *auto*
also have $\ldots = [t]\ {}^*\backslash^*\ U\ @\ T\ {}^*\backslash^*\ ((u \setminus t)\ \#\ (U'\ {}^*\backslash^*\ [t \setminus u]))$
proof −
have $T\ {}^*\backslash^*\ ((u \setminus t)\ \#\ (U'\ {}^*\backslash^*\ [t \setminus u])) = (T\ {}^*\backslash^*\ [u \setminus t])\ {}^*\backslash^*\ (U'\ {}^*\backslash^*\ [t \setminus u])$
using T U U' *ind* [*of* T $U'\ {}^*\backslash^*\ [t \setminus u]$] *Con Con-rec(4) Con-sym len length-Resid*
by *fastforce*
thus *?thesis* **by** *auto*
qed
also have $\ldots = [t]\ {}^*\backslash^*\ U\ @\ T\ {}^*\backslash^*\ (U\ {}^*\backslash^*\ [t])$
using T U U' *1 Con Con-rec(4) Con-sym1 Residx1-as-Resid*
Resid1x-as-Resid Resid-rec(2) Con-sym Con-initial-left
by *auto*
finally show *?thesis* **by** *simp*
qed
qed
show $t\ \#\ T\ {}^*\!\frown^*\ U \Longrightarrow [t]\ {}^*\!\frown^*\ U$
by (*simp add*: *Con-initial-left*)
show $t\ \#\ T\ {}^*\!\frown^*\ U \Longrightarrow T\ {}^*\!\frown^*\ (U\ {}^*\backslash^*\ [t])$
by (*metis 1 Suc-inject T append-Nil2 length-0-conv length-Cons length-Resid*)
show $[t]\ {}^*\!\frown^*\ U \wedge T\ {}^*\!\frown^*\ U\ {}^*\backslash^*\ [t] \Longrightarrow t\ \#\ T\ {}^*\!\frown^*\ U$
proof (*cases* U)
show $[\![[t]\ {}^*\!\frown^*\ U \wedge T\ {}^*\!\frown^*\ U\ {}^*\backslash^*\ [t];\ U = []]\!] \Longrightarrow t\ \#\ T\ {}^*\!\frown^*\ U$
using U **by** *simp*
fix $u\ U'$
assume U: $U = u\ \#\ U'$

```
  assume Con: [t] *⌢* U ∧ T *⌢* U *\* [t]
  show t # T *⌢* U
  proof (cases U' = [])
    show U' = [] ⟹ ?thesis
      using T U Con
      by (metis Con-rec(2) Resid.simps(3) R.con-sym)
    assume U': U' ≠ []
    show ?thesis
    proof −
      have t ⌢ u
        using T U U' Con Con-rec(3) by blast
      moreover have T *⌢* [u \ t]
        using T U U' Con Con-initial-right Con-sym1 Residx1-as-Resid
              Resid1x-as-Resid Resid-rec(2)
        by (metis Con-sym)
      moreover have [t \ u] *⌢* U'
        using T U U' Con Resid-rec(3) by force
      moreover have T *\* [u \ t] *⌢* U' *\* [t \ u]
        by (metis (no-types, opaque-lifting) Con Con-sym Resid-rec(2) Suc-le-mono
            T U U' add-Suc-right calculation(3) ind len length-Cons length-Resid)
      ultimately show ?thesis
        using T U U' Con-rec(4) by simp
    qed
  qed
qed
next
fix u
show 1: T *⌢* u # U ⟹ T *\* (u # U) = (T *\* [u]) *\* U
proof (cases T)
  show 2: ⟦T *⌢* u # U; T = []⟧ ⟹ T *\* (u # U) = (T *\* [u]) *\* U
    using T by simp
  fix t T'
  assume T: T = t # T'
  assume Con: T *⌢* u # U
  show ?thesis
  proof (cases T' = [])
    show T' = [] ⟹ ?thesis
      using T U Con Con-rec(3) Resid1x-as-Resid Resid-rec(3) by force
    assume T': T' ≠ []
    have T *\* (u # U) = [t \ u] *\* U @ (T' *\* [u \ t]) *\* (U *\* [t \ u])
      using T U T' Con Resid-rec(4) [of T' U t u] by simp
    also have ... = ((t \ u) # (T' *\* [u \ t])) *\* U
    proof −
      have length (T' *\* [u \ t]) + length U ≤ n
        by (metis (no-types, lifting) Con Con-rec(4) One-nat-def Suc-eq-plus1 Suc-leI
            T T' U add-Suc le-less-trans len length-Resid lessI list.size(4)
            not-le)
      thus ?thesis
        using ind [of T' *\* [u \ t] U] Con Con-rec(4) T T' U by auto
```

```
    qed
    also have ... = (T *\* [u]) *\* U
      using T U T' Con Con-rec(2,4) Resid-rec(2) by force
    finally show ?thesis by simp
  qed
qed
show T *⌢* u # U ⟹ T *⌢* [u]
  using 1 by force
show T *⌢* u # U ⟹ T *\* [u] *⌢* U
  using 1 by fastforce
show T *⌢* [u] ∧ T *\* [u] *⌢* U ⟹ T *⌢* u # U
proof (cases T)
  show ⟦T *⌢* [u] ∧ T *\* [u] *⌢* U; T = []⟧ ⟹ T *⌢* u # U
    using T by simp
  fix t T'
  assume T: T = t # T'
  assume Con: T *⌢* [u] ∧ T *\* [u] *⌢* U
  show Con T (u # U)
  proof (cases T' = [])
    show T' = [] ⟹ ?thesis
      using Con T U Con-rec(1,3) by auto
    assume T': T' ≠ []
    have t ⌢ u
      using Con T U T' Con-rec(2) by blast
    moreover have 2: T' *⌢* [u \ t]
      using Con T U T' Con-rec(2) by blast
    moreover have [t \ u] *⌢* U
      using Con T U T'
      by (metis Con-initial-left Resid-rec(2))
    moreover have T' *\* [u \ t] *⌢* U *\* [t \ u]
    proof −
      have 0: length (U *\* [t \ u]) = length U
        using Con T U T' length-Resid Con-sym calculation(3) by blast
      hence 1: length T' + length (U *\* [t \ u]) ≤ n
        using Con T U T' len length-Resid Con-sym by simp
      have length ((T *\* [u]) *\* U) =
            length ([t \ u] *\* U) + length ((T' *\* [u \ t]) *\* (U *\* [t \ u]))
      proof −
        have (T *\* [u]) *\* U =
              [t \ u] *\* U @ (T' *\* [u \ t]) *\* (U *\* [t \ u])
          by (metis 0 1 2 Con Resid-rec(2) T T' U ind length-Resid)
        thus ?thesis
          using Con T U T' length-Resid by simp
      qed
      moreover have length ((T *\* [u]) *\* U) = length T
        using Con T U T' length-Resid by metis
      moreover have length ([t \ u] *\* U) ≤ 1
        using Con T U T' Resid1x-as-Resid
        by (metis One-nat-def length-Cons list.size(3) order-refl zero-le)
```

```
      ultimately show ?thesis
        using Con T U T' length-Resid by auto
    qed
    ultimately show T *⌢* u # U
      using T Con-rec(4) [of T' U t u] by fastforce
  qed
 qed
qed
```

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

```
lemma Con-cons:
assumes T ≠ [] and U ≠ []
shows t # T *⌢* U ⟷ [t] *⌢* U ∧ T *⌢* U *\* [t]
and T *⌢* u # U ⟷ T *⌢* [u] ∧ T *\* [u] *⌢* U
  using assms Resid-cons-ind [of T U] by blast+

lemma Con-consI [intro, simp]:
shows ⟦T ≠ []; U ≠ []; [t] *⌢* U; T *⌢* U *\* [t]⟧ ⟹ t # T *⌢* U
and ⟦T ≠ []; U ≠ []; T *⌢* [u]; T *\* [u] *⌢* U⟧ ⟹ T *⌢* u # U
  using Con-cons by auto


lemma Resid-cons:
assumes U ≠ []
shows t # T *⌢* U ⟹ (t # T) *\* U = ([t] *\* U) @ (T *\* (U *\* [t]))
and T *⌢* u # U ⟹ T *\* (u # U) = (T *\* [u]) *\* U
  using assms Resid-cons-ind [of T U] Resid.simps(1)
  by blast+
```

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

```
lemma Resid-cons':
assumes T ≠ []
shows t # T *⌢* U ⟹ (t # T) *\* U = (t ¹\* U) # (T *\* (U *\* [t]))
  using assms
  by (metis Con-sym Resid.simps(1) Resid1x-as-Resid Resid-cons(1)
      append-Cons append-Nil)

lemma Srcs-Resid-Arr-single:
assumes T *⌢* [u]
shows Srcs (T *\* [u]) = R.targets u
proof (cases T)
  show T = [] ⟹ Srcs (T *\* [u]) = R.targets u
```

```
    using assms by simp
  fix t T'
  assume T: T = t # T'
  show Srcs (T *\* [u]) = R.targets u
  proof (cases T' = [])
    show T' = [] ⟹ ?thesis
      using assms T R.sources-resid by auto
    assume T': T' ≠ []
    have Srcs (T *\* [u]) = Srcs ((t # T') *\* [u])
      using T by simp
    also have ... = Srcs ((t \ u) # (T' *\* ([u] *\* T')))
      using assms T
      by (metis Resid-rec(2) Srcs.elims T' list.distinct(1) list.sel(1))
    also have ... = R.sources (t \ u)
      using Srcs.elims by blast
    also have ... = R.targets u
      using assms Con-rec(2) T T' R.sources-resid by force
    finally show ?thesis by blast
  qed
qed

lemma Srcs-Resid-single-Arr:
shows [u] *⌢* T ⟹ Srcs ([u] *\* T) = Trgs T
proof (induct T arbitrary: u)
  show ⋀u. [u] *⌢* [] ⟹ Srcs ([u] *\* []) = Trgs []
    by simp
  fix t u T
  assume ind: ⋀u. [u] *⌢* T ⟹ Srcs ([u] *\* T) = Trgs T
  assume Con: [u] *⌢* t # T
  show Srcs ([u] *\* (t # T)) = Trgs (t # T)
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      using Con Srcs-Resid-Arr-single Trgs.simps(2) by presburger
    assume T: T ≠ []
    have Srcs ([u] *\* (t # T)) = Srcs ([u \ t] *\* T)
      using Con Resid-rec(3) T by force
    also have ... = Trgs T
      using Con ind Con-rec(3) T by auto
    also have ... = Trgs (t # T)
      by (metis T Trgs.elims Trgs.simps(3))
    finally show ?thesis by simp
  qed
qed

lemma Trgs-Resid-sym-Arr-single:
shows T *⌢* [u] ⟹ Trgs (T *\* [u]) = Trgs ([u] *\* T)
proof (induct T arbitrary: u)
  show ⋀u. [] *⌢* [u] ⟹ Trgs ([] *\* [u]) = Trgs ([u] *\* [])
    by simp
```

```
  fix t u T
  assume ind: ⋀u. T *⌢* [u] ⟹ Trgs (T *\* [u]) = Trgs ([u] *\* T)
  assume Con: t # T *⌢* [u]
  show Trgs ((t # T) *\* [u]) = Trgs ([u] *\* (t # T))
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      using R.targets-resid-sym
      by (simp add: R.con-sym)
    assume T: T ≠ []
    show ?thesis
    proof −
      have Trgs ((t # T) *\* [u]) = Trgs ((t \ u) # (T *\* [u \ t]))
        using Con Resid-rec(2) T by auto
      also have ... = Trgs (T *\* [u \ t])
        using T Con Con-rec(2) [of T t u]
        by (metis Trgs.elims Trgs.simps(3))
      also have ... = Trgs ([u \ t] *\* T)
        using T Con ind Con-sym by metis
      also have ... = Trgs ([u] *\* (t # T))
        using T Con Con-sym Resid-rec(3) by presburger
      finally show ?thesis by blast
    qed
  qed
qed

lemma Srcs-Resid [simp]:
shows T *⌢* U ⟹ Srcs (T *\* U) = Trgs U
proof (induct U arbitrary: T)
  show ⋀T. T *⌢* [] ⟹ Srcs (T *\* []) = Trgs []
    using Con-sym Resid.simps(1) by blast
  fix u U T
  assume ind: ⋀T. T *⌢* U ⟹ Srcs (T *\* U) = Trgs U
  assume Con: T *⌢* u # U
  show Srcs (T *\* (u # U)) = Trgs (u # U)
    by (metis Con Resid-cons(2) Srcs-Resid-Arr-single Trgs.simps(2−3) ind
        list.exhaust-sel)
qed

lemma Trgs-Resid-sym [simp]:
shows T *⌢* U ⟹ Trgs (T *\* U) = Trgs (U *\* T)
proof (induct U arbitrary: T)
  show ⋀T. T *⌢* [] ⟹ Trgs (T *\* []) = Trgs ([] *\* T)
    by (meson Con-sym Resid.simps(1))
  fix u U T
  assume ind: ⋀T. T *⌢* U ⟹ Trgs (T *\* U) = Trgs (U *\* T)
  assume Con: T *⌢* u # U
  show Trgs (T *\* (u # U)) = Trgs ((u # U) *\* T)
  proof (cases U = [])
    show U = [] ⟹ ?thesis
```

```
      using Con Trgs-Resid-sym-Arr-single by blast
    assume U: U ≠ []
    show ?thesis
    proof −
      have Trgs (T *\* (u # U)) = Trgs ((T *\* [u]) *\* U)
        using U by (metis Con Resid-cons(2))
      also have ... = Trgs (U *\* (T *\* [u]))
        using U Con by (metis Con-sym ind)
      also have ... = Trgs ((u # U) *\* T)
       by (metis (no-types, opaque-lifting) Con-cons(1) Con-sym Resid.simps(1) Resid-cons'
           Trgs.simps(3) U neq-Nil-conv)
      finally show ?thesis by simp
    qed
  qed
qed

lemma img-Resid-Srcs:
shows Arr T ⟹ (λa. [a] *\* T) ` Srcs T ⊆ (λb. [b]) ` Trgs T
proof (induct T)
  show Arr [] ⟹ (λa. [a] *\* []) ` Srcs [] ⊆ (λb. [b]) ` Trgs []
    by simp
  fix t :: 'a and T :: 'a list
  assume tT: Arr (t # T)
  assume ind: Arr T ⟹ (λa. [a] *\* T) ` Srcs T ⊆ (λb. [b]) ` Trgs T
  show (λa. [a] *\* (t # T)) ` Srcs (t # T) ⊆ (λb. [b]) ` Trgs (t # T)
  proof
    fix B
    assume B: B ∈ (λa. [a] *\* (t # T)) ` Srcs (t # T)
    show B ∈ (λb. [b]) ` Trgs (t # T)
    proof (cases T = [])
      assume T: T = []
      obtain a where a: a ∈ R.sources t ∧ [a \ t] = B
        by (metis (no-types, lifting) B R.composite-of-source-arr R.con-prfx-composite-of(1)
            Resid-rec(1) Srcs.simps(2) T Arr.simps(2) Con-rec(1) imageE tT)
      have a \ t ∈ Trgs (t # T)
        using tT T a
        by (simp add: R.resid-source-in-targets)
      thus ?thesis
        using B a image-iff by fastforce
      next
      assume T: T ≠ []
      obtain a where a: a ∈ R.sources t ∧ [a] *\* (t # T) = B
        using tT T B Srcs.elims by blast
      have [a \ t] *\* T = B
        using tT T B a
        by (metis Con-rec(3) R.arrI R.resid-source-in-targets R.targets-are-cong
            Resid-rec(3) R.arr-resid-iff-con R.ide-implies-arr)
      moreover have a \ t ∈ Srcs T
        using a tT
```

```
      by (metis Arr.simps(3) R.resid-source-in-targets T neq-Nil-conv subsetD)
    ultimately show ?thesis
      using T tT ind
      by (metis Trgs.simps(3) Arr.simps(3) image-iff list.exhaust-sel subsetD)
  qed
 qed
qed

lemma Resid-Arr-Src:
shows ⟦Arr T; a ∈ Srcs T⟧ ⟹ T *\* [a] = T
proof (induct T arbitrary: a)
 show ⋀a. ⟦Arr []; a ∈ Srcs []⟧ ⟹ [] *\* [a] = []
  by simp
 fix a t T
 assume ind: ⋀a. ⟦Arr T; a ∈ Srcs T⟧ ⟹ T *\* [a] = T
 assume Arr: Arr (t # T)
 assume a: a ∈ Srcs (t # T)
 show (t # T) *\* [a] = t # T
 proof (cases T = [])
  show T = [] ⟹ ?thesis
   using a R.resid-arr-ide R.sources-def by auto
  assume T: T ≠ []
  show (t # T) *\* [a] = t # T
  proof -
   have 1: R.arr t ∧ Arr T ∧ R.targets t ⊆ Srcs T
    using Arr T
    by (metis Arr.elims(2) list.sel(1) list.sel(3))
   have 2: t # T *⌢* [a]
    using T a Arr Con-rec(2)
    by (metis (no-types, lifting) img-Resid-Srcs Con-sym imageE image-subset-iff
        list.distinct(1))
   have (t # T) *\* [a] = (t \ a) # (T *\* [a \ t])
    using 2 T Resid-rec(2) by simp
   moreover have t \ a = t
    using Arr a R.sources-def
    by (metis 2 CollectD Con-rec(2) T Srcs-are-ide in-mono R.resid-arr-ide)
   moreover have T *\* [a \ t] = T
    by (metis 1 2 R.in-sourcesI R.resid-source-in-targets Srcs-are-ide T a
              Con-rec(2) in-mono ind mem-Collect-eq)
   ultimately show ?thesis by simp
  qed
 qed
qed

lemma Con-single-ide-ind:
shows R.ide a ⟹ [a] *⌢* T ⟷ Arr T ∧ a ∈ Srcs T
proof (induct T arbitrary: a)
 show ⋀a. [a] *⌢* [] ⟷ Arr [] ∧ a ∈ Srcs []
  by simp
```

```
  fix a t T
  assume ind: ⋀a. R.ide a ⟹ [a] *⌢* T ⟷ Arr T ∧ a ∈ Srcs T
  assume a: R.ide a
  show [a] *⌢* (t # T) ⟷ Arr (t # T) ∧ a ∈ Srcs (t # T)
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      using a Con-sym
      by (metis Arr.simps(2) Resid-Arr-Src Srcs.simps(2) R.arr-iff-has-source
          Con-rec(1) empty-iff R.in-sourcesI list.distinct(1))
    assume T: T ≠ []
    have 1: [a] *⌢* (t # T) ⟷ a ⌢ t ∧ [a \ t] *⌢* T
      using a T Con-cons(2) [of [a] T t] by simp
    also have 2: ... ⟷ a ⌢ t ∧ Arr T ∧ a \ t ∈ Srcs T
      using a T ind R.resid-ide-arr by blast
    also have ... ⟷ Arr (t # T) ∧ a ∈ Srcs (t # T)
      using a T Con-sym R.con-sym Resid-Arr-Src R.con-implies-arr Srcs-are-ide
      apply (cases T)
       apply simp
      by (metis Arr.simps(3) R.resid-arr-ide R.targets-resid-sym Srcs.simps(3)
          Srcs-Resid-Arr-single calculation dual-order.eq-iff list.distinct(1)
          R.in-sourcesI)
    finally show ?thesis by simp
  qed
qed

lemma Con-single-ide-iff:
assumes R.ide a
shows [a] *⌢* T ⟷ Arr T ∧ a ∈ Srcs T
  using assms Con-single-ide-ind by simp

lemma Con-single-ideI [intro]:
assumes R.ide a and Arr T and a ∈ Srcs T
shows [a] *⌢* T and T *⌢* [a]
  using assms Con-single-ide-iff Con-sym by auto

lemma Resid-single-ide:
assumes R.ide a and [a] *⌢* T
shows [a] *\* T ∈ (λb. [b]) ‘ Trgs T and [simp]: T *\* [a] = T
  using assms Con-single-ide-ind img-Resid-Srcs Resid-Arr-Src Con-sym
  by blast+

lemma Resid-Arr-Ide-ind:
shows ⟦Ide A; T *⌢* A⟧ ⟹ T *\* A = T
proof (induct A)
  show ⟦Ide []; T *⌢* []⟧ ⟹ T *\* [] = T
    by simp
  fix a A
  assume ind: ⟦Ide A; T *⌢* A⟧ ⟹ T *\* A = T
  assume Ide: Ide (a # A)
```

```
  assume Con: T *⌢* a # A
  show T *\* (a # A) = T
    by (metis (no-types, lifting) Con Con-initial-left Con-sym Ide Ide.elims(2)
        Resid-cons(2) Resid-single-ide(2) ind list.inject)
qed

lemma Resid-Ide-Arr-ind:
shows ⟦Ide A; A *⌢* T⟧ ⟹ Ide (A *\* T)
proof (induct A)
  show ⟦Ide []; [] *⌢* T⟧ ⟹ Ide ([] *\* T)
    by simp
  fix a A
  assume ind: ⟦Ide A; A *⌢* T⟧ ⟹ Ide (A *\* T)
  assume Ide: Ide (a # A)
  assume Con: a # A *⌢* T
  have T: Arr T
    using Con Ide Con-single-ide-ind Con-initial-left Ide.elims(2)
    by blast
  show Ide ((a # A) *\* T)
  proof (cases A = [])
    show A = [] ⟹ ?thesis
      by (metis Con Con-sym1 Ide Ide.simps(2) Resid1x-as-Resid Resid1x-ide
          Residx1-as-Resid Con-sym)
    assume A: A ≠ []
    show ?thesis
    proof −
      have Ide ([a] *\* T)
        by (metis Con Con-initial-left Con-sym Con-sym1 Ide Ide.simps(3)
            Resid1x-as-Resid Residx1-as-Resid Ide.simps(2) Resid1x-ide
            list.exhaust-sel)
      moreover have Trgs ([a] *\* T) ⊆ Srcs (A *\* T)
        using A T Ide Con
        by (metis (no-types, lifting) Con-sym Ide.elims(2) Ide.simps(2) Resid-Arr-Ide-ind
            Srcs-Resid Trgs-Resid-sym Con-cons(2) dual-order.eq-iff list.inject)
      moreover have Ide (A *\* (T *\* [a]))
        by (metis A Con Con-cons(1) Con-sym Ide Ide.simps(3) Resid-Arr-Ide-ind
            Resid-single-ide(2) ind list.exhaust-sel)
      moreover have Ide ((a # A) *\* T) ⟷
                    Ide ([a] *\* T) ∧ Ide (A *\* (T *\* [a])) ∧
                      Trgs ([a] *\* T) ⊆ Srcs (A *\* T)
        using calculation(1−3)
        by (metis Arr.simps(1) Con Ide Ide.simps(3) Resid1x-as-Resid Resid-cons'
            Trgs.simps(2) Con-single-ide-iff Ide.simps(2) Ide-implies-Arr Resid-Arr-Src
            list.exhaust-sel)
      ultimately show ?thesis by blast
    qed
  qed
qed
```

lemma *Resid-Ide*:
assumes *Ide A* **and** *A* $^*\frown^*$ *T*
shows *T* $^*\backslash^*$ *A* = *T* **and** *Ide* (*A* $^*\backslash^*$ *T*)
using *assms Resid-Ide-Arr-ind Resid-Arr-Ide-ind Con-sym* **by** *auto*

lemma *Con-Ide-iff*:
shows *Ide A* $\Longrightarrow$ *A* $^*\frown^*$ *T* $\longleftrightarrow$ *Arr T* $\wedge$ *Srcs T* = *Srcs A*
proof (*induct A*)
show *Ide* [] $\Longrightarrow$ [] $^*\frown^*$ *T* $\longleftrightarrow$ *Arr T* $\wedge$ *Srcs T* = *Srcs* []
by *simp*
fix *a A*
assume *ind*: *Ide A* $\Longrightarrow$ *A* $^*\frown^*$ *T* $\longleftrightarrow$ *Arr T* $\wedge$ *Srcs T* = *Srcs A*
assume *Ide*: *Ide* (*a* # *A*)
show *a* # *A* $^*\frown^*$ *T* $\longleftrightarrow$ *Arr T* $\wedge$ *Srcs T* = *Srcs* (*a* # *A*)
proof (*cases A* = [])
show *A* = [] $\Longrightarrow$ *?thesis*
using *Con-single-ide-ind Ide*
by (*metis Arr.simps*(*2*) *Con-sym Ide.simps*(*2*) *Ide-implies-Arr R.arrE*
Resid-Arr-Src Srcs.simps(*2*) *Srcs-Resid R.in-sourcesI*)
assume *A*: *A* $\neq$ []
have *a* # *A* $^*\frown^*$ *T* $\longleftrightarrow$ [*a*] $^*\frown^*$ *T* $\wedge$ *A* $^*\frown^*$ *T* $^*\backslash^*$ [*a*]
using *A Ide Con-cons*(*1*) [*of A T a*] **by** *fastforce*
also have *1*: ... $\longleftrightarrow$ *Arr T* $\wedge$ *a* $\in$ *Srcs T*
by (*metis A Arr-has-Src Con-single-ide-ind Ide Ide.elims*(*2*) *Resid-Arr-Src*
Srcs-Resid-Arr-single Con-sym Srcs-eqI ind inf.absorb-iff2 list.inject)
also have ... $\longleftrightarrow$ *Arr T* $\wedge$ *Srcs T* = *Srcs* (*a* # *A*)
by (*metis A 1 Con-sym Ide Ide.simps*(*3*) *R.ideE*
R.sources-resid Resid-Arr-Src Srcs.simps(*3*) *Srcs-Resid-Arr-single*
list.exhaust-sel R.in-sourcesI)
finally show *a* # *A* $^*\frown^*$ *T* $\longleftrightarrow$ *Arr T* $\wedge$ *Srcs T* = *Srcs* (*a* # *A*)
by *blast*
qed
qed

lemma *Con-IdeI*:
assumes *Ide A* **and** *Arr T* **and** *Srcs T* = *Srcs A*
shows *A* $^*\frown^*$ *T* **and** *T* $^*\frown^*$ *A*
using *assms Con-Ide-iff Con-sym* **by** *auto*

lemma *Con-Arr-self*:
shows *Arr T* $\Longrightarrow$ *T* $^*\frown^*$ *T*
proof (*induct T*)
show *Arr* [] $\Longrightarrow$ [] $^*\frown^*$ []
by *simp*
fix *t T*
assume *ind*: *Arr T* $\Longrightarrow$ *T* $^*\frown^*$ *T*
assume *Arr*: *Arr* (*t* # *T*)
show *t* # *T* $^*\frown^*$ *t* # *T*
proof (*cases T* = [])

```
    show T = [] ⟹ ?thesis
      using Arr R.arrE by simp
    assume T: T ≠ []
    have t ⌢ t ∧ T *⌢* [t \ t] ∧ [t \ t] *⌢* T ∧ T *\* [t \ t] *⌢* T *\* [t \ t]
    proof −
      have t ⌢ t
        using Arr Arr.elims(1) by auto
      moreover have T *⌢* [t \ t]
      proof −
        have Ide [t \ t]
          by (simp add: R.arr-def R.prfx-reflexive calculation)
        moreover have Srcs [t \ t] = Srcs T
          by (metis Arr Arr.simps(2) Arr-has-Trg R.arrE R.sources-resid Srcs.simps(2)
              Srcs-eqI T Trgs.simps(2) Arr.simps(3) inf.absorb-iff2 list.exhaust)
        ultimately show ?thesis
          by (metis Arr Con-sym T Arr.simps(3) Con-Ide-iff neq-Nil-conv)
      qed
      ultimately show ?thesis
        by (metis Con-single-ide-ind Con-sym R.prfx-reflexive
            Resid-single-ide(2) ind R.con-implies-arr(1))
    qed
    thus ?thesis
      using Con-rec(4) [of T T t t] by force
  qed
qed

lemma Resid-Arr-self:
shows Arr T ⟹ Ide (T *\* T)
proof (induct T)
  show Arr [] ⟹ Ide ([] *\* [])
    by simp
  fix t T
  assume ind: Arr T ⟹ Ide (T *\* T)
  assume Arr: Arr (t # T)
  show Ide ((t # T) *\* (t # T))
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      using Arr R.prfx-reflexive by auto
    assume T: T ≠ []
    have 1: (t # T) *\* (t # T) = t ¹\* (t # T) # T *\* ((t # T) *\* [t])
      using Arr T Resid-cons′ [of T t t # T] Con-Arr-self by presburger
    also have ... = (t \ t) ¹\* T # T *\* (t ¹\* [t] # T *\* ([t] *\* [t]))
      using Arr T Resid-cons′ [of T t [t]]
      by (metis Con-initial-right Resid1x.simps(3) calculation neq-Nil-conv)
    also have ... = (t \ t) ¹\* T # (T *\* ([t] *\* [t])) *\* (T *\* ([t] *\* [t]))
      by (metis 1 Resid1x.simps(2) Residx1.simps(2) Residx1-as-Resid T calculation
          Con-cons(1) Con-rec(4) Resid-cons(2) list.distinct(1) list.inject)
    finally have 2: (t # T) *\* (t # T) =
                    (t \ t) ¹\* T # (T *\* ([t] *\* [t])) *\* (T *\* ([t] *\* [t]))
```

```
      by blast
    moreover have Ide ...
    proof −
      have R.ide ((t \ t) ¹\* T)
        using Arr T
        by (metis Con-initial-right Con-rec(2) Con-sym1 R.con-implies-arr(1)
            Resid1x-ide Con-Arr-self Residx1-as-Resid R.prfx-reflexive)
      moreover have Ide ((T *\* ([t] *\* [t])) *\* (T *\* ([t] *\* [t])))
        using Arr T
        by (metis Con-Arr-self Con-rec(4) Resid-single-ide(2) Con-single-ide-ind
            Resid.simps(3) ind R.prfx-reflexive R.con-implies-arr(2))
      moreover have R.targets ((t \ t) ¹\* T) ⊆
                      Srcs ((T *\* ([t] *\* [t])) *\* (T *\* ([t] *\* [t])))
        by (metis (no-types, lifting) 1 2 Con-cons(1) Resid1x-as-Resid T Trgs.simps(2)
            Trgs-Resid-sym Srcs-Resid dual-order.eq-iff list.discI list.inject)
      ultimately show ?thesis
        using Arr T
        by (metis Ide.simps(1,3) list.exhaust-sel)
    qed
    ultimately show ?thesis by auto
  qed
qed

lemma Con-imp-eq-Srcs:
assumes T *⌢* U
shows Srcs T = Srcs U
proof (cases T)
  show T = [] ⟹ ?thesis
    using assms by simp
  fix t T'
  assume T: T = t # T'
  show Srcs T = Srcs U
  proof (cases U)
    show U = [] ⟹ ?thesis
      using assms T by simp
    fix u U'
    assume U: U = u # U'
    show Srcs T = Srcs U
      by (metis Con-initial-right Con-rec(1) Con-sym R.con-imp-common-source
          Srcs.simps(2−3) Srcs-eqI T Trgs.cases U assms)
  qed
qed

lemma Arr-iff-Con-self:
shows Arr T ⟷ T *⌢* T
proof (induct T)
  show Arr [] ⟷ [] *⌢* []
    by simp
  fix t T
```

```
  assume ind: Arr T ⟷ T *⌢* T
  show Arr (t # T) ⟷ t # T *⌢* t # T
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      by auto
    assume T: T ≠ []
    show ?thesis
    proof
      show Arr (t # T) ⟹ t # T *⌢* t # T
        using Con-Arr-self by simp
      show t # T *⌢* t # T ⟹ Arr (t # T)
      proof −
        assume Con: t # T *⌢* t # T
        have R.arr t
          using T Con Con-rec(4) [of T T t t] by blast
        moreover have Arr T
          using T Con Con-rec(4) [of T T t t] ind R.arrI
          by (meson R.prfx-reflexive Con-single-ide-ind)
        moreover have R.targets t ⊆ Srcs T
          using T Con
          by (metis Con-cons(2) Con-imp-eq-Srcs Trgs.simps(2)
              Srcs-Resid list.distinct(1) subsetI)
        ultimately show ?thesis
          by (cases T) auto
      qed
    qed
  qed
qed

lemma Arr-Resid-single:
shows T *⌢* [u] ⟹ Arr (T *\* [u])
proof (induct T arbitrary: u)
  show ⋀u. [] *⌢* [u] ⟹ Arr ([] *\* [u])
    by simp
  fix t u T
  assume ind: ⋀u. T *⌢* [u] ⟹ Arr (T *\* [u])
  assume Con: t # T *⌢* [u]
  show Arr ((t # T) *\* [u])
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      using Con Arr-iff-Con-self R.con-imp-arr-resid Con-rec(1) by fastforce
    assume T: T ≠ []
    have Arr ((t # T) *\* [u]) ⟷ Arr ((t \ u) # (T *\* [u \ t]))
      using Con T Resid-rec(2) by auto
    also have ... ⟷ R.arr (t \ u) ∧ Arr (T *\* [u \ t]) ∧
                    R.targets (t \ u) ⊆ Srcs (T *\* [u \ t])
      using Con T
      by (metis Arr.simps(3) Con-rec(2) neq-Nil-conv)
    also have ... ⟷ R.con t u ∧ Arr (T *\* [u \ t])
```

```
    using Con T
    by (metis Srcs-Resid-Arr-single Con-rec(2) R.arr-resid-iff-con subsetI
        R.targets-resid-sym)
  also have ... ⟷ True
    using Con ind T Con-rec(2) by blast
  finally show ?thesis by auto
 qed
qed

lemma Con-imp-Arr-Resid:
shows T *⌢* U ⟹ Arr (T *\* U)
proof (induct U arbitrary: T)
 show ⋀T. T *⌢* [] ⟹ Arr (T *\* [])
  by (meson Con-sym Resid.simps(1))
 fix u U T
 assume ind: ⋀T. T *⌢* U ⟹ Arr (T *\* U)
 assume Con: T *⌢* u # U
 show Arr (T *\* (u # U))
  by (metis Arr-Resid-single Con Resid-cons(2) ind)
qed

lemma Cube-ind:
shows ⟦T *⌢* U; V *⌢* T; length T + length U + length V ≤ n⟧ ⟹
      (V *\* T *⌢* U *\* T ⟷ V *\* U *⌢* T *\* U) ∧
      (V *\* T *⌢* U *\* T ⟶
        (V *\* T) *\* (U *\* T) = (V *\* U) *\* (T *\* U))
proof (induct n arbitrary: T U V)
 show ⋀T U V. ⟦T *⌢* U; V *⌢* T; length T + length U + length V ≤ 0⟧ ⟹
              (V *\* T *⌢* U *\* T ⟷ V *\* U *⌢* T *\* U) ∧
              (V *\* T *⌢* U *\* T ⟶
                (V *\* T) *\* (U *\* T) = (V *\* U) *\* (T *\* U))
  by simp
 fix n and T U V :: 'a list
 assume Con-TU: T *⌢* U and Con-VT: V *⌢* T
 have T: T ≠ []
  using Con-TU by auto
 have U: U ≠ []
  using Con-TU Con-sym Resid.simps(1) by blast
 have V: V ≠ []
  using Con-VT by auto
 assume len: length T + length U + length V ≤ Suc n
 assume ind: ⋀T U V. ⟦T *⌢* U; V *⌢* T; length T + length U + length V ≤ n⟧ ⟹
                      (V *\* T *⌢* U *\* T ⟷ V *\* U *⌢* T *\* U) ∧
                      (V *\* T *⌢* U *\* T ⟶
                        (V *\* T) *\* (U *\* T) = (V *\* U) *\* (T *\* U))
 show (V *\* T *⌢* U *\* T ⟷ V *\* U *⌢* T *\* U) ∧
      (V *\* T *⌢* U *\* T ⟶ (V *\* T) *\* (U *\* T) = (V *\* U) *\* (T *\* U))
 proof (cases V)
  show V = [] ⟹ ?thesis
```

```
  using V by simp

fix v V'
assume V: V = v # V'
show ?thesis
proof (cases U)
  show U = [] ⟹ ?thesis
    using U by simp
  fix u U'
  assume U: U = u # U'
  show ?thesis
  proof (cases T)
    show T = [] ⟹ ?thesis
      using T by simp
    fix t T'
    assume T: T = t # T'
    show ?thesis
    proof (cases V' = [], cases U' = [], cases T' = [])
      show ⟦V' = []; U' = []; T' = []⟧ ⟹ ?thesis
        using T U V R.cube Con-TU Resid.simps(2−3) R.arr-resid-iff-con
              R.con-implies-arr Con-sym
        by metis
      assume T': T' ≠ [] and V': V' = [] and U': U' = []
      have 1: U *⌢* [t]
        using T Con-TU Con-cons(2) Con-sym Resid.simps(2) by metis
      have 2: V *⌢* [t]
        using V Con-VT Con-initial-right T by blast
      show ?thesis
      proof (intro conjI impI)
        have 3: length [t] + length U + length V ≤ n
          using T T' le-Suc-eq len by fastforce
        show ∗: V *\* T *⌢* U *\* T ⟷ V *\* U *⌢* T *\* U
        proof −
          have V *\* T *⌢* U *\* T ⟷ (V *\* [t]) *\* T' *⌢* (U *\* [t]) *\* T'
            using Con-TU Con-VT Con-sym Resid-cons(2) T T' by force
          also have ... ⟷ V *\* [t] *⌢* U *\* [t] ∧
                          (V *\* [t]) *\* (U *\* [t]) *⌢* T' *\* (U *\* [t])
          proof (intro iffI conjI)
            show (V *\* [t]) *\* T' *⌢* (U *\* [t]) *\* T' ⟹ V *\* [t] *⌢* U *\* [t]
              using T U V T' U' V' 1 ind [of T'] len Con-TU Con-rec(2) Resid-rec(1)
                    Resid.simps(1) length-Cons Suc-le-mono add-Suc
              by (metis (no-types))
            show (V *\* [t]) *\* T' *⌢* (U *\* [t]) *\* T' ⟹
                 (V *\* [t]) *\* (U *\* [t]) *⌢* T' *\* (U *\* [t])
              using T U V T' U' V'
              by (metis Con-sym Resid.simps(1) Resid-rec(1) Suc-le-mono ind len
                  length-Cons list.size(3−4))
            show V *\* [t] *⌢* U *\* [t] ∧
                 (V *\* [t]) *\* (U *\* [t]) *⌢* T' *\* (U *\* [t]) ⟹
```

```
        (V *\* [t]) *\* T' *⌢* (U *\* [t]) *\* T'
      using T U V T' U' V' 1 ind len Con-TU Con-VT Con-rec(1−3)
      by (metis (no-types, lifting) One-nat-def Resid-rec(1) Suc-le-mono
        add.commute list.size(3) list.size(4) plus-1-eq-Suc)
    qed
    also have ... ⟷ (V *\* U) *\* ([t] *\* U) *⌢* T' *\* (U *\* [t])
      by (metis 2 3 Con-sym ind Resid.simps(1))
    also have ... ⟷ V *\* U *⌢* T *\* U
      using Con-rec(2) [of T' t]
      by (metis (no-types, lifting) 1 Con-TU Con-cons(2) Resid.simps(1)
        Resid.simps(3) Resid-rec(2) T T' U U')
    finally show ?thesis by simp
  qed
  assume Con: V *\* T *⌢* U *\* T
  show (V *\* T) *\* (U *\* T) = (V *\* U) *\* (T *\* U)
  proof −
    have (V *\* T) *\* (U *\* T) = ((V *\* [t]) *\* T') *\* ((U *\* [t]) *\* T')
      using Con-TU Con-VT Con-sym Resid-cons(2) T T' by force
    also have ... = ((V *\* [t]) *\* (U *\* [t])) *\* (T' *\* (U *\* [t]))
      using T U V T' U' V' 1 Con ind [of T' Resid U [t] Resid V [t]]
      by (metis One-nat-def add.commute calculation len length-0-conv length-Resid
        list.size(4) nat-add-left-cancel-le Con-sym plus-1-eq-Suc)
    also have ... = ((V *\* U) *\* ([t] *\* U)) *\* (T' *\* (U *\* [t]))
      by (metis 1 2 3 Con-sym ind)
    also have ... = (V *\* U) *\* (T *\* U)
      using T U T' U' Con ∗
      by (metis Con-sym Resid-rec(1−2) Resid.simps(1) Resid-cons(2))
    finally show ?thesis by simp
  qed
qed
next
assume U': U' ≠ [] and V': V' = []
show ?thesis
proof (intro conjI impI)
  show ∗: V *\* T *⌢* U *\* T ⟷ V *\* U *⌢* T *\* U
  proof (cases T' = [])
    assume T': T' = []
    show ?thesis
    proof −
      have V *\* T *⌢* U *\* T ⟷ V *\* [t] *⌢* (u \ t) # (U' *\* [t \ u])
        using Con-TU Con-sym Resid-rec(2) T T' U U' by auto
      also have ... ⟷ (V *\* [t]) *\* [u \ t] *⌢* U' *\* [t \ u]
       by (metis Con-TU Con-cons(2) Con-rec(3) Con-sym Resid.simps(1) T U U')
      also have ... ⟷ (V *\* [u]) *\* [t \ u] *⌢* U' *\* [t \ u]
        using T U V V' R.cube-ax
        apply simp
        by (metis R.con-implies-arr(1) R.not-arr-null R.con-def)
      also have ... ⟷ (V *\* [u]) *\* U' *⌢* [t \ u] *\* U'
      proof −
```

```
    have length [t \ u] + length U' + length (V *\* [u]) ≤ n
      using T U V V' len by force
    thus ?thesis
      by (metis Con-sym Resid.simps(1) add.commute ind)
  qed
  also have ... ⟷ V *\* U *⌢* T *\* U
    by (metis Con-TU Resid-cons(2) Resid-rec(3) T T' U U' Con-cons(2)
      length-Resid length-0-conv)
  finally show ?thesis by simp
 qed
 next
 assume T': T' ≠ []
 show ?thesis
 proof −
  have V *\* T *⌢* U *\* T ⟷ (V *\* [t]) *\* T' *⌢* ((U *\* [t]) *\* T')
    using Con-TU Con-VT Con-sym Resid-cons(2) T T' by force
  also have ... ⟷ (V *\* [t]) *\* (U *\* [t]) *⌢* T' *\* (U *\* [t])
  proof −
    have length T' + length (U *\* [t]) + length (V *\* [t]) ≤ n
      by (metis (no-types, lifting) Con-TU Con-VT Con-initial-right Con-sym
        One-nat-def Suc-eq-plus1 T ab-semigroup-add-class.add-ac(1)
        add-le-imp-le-left len length-Resid list.size(4) plus-1-eq-Suc)
    thus ?thesis
   by (metis Con-TU Con-VT Con-cons(1) Con-cons(2) T T' U V ind list.discI)
  qed
  also have ... ⟷ (V *\* U) *\* ([t] *\* U) *⌢* T' *\* (U *\* [t])
  proof −
    have length [t] + length U + length V ≤ n
      using T T' le-Suc-eq len by fastforce
    thus ?thesis
      by (metis Con-TU Con-VT Con-initial-left Con-initial-right T ind)
  qed
  also have ... ⟷ V *\* U *⌢* T *\* U
    by (metis Con-cons(2) Con-sym Resid.simps(1) Resid1x-as-Resid
      Residx1-as-Resid Resid-cons' T T')
  finally show ?thesis by blast
 qed
qed
show V *\* T *⌢* U *\* T ⟹
      (V *\* T) *\* (U *\* T) = (V *\* U) *\* (T *\* U)
proof −
 assume Con: V *\* T *⌢* U *\* T
 show ?thesis
 proof (cases T' = [])
  assume T': T' = []
  show ?thesis
  proof −
    have 1: (V *\* T) *\* (U *\* T) =
          (V *\* T) *\* ((u \ t) # (U'*\* [t \ u]))
```

```
    using Con-TU Con-sym Resid-rec(2) T T' U U' by force
  also have ... = ((V *\* [t]) *\* [u \ t]) *\* (U' *\* [t \ u])
    by (metis Con Con-TU Con-rec(2) Con-sym Resid-cons(2) T T' U U'
      calculation)
  also have ... = ((V *\* [u]) *\* [t \ u]) *\* (U' *\* [t \ u])
    by (metis * Con Con-rec(3) R.cube Resid.simps(1,3) T T' U V V'
      calculation R.conI R.conE)
  also have ... = ((V *\* [u]) *\* U') *\* ([t \ u] *\* U')
  proof -
    have length [t \ u] + length (U' *\* [t \ u]) + length (V *\* [u]) ≤ n
      by (metis (no-types, lifting) Nat.le-diff-conv2 One-nat-def T U V V'
        add.commute add-diff-cancel-left' add-leD2 len length-Cons
        length-Resid list.size(3) plus-1-eq-Suc)
    thus ?thesis
      by (metis Con-sym add.commute Resid.simps(1) ind length-Resid)
  qed
  also have ... = (V *\* U) *\* (T *\* U)
    by (metis Con-TU Con-cons(2) Resid-cons(2) T T' U U'
      Resid-rec(3) length-0-conv length-Resid)
  finally show ?thesis by blast
qed
next
assume T': T' ≠ []
show ?thesis
proof -
  have (V *\* T) *\* (U *\* T) =
      ((V *\* T) *\* ([u] *\* T)) *\* (U' *\* (T *\* [u]))
    by (metis Con Con-TU Resid.simps(2) Resid1x-as-Resid U U'
      Con-cons(2) Con-sym Resid-cons' Resid-cons(2))
  also have ... = ((V *\* [u]) *\* (T *\* [u])) *\* (U' *\* (T *\* [u]))
  proof -
    have length T + length [u] + length V ≤ n
      using U U' antisym-conv len not-less-eq-eq by fastforce
    thus ?thesis
      by (metis Con-TU Con-VT Con-initial-right U ind)
  qed
  also have ... = ((V *\* [u]) *\* U') *\* ((T *\* [u]) *\* U')
  proof -
    have length (T *\* [u]) + length U' + length (V *\* [u]) ≤ n
      using Con-TU Con-initial-right U V V' len length-Resid by force
    thus ?thesis
      by (metis Con Con-TU Con-cons(2) U U' calculation ind length-0-conv
        length-Resid)
  qed
  also have ... = (V *\* U) *\* (T *\* U)
    by (metis * Con Con-TU Resid-cons(2) U U' length-Resid length-0-conv)
  finally show ?thesis by blast
qed
qed
```

```
  qed
qed
next
assume V': V' ≠ []
show ?thesis
proof (cases U' = [])
  assume U': U' = []
  show ?thesis
  proof (cases T' = [])
    assume T': T' = []
    show ?thesis
    proof (intro conjI impI)
      show ∗: V *\* T *⌢* U *\* T ⟷ V *\* U *⌢* T *\* U
      proof −
        have V *\* T *⌢* U *\* T ⟷ (v \ t) # (V' *\* [t \ v]) *⌢* [u \ t]
          using Con-TU Con-VT Con-sym Resid-rec(1−2) T T' U U' V V'
          by metis
        also have ... ⟷ [v \ t] *⌢* [u \ t] ∧
                          V' *\* [t \ v] *⌢* [u \ v] *\* [t \ v]
        proof −
          have V' *⌢* [t \ v]
            using T T' V V' Con-VT Con-rec(2) by blast
          thus ?thesis
            using R.con-def R.con-sym R.cube
                  Con-rec(2) [of V' *\* [t \ v] v \ t u \ t]
            by auto
        qed
        also have ... ⟷ [v \ t] *⌢* [u \ t] ∧
                          V' *\* [u \ v] *⌢* [t \ v] *\* [u \ v]
        proof −
          have length [t \ v] + length [u \ v] + length V' ≤ n
            using T U V len by fastforce
          thus ?thesis
            by (metis Con-imp-Arr-Resid Arr-has-Src Con-VT T T' Trgs.simps(1)
                Trgs-Resid-sym V V' Con-rec(2) Srcs-Resid ind)
        qed
        also have ... ⟷ [v \ t] *⌢* [u \ t] ∧
                          V' *\* [u \ v] *⌢* [t \ u] *\* [v \ u]
          by (simp add: R.con-def R.cube)
        also have ... ⟷ V *\* U *⌢* T *\* U
        proof
          assume 1: V *\* U *⌢* T *\* U
          have tu-vu: t \ u ⌢ v \ u
            by (metis (no-types, lifting) 1 T T' U U' V V' Con-rec(3)
                Resid-rec(1−2) Con-sym length-Resid length-0-conv)
          have vt-ut: v \ t ⌢ u \ t
            using 1
            by (metis R.con-def R.con-sym R.cube tu-vu)
          show [v \ t] *⌢* [u \ t] ∧ V' *\* [u \ v] *⌢* [t \ u] *\* [v \ u]
```

```
    by (metis (no-types, lifting) 1 Con-TU Con-cons(1) Con-rec(1−2)
        Resid-rec(1) T T′ U U′ V V′ Resid-rec(2) length-Resid
        length-0-conv vt-ut)
  next
  assume 1: [v \ t] *⌢* [u \ t] ∧
            V′ *\* [u \ v] *⌢* [t \ u] *\* [v \ u]
  have tu-vu: t \ u ⌢ v \ u ∧ v \ t ⌢ u \ t
    by (metis 1 Con-sym Resid.simps(1) Residx1.simps(2)
        Residx1-as-Resid)
  have tu: t ⌢ u
    using Con-TU Con-rec(1) T T′ U U′ by blast
  show V *\* U *⌢* T *\* U
    by (metis (no-types, opaque-lifting) 1 Con-rec(2) Con-sym
        R.con-implies-arr(2) Resid.simps(1,3) T T′ U U′ V V′
        Resid-rec(2) R.arr-resid-iff-con)
qed
finally show ?thesis by simp
qed
show V *\* T *⌢* U *\* T ⟹
      (V *\* T) *\* (U *\* T) = (V *\* U) *\* (T *\* U)
proof −
  assume Con: V *\* T *⌢* U *\* T
  have (V *\* T) *\* (U *\* T) = ((v \ t) # (V′ *\* [t \ v])) *\* [u \ t]
   using Con-TU Con-VT Con-sym Resid-rec(1−2) T T′ U U′ V V′ by metis
  also have 1: ... = ((v \ t) \ (u \ t)) #
                     (V′ *\* [t \ v]) *\* ([u \ v] *\* [t \ v])
    apply simp
    by (metis Con Con-VT Con-rec(2) R.conE R.conI R.con-sym R.cube
        Resid-rec(2) T T′ V V′ calculation(1))
  also have ... = ((v \ t) \ (u \ t)) #
                  (V′ *\* [u \ v]) *\* ([t \ v] *\* [u \ v])
  proof −
    have length [t \ v] + length [u \ v] + length V′ ≤ n
      using T U V len by fastforce
    moreover have u \ v ⌢ t \ v
      by (metis 1 Con-VT Con-rec(2) R.con-sym-ax T T′ V V′ list.discI
          R.conE R.conI R.cube)
    moreover have t \ v ⌢ u \ v
      using R.con-sym calculation(2) by blast
    ultimately show ?thesis
      by (metis Con-VT Con-rec(2) T T′ V V′ Con-rec(1) ind)
  qed
  also have ... = ((v \ t) \ (u \ t)) #
                  ((V′ *\* [u \ v]) *\* ([t \ u] *\* [v \ u]))
    using R.cube by fastforce
  also have ... = ((v \ u) \ (t \ u)) #
                  ((V′ *\* [u \ v]) *\* ([t \ u] *\* [v \ u]))
    by (metis R.cube)
  also have ... = (V *\* U) *\* (T *\* U)
```

```
    proof −
     have (V *\* U) *\* (T *\* U) = ((v \ u) # ((V' *\* [u \ v]))) *\* [t \ u]
       using T T' U U' V Resid-cons(1) [of [u] v V']
       by (metis ∗ Con Con-TU Resid.simps(1) Resid-rec(1) Resid-rec(2))
     also have ... = ((v \ u) \ (t \ u)) #
                     ((V' *\* [u \ v]) *\* ([t \ u] *\* [v \ u]))
       by (metis ∗ Con Con-initial-left calculation Con-sym Resid.simps(1)
           Resid-rec(1−2))
     finally show ?thesis by simp
    qed
    finally show ?thesis by simp
  qed
qed
next
assume T': T' ≠ []
show ?thesis
proof (intro conjI impI)
  show ∗: V *\* T *⌢* U *\* T ⟷ V *\* U *⌢* T *\* U
  proof −
    have V *\* T *⌢* U *\* T ⟷ (V *\* [t]) *\* T' *⌢* [u \ t] *\* T'
      using Con-TU Con-VT Con-sym Resid-cons(2) Resid-rec(3) T T' U U'
      by force
    also have ... ⟷ (V *\* [t]) *\* [u \ t] *⌢* T' *\* [u \ t]
    proof −
      have length [u \ t] + length T' + length (V *\* [t]) ≤ n
        using Con-VT Con-initial-right T U length-Resid len by fastforce
      thus ?thesis
     by (metis Con-TU Con-VT Con-rec(2) T T' U V add.commute Con-cons(2)
         ind list.discI)
    qed
    also have ... ⟷ (V *\* [u]) *\* [t \ u] *⌢* T' *\* [u \ t]
    proof −
      have length [t] + length [u] + length V ≤ n
        using T T' U le-Suc-eq len by fastforce
      hence (V *\* [t]) *\* ([u] *\* [t]) = (V *\* [u]) *\* ([t] *\* [u])
        using ind [of [t] [u] V]
        by (metis Con-TU Con-VT Con-initial-left Con-initial-right T U)
      thus ?thesis
       by (metis (full-types) Con-TU Con-initial-left Con-sym Resid-rec(1) T U)
    qed
    also have ... ⟷ V *\* U *⌢* T *\* U
      by (metis Con-TU Con-cons(2) Con-rec(2) Resid.simps(1) Resid-rec(2)
          T T' U U')
    finally show ?thesis by simp
  qed
  show V *\* T *⌢* U *\* T ⟹
       (V *\* T) *\* (U *\* T) = (V *\* U) *\* (T *\* U)
  proof −
    assume Con: V *\* T *⌢* U *\* T
```

```
      have (V *\* T) *\* (U *\* T) = ((V *\* [t]) *\* T') *\* ([u \ t] *\* T')
        using Con-TU Con-VT Con-sym Resid-cons(2) Resid-rec(3) T T' U U'
        by force
      also have ... = ((V *\* [t]) *\* [u \ t]) *\* (T' *\* [u \ t])
      proof −
        have length [u \ t] + length T' + length (Resid V [t]) ≤ n
          using Con-VT Con-initial-right T U length-Resid len by fastforce
        thus ?thesis
       by (metis Con-TU Con-VT Con-cons(2) Con-rec(2) T T' U V add.commute
             ind list.discI)
      qed
      also have ... = ((V *\* [u]) *\* [t \ u]) *\* (T' *\* [u \ t])
      proof −
        have length [t] + length [u] + length V ≤ n
          using T T' U le-Suc-eq len by fastforce
        thus ?thesis
          using ind [of [t] [u] V]
          by (metis Con-TU Con-VT Con-initial-left Con-sym Resid-rec(1) T U)
      qed
      also have ... = (V *\* U) *\* (T *\* U)
        using ∗ Con Con-TU Con-rec(2) Resid-cons(2) Resid-rec(2) T T' U U'
        by auto
      finally show ?thesis by simp
    qed
  qed
qed
next
assume U': U' ≠ []
show ?thesis
proof (cases T' = [])
  assume T': T' = []
  show ?thesis
  proof (intro conjI impI)
    show ∗: V *\* T *⌢* U *\* T ⟷ V *\* U *⌢* T *\* U
    proof −
      have V *\* T *⌢* U *\* T ⟷ V *\* [t] *⌢* (u \ t) # (U' *\* [t \ u])
        using T U V T' U' V' Con-TU Con-VT Con-sym Resid-rec(2) by auto
      also have ... ⟷ V *\* [t] *⌢* [u \ t] ∧
                      (V *\* [t]) *\* [u \ t] *⌢* U' *\* [t \ u]
        by (metis Con-TU Con-VT Con-cons(2) Con-initial-right
            Con-rec(2) Con-sym T U U')
      also have ... ⟷ V *\* [t] *⌢* [u \ t] ∧
                      (V *\* [u]) *\* [t \ u] *⌢* U' *\* [t \ u]
      proof −
        have length [u] + length [t] + length V ≤ n
          using T U V T' U' V' len not-less-eq-eq order-trans by fastforce
        thus ?thesis
          using ind [of [t] [u] V]
          by (metis Con-TU Con-VT Con-initial-right Resid-rec(1) T U
```

```
        Con-sym length-Cons)
  qed
  also have ... ⟷ V *\* [u] *⌢* [t \ u] ∧
                  (V *\* [u]) *\* [t \ u] *⌢* U' *\* [t \ u]
  proof −
    have length [t] + length [u] + length V ≤ n
      using T U V T' U' V' len antisym-conv not-less-eq-eq by fastforce
    thus ?thesis
      using ind [of [t]]
      by (metis (full-types) Con-TU Con-VT Con-initial-right Con-sym
          Resid-rec(1) T U)
  qed
  also have ... ⟷ (V *\* [u]) *\* U' *⌢* [t \ u] *\* U'
  proof −
    have length [t \ u] + length U' + length (V *\* [u]) ≤ n
      by (metis T T' U add.assoc add.right-neutral add-leD1
          add-le-cancel-left length-Resid len length-Cons list.size(3)
          plus-1-eq-Suc)
    thus ?thesis
      by (metis (no-types, opaque-lifting) Con-sym Resid.simps(1)
          add.commute ind)
  qed
  also have ... ⟷ V *\* U *⌢* T *\* U
    by (metis Con-TU Resid-cons(2) Resid-rec(3) T T' U U'
        Con-cons(2) length-Resid length-0-conv)
  finally show ?thesis by blast
qed
show V *\* T *⌢* U *\* T ⟹
      (V *\* T) *\* (U *\* T) = (V *\* U) *\* (T *\* U)
proof −
  assume Con: V *\* T *⌢* U *\* T
  have (V *\* T) *\* (U *\* T) =
       (V *\* [t]) *\* ((u \ t) # (U' *\* [t \ u]))
    using Con-TU Con-sym Resid-rec(2) T T' U U' by auto
 also have ... = ((V *\* [t]) *\* [u \ t]) *\* (U' *\* [t \ u])
    by (metis Con Con-TU Con-rec(2) Con-sym T T' U U' calculation
        Resid-cons(2))
  also have ... = ((V *\* [u]) *\* [t \ u]) *\* (U' *\* [t \ u])
  proof −
    have length [t] + length [u] + length V ≤ n
      using T U U' le-Suc-eq len by fastforce
    thus ?thesis
      using T U Con-TU Con-VT Con-sym ind [of [t] [u] V]
      by (metis (no-types, opaque-lifting) Con-initial-right Resid.simps(3))
  qed
  also have ... = ((V *\* [u]) *\* U') *\* ([t \ u] *\* U')
  proof −
    have length [t \ u] + length U' + length (V *\* [u]) ≤ n
      by (metis (no-types, opaque-lifting) T T' U add.left-commute
```

add.right-neutral add-leD2 add-le-cancel-left len length-Cons length-Resid list.size(3) plus-1-eq-Suc)
    thus *?thesis*
     by (*metis Con Con-TU Con-rec(3) T T′ U U′ calculation ind length-0-conv length-Resid*)
   qed
   also have ... = (*V* ** *U*) ** (*T* ** *U*)
    by (*metis* ∗ *Con Con-TU Resid-rec(3) T T′ U U′ Resid-cons(2) length-Resid length-0-conv*)
   finally show *?thesis* **by** *blast*
  qed
qed
next
assume *T′*: *T′* ≠ []
show *?thesis*
proof (*intro conjI impI*)
  have *1*: *U* *⌢* [*t*]
   using *T Con-TU*
   by (*metis Con-cons(2) Con-sym Resid.simps(2)*)
  have *2*: *V* *⌢* [*t*]
   using *V Con-VT Con-initial-right T* **by** *blast*
  have *3*: *length T′* + *length* (*U* ** [*t*]) + *length* (*V* ** [*t*]) ≤ *n*
   using *1 2 T len length-Resid* **by** *force*
  have *4*: *length* [*t*] + *length U* + *length V* ≤ *n*
   using *T T′ len antisym-conv not-less-eq-eq* **by** *fastforce*
  show ∗: *V* ** *T* *⌢* *U* ** *T* ⟷ *V* ** *U* *⌢* *T* ** *U*
  proof −
   have *V* ** *T* *⌢* *U* ** *T* ⟷ (*V* ** [*t*]) ** *T′* *⌢* (*U* ** [*t*]) ** *T′*
    using *Con-TU Con-VT Con-sym Resid-cons(2) T T′* **by** *force*
   also have ... ⟷ (*V* ** [*t*]) ** (*U* ** [*t*]) *⌢* *T′* ** (*U* ** [*t*])
    by (*metis 3 Con-TU Con-VT Con-cons(1) Con-cons(2) T T′ U V ind list.discI*)
   also have ... ⟷ (*V* ** *U*) ** ([*t*] ** *U*) *⌢* *T′* ** (*U* ** [*t*])
    by (*metis 1 2 4 Con-sym ind*)
   also have ... ⟷ *V* ** *U* *⌢* *hd* ([*t*] ** *U*) # *T′* ** (*U* ** [*t*])
    by (*metis 1 Con-TU Con-cons(1) Con-cons(2) Resid.simps(1) Resid1x-as-Resid T T′ list.sel(1)*)
   also have ... ⟷ *V* ** *U* *⌢* *T* ** *U*
    using *1 Resid-cons′* [*of T′ t U*] *Con-TU T T′ Resid1x-as-Resid Con-sym*
    by *force*
   finally show *?thesis* **by** *simp*
  qed
  show (*V* ** *T*) ** (*U* ** *T*) = (*V* ** *U*) ** (*T* ** *U*)
  proof −
   have (*V* ** *T*) ** (*U* ** *T*) =
       ((*V* ** [*t*]) ** *T′*) ** ((*U* ** [*t*]) ** *T′*)
    using *Con-TU Con-VT Con-sym Resid-cons(2) T T′* **by** *force*
   also have ... = ((*V* ** [*t*]) ** (*U* ** [*t*])) ** (*T′* ** (*U* ** [*t*]))

```
      by (metis (no-types, lifting) 3 Con-TU Con-VT T T' U V Con-cons(1)
          Con-cons(2) ind list.simps(3))
    also have ... = ((V *\* U) *\* ([t] *\* U)) *\* (T' *\* (U *\* [t]))
      by (metis 1 2 4 Con-sym ind)
    also have ... = (V *\* U) *\* ((t # T') *\* U)
      by (metis * Con-TU Con-cons(1) Resid1x-as-Resid
          Resid-cons' T T' U calculation Resid-cons(2) list.distinct(1))
    also have ... = (V *\* U) *\* (T *\* U)
      using T by fastforce
    finally show ?thesis by simp
  qed
 qed
qed

lemma Cube:
shows T *\* U *⌢* V *\* U ⟷ T *\* V *⌢* U *\* V
and T *\* U *⌢* V *\* U ⟹ (T *\* U) *\* (V *\* U) = (T *\* V) *\* (U *\* V)
proof −
  show T *\* U *⌢* V *\* U ⟷ T *\* V *⌢* U *\* V
    using Cube-ind by (metis Con-sym Resid.simps(1) le-add2)
  show T *\* U *⌢* V *\* U ⟹ (T *\* U) *\* (V *\* U) = (T *\* V) *\* (U *\* V)
    using Cube-ind by (metis Con-sym Resid.simps(1) order-refl)
qed

lemma Con-implies-Arr:
assumes T *⌢* U
shows Arr T and Arr U
  using assms Con-sym
  by (metis Con-imp-Arr-Resid Arr-iff-Con-self Cube(1) Resid.simps(1))+

sublocale partial-magma Resid
  by (unfold-locales, metis Resid.simps(1) Con-sym)

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

lemma null-char:
shows null = []
  by (metis null-is-zero(2) Resid.simps(1))

sublocale residuation Resid
  using null-char Con-sym Arr-iff-Con-self Con-imp-Arr-Resid Cube null-is-zero(2)
```

by *unfold-locales auto*

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

lemma *arr-char*:
shows *arr T* $\longleftrightarrow$ *Arr T*
using *null-char Arr-iff-Con-self* **by** *fastforce*

lemma *arrI*$_P$ [*intro*]:
assumes *Arr T*
shows *arr T*
using *assms arr-char* **by** *auto*

lemma *ide-char*:
shows *ide T* $\longleftrightarrow$ *Ide T*
by (*metis Con-Arr-self Ide-implies-Arr Resid-Arr-Ide-ind Resid-Arr-self arr-char ide-def arr-def*)

lemma *con-char*:
shows *con T U* $\longleftrightarrow$ *Con T U*
using *null-char* **by** *auto*

lemma *conI*$_P$ [*intro*]:
assumes *Con T U*
shows *con T U*
using *assms con-char* **by** *auto*

sublocale *rts Resid*
proof
show $\bigwedge A\ T.\ [\![ide\ A;\ con\ T\ A]\!] \Longrightarrow T\ {}^*\backslash^*\ A = T$
using *Resid-Arr-Ide-ind ide-char null-char* **by** *auto*
show $\bigwedge T.\ arr\ T \Longrightarrow ide\ (trg\ T)$
by (*metis arr-char Resid-Arr-self ide-char resid-arr-self*)
show $\bigwedge A\ T.\ [\![ide\ A;\ con\ A\ T]\!] \Longrightarrow ide\ (A\ {}^*\backslash^*\ T)$
by (*simp add*: *Resid-Ide-Arr-ind con-char ide-char*)
show $\bigwedge T\ U.\ con\ T\ U \Longrightarrow \exists A.\ ide\ A \wedge con\ A\ T \wedge con\ A\ U$
proof −
fix *T U*
assume *TU*: *con T U*
have *1*: *Srcs T* = *Srcs U*
using *TU Con-imp-eq-Srcs con-char* **by** *force*
obtain *a* **where** *a*: $a \in Srcs\ T \cap Srcs\ U$
using *1*
by (*metis Int-absorb Int-emptyI TU arr-char Arr-has-Src con-implies-arr*(*1*))
show $\exists A.\ ide\ A \wedge con\ A\ T \wedge con\ A\ U$
using *a 1*
by (*metis* (*full-types*) *Ball-Collect Con-single-ide-ind Ide.simps*(*2*) *Int-absorb TU*

```
      Srcs-are-ide arr-char con-char con-implies-arr(1−2) ide-char)
  qed
  show ⋀T U V. ⟦ide (Resid T U); con U V⟧ ⟹ con (T *\* U) (V *\* U)
    using null-char ide-char
    by (metis Con-imp-Arr-Resid Con-Ide-iff Srcs-Resid con-char con-sym arr-resid-iff-con
        ide-implies-arr)
qed

theorem is-rts:
shows rts Resid
  ..

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

lemma sources-charP:
shows sources T = {A. Ide A ∧ Arr T ∧ Srcs A = Srcs T}
  using Con-Ide-iff Con-sym con-char ide-char sources-def by fastforce

lemma sources-cons:
shows Arr (t # T) ⟹ sources (t # T) = sources [t]
  apply (induct T)
   apply simp
  using sources-charP by auto

lemma targets-charP:
shows targets T = {B. Ide B ∧ Arr T ∧ Srcs B = Trgs T}
  unfolding targets-def
  by (metis (no-types, lifting) trg-def Arr.simps(1) Ide-implies-Arr Resid-Arr-self
      arr-char Con-Ide-iff Srcs-Resid con-char ide-char con-implies-arr(1))

lemma seq-char':
shows seq T U ⟷ Arr T ∧ Arr U ∧ Trgs T ∩ Srcs U ≠ {}
proof
  show seq T U ⟹ Arr T ∧ Arr U ∧ Trgs T ∩ Srcs U ≠ {}
    unfolding seq-def
    using Arr-has-Trg arr-char Con-Arr-self sources-charP trg-def trg-in-targets
    by fastforce
  assume 1: Arr T ∧ Arr U ∧ Trgs T ∩ Srcs U ≠ {}
  have targets T = sources U
  proof −
    obtain a where a: R.ide a ∧ a ∈ Trgs T ∧ a ∈ Srcs U
      using 1 Trgs-are-ide by blast
    have Trgs [a] = Trgs T
      using a 1
      by (metis Con-single-ide-ind Con-sym Resid-Arr-Src Srcs-Resid Trgs-eqI)
    moreover have Srcs [a] = Srcs U
      using a 1 Con-single-ide-ind Con-imp-eq-Srcs by blast
    moreover have Trgs [a] = Srcs [a]
```

```
      using a
      by (metis R.sources-resid Srcs.simps(2) Trgs.simps(2) R.ideE)
    ultimately show ?thesis
      using 1 sources-char_P targets-char_P by auto
  qed
  thus seq T U
    using 1 by blast
qed

lemma seq-char:
shows seq T U ⟷ Arr T ∧ Arr U ∧ Trgs T = Srcs U
  by (metis Int-absorb Srcs-Resid Arr-has-Src Arr-iff-Con-self Srcs-eqI seq-char')

lemma seqI_P [intro]:
assumes Arr T and Arr U and Trgs T ∩ Srcs U ≠ {}
shows seq T U
  using assms seq-char' by auto

lemma coinitial-char:
shows coinitial T U ⟹ Arr T ∧ Arr U ∧ Srcs T = Srcs U
and Arr T ∧ Arr U ∧ Srcs T ∩ Srcs U ≠ {} ⟹ coinitial T U
proof −
  show coinitial T U ⟹ Arr T ∧ Arr U ∧ Srcs T = Srcs U
    unfolding seq-def
    by (metis Con-imp-eq-Srcs arr-char coinitial-iff
        con-char prfx-implies-con source-is-prfx src-in-sources)
  assume 1: Arr T ∧ Arr U ∧ Srcs T ∩ Srcs U ≠ {}
  have sources T = sources U
  proof −
    obtain a where a: R.ide a ∧ a ∈ Srcs T ∧ a ∈ Srcs U
      using 1 Srcs-are-ide by blast
    have Srcs [a] = Srcs T
      using a 1
      by (metis Arr.simps(1) Con-imp-eq-Srcs Resid-Arr-Src)
    moreover have Srcs [a] = Srcs U
      using a 1 Con-single-ide-ind Con-imp-eq-Srcs by blast
    ultimately show ?thesis
      using 1 sources-char_P targets-char_P by auto
  qed
  thus coinitial T U
    using 1 by blast
qed

lemma coinitialI_P [intro]:
assumes Arr T and Arr U and Srcs T ∩ Srcs U ≠ {}
shows coinitial T U
  using assms coinitial-char(2) by auto

lemma Ide-imp-sources-eq-targets:
```

assumes *Ide T*
shows *sources T = targets T*
using *assms*
by (*metis Resid-Arr-Ide-ind arr-iff-has-source arr-iff-has-target con-char arr-def sources-resid*)

2.4.2 Inclusion Map

Inclusion of an RTS to the RTS of its paths.

abbreviation *incl*
where *incl* ≡ λ*t*. *if R.arr t then* [*t*] *else null*

sublocale *incl*: *simulation resid Resid incl*
using *R.con-implies-arr*(*1*−*2*) *con-char R.arr-resid-iff-con null-char*
by *unfold-locales auto*

lemma *incl-is-simulation*:
shows *simulation resid Resid incl*
..

lemma *incl-is-injective*:
shows *inj-on incl* (*Collect R.arr*)
by (*intro inj-onI*) *simp*

lemma *reflects-con*:
assumes *incl t* ⁀* *incl u*
shows *t* ⁀ *u*
using *assms*
by (*metis* (*full-types*) *Arr.simps*(*1*) *Con-implies-Arr*(*1*−*2*) *Con-rec*(*1*) *null-char*)

end

2.4.3 Composites of Paths

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

context *paths-in-rts*
begin

lemma *Srcs-append* [*simp*]:
assumes *T* ≠ []
shows *Srcs* (*T* @ *U*) = *Srcs T*
by (*metis Nil-is-append-conv Srcs.simps*(*2*) *Srcs.simps*(*3*) *assms hd-append list.exhaust-sel*)

lemma *Trgs-append* [*simp*]:
shows *U* ≠ [] ⟹ *Trgs* (*T* @ *U*) = *Trgs U*
proof (*induct T*)
show *U* ≠ [] ⟹ *Trgs* ([] @ *U*) = *Trgs U*
by *auto*
show ⋀*t T*. ⟦*U* ≠ [] ⟹ *Trgs* (*T* @ *U*) = *Trgs U*; *U* ≠ []⟧

$\Longrightarrow$ *Trgs* ((*t* # *T*) @ *U*) = *Trgs U*
by (*metis Nil-is-append-conv Trgs.simps*(*3*) *append-Cons list.exhaust*)
qed

lemma *seq-implies-Trgs-eq-Srcs*:
shows ⟦*Arr T*; *Arr U*; *Trgs T* ⊆ *Srcs U*⟧ $\Longrightarrow$ *Trgs T* = *Srcs U*
by (*metis inf.orderE Arr-has-Trg seqI*$_P$ *seq-char*)

lemma *Arr-append-iff*$_P$:
shows ⟦*T* ≠ []; *U* ≠ []⟧ $\Longrightarrow$ *Arr* (*T* @ *U*) ⟷ *Arr T* ∧ *Arr U* ∧ *Trgs T* ⊆ *Srcs U*
proof (*induct T arbitrary*: *U*)
show ⋀*U*. ⟦[] ≠ []; *U* ≠ []⟧ $\Longrightarrow$ *Arr* ([] @ *U*) = (*Arr* [] ∧ *Arr U* ∧ *Trgs* [] ⊆ *Srcs U*)
by *simp*
fix *t T* **and** *U* :: *'a list*
assume *ind*: ⋀*U*. ⟦*T* ≠ []; *U* ≠ []⟧
$\Longrightarrow$ *Arr* (*T* @ *U*) = (*Arr T* ∧ *Arr U* ∧ *Trgs T* ⊆ *Srcs U*)
assume *U*: *U* ≠ []
show *Arr* ((*t* # *T*) @ *U*) ⟷ *Arr* (*t* # *T*) ∧ *Arr U* ∧ *Trgs* (*t* # *T*) ⊆ *Srcs U*
proof (*cases T* = [])
show *T* = [] $\Longrightarrow$ *?thesis*
using *Arr.elims*(*1*) *U* **by** *auto*
assume *T*: *T* ≠ []
have *Arr* ((*t* # *T*) @ *U*) ⟷ *Arr* (*t* # (*T* @ *U*))
by *simp*
also have ... ⟷ *R.arr t* ∧ *Arr* (*T* @ *U*) ∧ *R.targets t* ⊆ *Srcs* (*T* @ *U*)
using *T U*
by (*metis Arr.simps*(*3*) *Nil-is-append-conv neq-Nil-conv*)
also have ... ⟷ *R.arr t* ∧ *Arr T* ∧ *Arr U* ∧ *Trgs T* ⊆ *Srcs U* ∧ *R.targets t* ⊆ *Srcs T*
using *T U ind* **by** *auto*
also have ... ⟷ *Arr* (*t* # *T*) ∧ *Arr U* ∧ *Trgs* (*t* # *T*) ⊆ *Srcs U*
using *T U*
by (*metis Arr.simps*(*3*) *Trgs.simps*(*3*) *neq-Nil-conv*)
finally show *?thesis* **by** *auto*
qed
qed

lemma *Arr-consI*$_P$ [*intro*, *simp*]:
assumes *R.arr t* **and** *Arr U* **and** *R.targets t* ⊆ *Srcs U*
shows *Arr* (*t* # *U*)
using *assms Arr.elims*(*3*) **by** *blast*

lemma *Arr-appendI*$_P$ [*intro*, *simp*]:
assumes *Arr T* **and** *Arr U* **and** *Trgs T* ⊆ *Srcs U*
shows *Arr* (*T* @ *U*)
using *assms*
by (*metis Arr.simps*(*1*) *Arr-append-iff*$_P$)

lemma *Arr-appendE*$_P$ [*elim*]:
assumes *Arr* (*T* @ *U*) **and** *T* ≠ [] **and** *U* ≠ []

and $[\![Arr\ T;\ Arr\ U;\ Trgs\ T = Srcs\ U]\!] \Longrightarrow thesis$
shows *thesis*
using *assms Arr-append-iff*$_P$ *seq-implies-Trgs-eq-Srcs* **by** *force*

lemma *Ide-append-iff*$_P$:
shows $[\![T \neq []; U \neq []]\!] \Longrightarrow Ide\ (T\ @\ U) \longleftrightarrow Ide\ T \wedge Ide\ U \wedge Trgs\ T \subseteq Srcs\ U$
using *Ide-char* **by** *auto*

lemma *Ide-appendI*$_P$ [*intro*, *simp*]:
assumes *Ide T* **and** *Ide U* **and** $Trgs\ T \subseteq Srcs\ U$
shows $Ide\ (T\ @\ U)$
using *assms*
by (*metis Ide.simps*(1) *Ide-append-iff*$_P$)

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 (*induct V arbitrary*: *T U*)
show $\bigwedge T\ U.\ [\![T \neq []; U \neq []; [] \neq []]\!] \Longrightarrow$
$([]\ @\ T\ *\frown^*\ U \longleftrightarrow []\ *\frown^*\ U \wedge T\ *\frown^*\ U\ *\backslash^*\ []) \wedge$
$(T\ *\frown^*\ []\ @\ U \longleftrightarrow T\ *\frown^*\ [] \wedge T\ *\backslash^*\ []\ *\frown^*\ U) \wedge$
$([]\ @\ T\ *\frown^*\ U \longrightarrow ([]\ @\ T)\ *\backslash^*\ U = []\ *\backslash^*\ U\ @\ T\ *\backslash^*\ (U\ *\backslash^*\ [])) \wedge$
$(T\ *\frown^*\ []\ @\ U \longrightarrow T\ *\backslash^*\ ([]\ @\ U) = (T\ *\backslash^*\ [])\ *\backslash^*\ U)$
by *simp*
fix $v :: 'a$ **and** *T U V* :: *'a list*
assume *ind*: $\bigwedge T\ U.\ [\![T \neq []; U \neq []; V \neq []]\!] \Longrightarrow$
$(V\ @\ T\ *\frown^*\ U \longleftrightarrow V\ *\frown^*\ U \wedge T\ *\frown^*\ U\ *\backslash^*\ V) \wedge$
$(T\ *\frown^*\ V\ @\ U \longleftrightarrow T\ *\frown^*\ V \wedge T\ *\backslash^*\ V\ *\frown^*\ U) \wedge$
$(V\ @\ T\ *\frown^*\ U \longrightarrow (V\ @\ T)\ *\backslash^*\ U = V\ *\backslash^*\ U\ @\ T\ *\backslash^*\ (U\ *\backslash^*\ V)) \wedge$
$(T\ *\frown^*\ V\ @\ U \longrightarrow T\ *\backslash^*\ (V\ @\ U) = (T\ *\backslash^*\ V)\ *\backslash^*\ U)$
assume *T*: $T \neq []$ **and** *U*: $U \neq []$
show $((v\ \#\ V)\ @\ T\ *\frown^*\ U \longleftrightarrow (v\ \#\ V)\ *\frown^*\ U \wedge T\ *\frown^*\ U\ *\backslash^*\ (v\ \#\ V)) \wedge$
$(T\ *\frown^*\ (v\ \#\ V)\ @\ U \longleftrightarrow T\ *\frown^*\ (v\ \#\ V) \wedge T\ *\backslash^*\ (v\ \#\ V)\ *\frown^*\ U) \wedge$
$((v\ \#\ V)\ @\ T\ *\frown^*\ U \longrightarrow$
$((v\ \#\ V)\ @\ T)\ *\backslash^*\ U = (v\ \#\ V)\ *\backslash^*\ U\ @\ T\ *\backslash^*\ (U\ *\backslash^*\ (v\ \#\ V))) \wedge$
$(T\ *\frown^*\ (v\ \#\ V)\ @\ U \longrightarrow T\ *\backslash^*\ ((v\ \#\ V)\ @\ U) = (T\ *\backslash^*\ (v\ \#\ V))\ *\backslash^*\ U)$
proof (*intro conjI iffI impI*)
show *1*: $(v\ \#\ V)\ @\ T\ *\frown^*\ U \Longrightarrow$
$((v\ \#\ V)\ @\ T)\ *\backslash^*\ U = (v\ \#\ V)\ *\backslash^*\ U\ @\ T\ *\backslash^*\ (U\ *\backslash^*\ (v\ \#\ V))$
proof (*cases V* = [])
show $V = [] \Longrightarrow (v\ \#\ V)\ @\ T\ *\frown^*\ U \Longrightarrow ?thesis$
using *T U Resid-cons*(1) *U* **by** *auto*
assume *V*: $V \neq []$
assume *Con*: $(v\ \#\ V)\ @\ T\ *\frown^*\ U$
have $((v\ \#\ V)\ @\ T)\ *\backslash^*\ U = (v\ \#\ (V\ @\ T))\ *\backslash^*\ U$
by *simp*

```
  also have ... = [v] *\* U @ (V @ T) *\* (U *\* [v])
    using T U Con Resid-cons by simp
  also have ... = [v] *\* U @ V *\* (U *\* [v]) @ T *\* ((U *\* [v]) *\* V)
    using T U V Con ind Resid-cons
    by (metis Con-sym Cons-eq-appendI append-is-Nil-conv Con-cons(1))
  also have ... = (v # V) *\* U @ T *\* (U *\* (v # V))
    using ind[of T]
    by (metis Con Con-cons(2) Cons-eq-appendI Resid-cons(1) Resid-cons(2) T U V
        append.assoc append-is-Nil-conv Con-sym)
  finally show ?thesis by simp
qed
show 2: T *⌢* (v # V) @ U ⟹ T *\* ((v # V) @ U) = (T *\* (v # V)) *\* U
proof (cases V = [])
  show V = [] ⟹ T *⌢* (v # V) @ U ⟹ ?thesis
    using Resid-cons(2) T U by auto
  assume V: V ≠ []
  assume Con: T *⌢* (v # V) @ U
  have T *\* ((v # V) @ U) = T *\* (v # (V @ U))
    by simp
  also have 1: ... = (T *\* [v]) *\* (V @ U)
    using V Con Resid-cons(2) T by force
  also have ... = ((T *\* [v]) *\* V) *\* U
    using T U V 1 Con ind
    by (metis Con-initial-right Cons-eq-appendI)
  also have ... = (T *\* (v # V)) *\* U
    using T V Con
    by (metis Con-cons(2) Con-initial-right Cons-eq-appendI Resid-cons(2))
  finally show ?thesis by blast
qed
show (v # V) @ T *⌢* U ⟹ v # V *⌢* U
  by (metis 1 Con-sym Resid.simps(1) append-Nil)
show (v # V) @ T *⌢* U ⟹ T *⌢* U *\* (v # V)
  using T U Con-sym
  by (metis 1 Con-initial-right Resid-cons(1−2) append.simps(2) ind self-append-conv)
show T *⌢* (v # V) @ U ⟹ T *⌢* v # V
  using 2 by fastforce
show T *⌢* (v # V) @ U ⟹ T *\* (v # V) *⌢* U
  using 2 by fastforce
show T *⌢* v # V ∧ T *\* (v # V) *⌢* U ⟹ T *⌢* (v # V) @ U
proof −
  assume Con: T *⌢* v # V ∧ T *\* (v # V) *⌢* U
  have T *⌢* (v # V) @ U ⟷ T *⌢* v # (V @ U)
    by simp
  also have ... ⟷ T *⌢* [v] ∧ T *\* [v] *⌢* V @ U
    using T U Con-cons(2) by simp
  also have ... ⟷ T *\* [v] *⌢* V @ U
    by fastforce
  also have ... ⟷ True
    using Con ind
```

```
      by (metis Con-cons(2) Resid-cons(2) T U self-append-conv2)
    finally show ?thesis by blast
  qed
  show v # V *⌢* U ∧ T *⌢* U *\* (v # V) ⟹ (v # V) @ T *⌢* U
  proof −
    assume Con: v # V *⌢* U ∧ T *⌢* U *\* (v # V)
    have (v # V) @ T *⌢* U ⟷ v # (V @ T) *⌢* U
      by simp
    also have ... ⟷ [v] *⌢* U ∧ V @ T *⌢* U *\* [v]
      using T U Con-cons(1) by simp
    also have ... ⟷ V @ T *⌢* U *\* [v]
      by (metis Con Con-cons(1) U)
    also have ... ⟷ True
      using Con ind
      by (metis Con-cons(1) Con-sym Resid-cons(2) T U append-self-conv2)
    finally show ?thesis by blast
  qed
 qed
qed

lemma Con-append:
assumes T ≠ [] and U ≠ [] and V ≠ []
shows T @ U *⌢* V ⟷ T *⌢* V ∧ U *⌢* V *\* T
and T *⌢* U @ V ⟷ T *⌢* U ∧ T *\* U *⌢* V
  using assms Resid-append-ind by blast+

lemma Con-appendI [intro]:
shows ⟦T *⌢* V; U *⌢* V *\* T⟧ ⟹ T @ U *⌢* V
and ⟦T *⌢* U; T *\* U *⌢* V⟧ ⟹ T *⌢* U @ V
  by (metis Con-append(1) Con-sym Resid.simps(1))+

lemma Resid-append [intro, simp]:
shows ⟦T ≠ []; T @ U *⌢* V⟧ ⟹ (T @ U) *\* V = (T *\* V) @ (U *\* (V *\* T))
and ⟦U ≠ []; V ≠ []; T *⌢* U @ V⟧ ⟹ T *\* (U @ V) = (T *\* U) *\* V
  using Resid-append-ind
   apply (metis Con-sym Resid.simps(1) append-self-conv)
  using Resid-append-ind
  by (metis Resid.simps(1))

lemma Resid-append2 [simp]:
assumes T ≠ [] and U ≠ [] and V ≠ [] and W ≠ []
and T @ U *⌢* V @ W
shows (T @ U) *\* (V @ W) =
      (T *\* V) *\* W @ (U *\* (V *\* T)) *\* (W *\* (T *\* V))
  using assms Resid-append
  by (metis Con-append(1−2) append-is-Nil-conv)

lemma append-is-composite-of:
assumes seq T U
```

shows *composite-of T U* (*T* @ *U*)
unfolding *composite-of-def*
using *assms*
apply (*intro conjI*)
apply (*metis Arr.simps*(*1*) *Resid-Arr-self Resid-Ide-Arr-ind Arr-appendI*$_P$
Resid-append-ind ide-char order-refl seq-char)
apply (*metis Arr.simps*(*1*) *Arr-appendI*$_P$ *Con-Arr-self Resid-Arr-self Resid-append-ind*
ide-char seq-char order-refl)
by (*metis Arr.simps*(*1*) *Con-Arr-self Con-append*(*1*) *Resid-Arr-self Arr-appendI*$_P$
Ide-append-iff$_P$ *Resid-append*(*1*) *ide-char seq-char order-refl*)

sublocale *rts-with-composites Resid*
using *append-is-composite-of composable-def* **by** *unfold-locales blast*

theorem *is-rts-with-composites*:
shows *rts-with-composites Resid*
..

lemma *arr-append* [*intro*, *simp*]:
assumes *seq T U*
shows *arr* (*T* @ *U*)
using *assms arrI*$_P$ *seq-char* **by** *simp*

lemma *arr-append-imp-seq*:
assumes *T* ≠ [] **and** *U* ≠ [] **and** *arr* (*T* @ *U*)
shows *seq T U*
using *assms arr-char seq-char Arr-append-iff*$_P$ *seq-implies-Trgs-eq-Srcs* **by** *simp*

lemma *sources-append* [*simp*]:
assumes *seq T U*
shows *sources* (*T* @ *U*) = *sources T*
using *assms*
by (*meson append-is-composite-of sources-composite-of*)

lemma *targets-append* [*simp*]:
assumes *seq T U*
shows *targets* (*T* @ *U*) = *targets U*
using *assms*
by (*meson append-is-composite-of targets-composite-of*)

lemma *cong-respects-seq*$_P$:
assumes *seq T U* **and** *T* $^*\sim^*$ *T'* **and** *U* $^*\sim^*$ *U'*
shows *seq T' U'*
by (*meson assms cong-respects-seq*)

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

```
proof
  have 1: ⋀T U T' U'. ⟦seq T U; T *∼* T'; U *∼* U'⟧ ⟹ seq T' U'
    using assms cong-respects-seq_P by simp
  have 2: ⋀T U T' U'. ⟦seq T U; T *∼* T'; U *∼* U'⟧ ⟹ T @ U *≲* T' @ U'
  proof −
    fix T U T' U'
    assume TU: seq T U and TT': T *∼* T' and UU': U *∼* U'
    have T'U': seq T' U'
      using TU TT' UU' cong-respects-seq_P by simp
    have 3: Ide (T *\* T') ∧ Ide (T' *\* T) ∧ Ide (U *\* U') ∧ Ide (U' *\* U)
      using TU TT' UU' ide-char by blast
    have (T @ U) *\* (T' @ U') =
          ((T *\* T') *\* U') @ U *\* ((T' *\* T) @ U' *\* (T *\* T'))
    proof −
      have 4: T ≠ [] ∧ U ≠ [] ∧ T' ≠ [] ∧ U' ≠ []
        using TU TT' UU' Arr.simps(1) seq-char ide-char by auto
      moreover have (T @ U) *\* (T' @ U') ≠ []
      proof (intro Con-appendI)
        show T *\* T' ≠ []
          using 3 by force
        show (T *\* T') *\* U' ≠ []
          using 3 T'U' ‹T *\* T' ≠ []› Con-Ide-iff seq-char by fastforce
        show U *\* ((T' @ U') *\* T) ≠ []
        proof −
          have U *\* ((T' @ U') *\* T) = U *\* ((T' *\* T) @ U' *\* (T *\* T'))
            by (metis Con-appendI(1) Resid-append(1) ‹(T *\* T') *\* U' ≠ []›
                ‹T *\* T' ≠ []› calculation Con-sym)
          also have ... = (U *\* (T' *\* T)) *\* (U' *\* (T *\* T'))
            by (metis Arr.simps(1) Con-append(2) Resid-append(2) ‹(T *\* T') *\* U' ≠ []›
                Con-implies-Arr(1) Con-sym)
          also have ... = U *\* U'
            by (metis (mono-tags, lifting) 3 Ide.simps(1) Resid-Ide(1) Srcs-Resid TU
                ‹(T *\* T') *\* U' ≠ []› Con-Ide-iff seq-char)
          finally show ?thesis
            using 3 UU' by force
        qed
      qed
      ultimately show ?thesis
        using Resid-append2 [of T U T' U'] seq-char
        by (metis Con-append(2) Con-sym Resid-append(2) Resid.simps(1))
    qed
    moreover have Ide ...
    proof
      have 3: Ide (T *\* T') ∧ Ide (T' *\* T) ∧ Ide (U *\* U') ∧ Ide (U' *\* U)
        using TU TT' UU' ide-char by blast
      show 4: Ide ((T *\* T') *\* U')
        using TU T'U' TT' UU' 1 3
        by (metis (full-types) Srcs-Resid Con-Ide-iff Resid-Ide-Arr-ind seq-char)
      show 5: Ide (U *\* ((T' *\* T) @ U' *\* (T *\* T')))
```

```
      proof −
        have U *\* (T' *\* T) = U
          by (metis (full-types) 3 TT' TU Con-Ide-iff Resid-Ide(1) Srcs-Resid
              con-char seq-char prfx-implies-con)
        moreover have U' *\* (T *\* T') = U'
          by (metis 3 4 Ide.simps(1) Resid-Ide(1))
        ultimately show ?thesis
          by (metis 3 4 Arr.simps(1) Con-append(2) Ide.simps(1) Resid-append(2)
              TU Con-sym seq-char)
      qed
      show Trgs ((T *\* T') *\* U') ⊆ Srcs (U *\* (T' *\* T @ U' *\* (T *\* T')))
        by (metis 4 5 Arr-append-iff_P Ide.simps(1) Nil-is-append-conv
            calculation Con-imp-Arr-Resid)
    qed
    ultimately show T @ U *≲* T' @ U'
      using ide-char by presburger
  qed
  show T @ U *≲* T' @ U'
    using assms 2 by simp
  show T' @ U' *≲* T @ U
    using assms 1 2 cong-symmetric by blast
qed

lemma cong-cons [intro]:
assumes seq [t] U and t ∼ t' and U *∼* U'
shows t # U *∼* t' # U'
  using assms cong-append [of [t] U [t'] U']
  by (simp add: R.prfx-implies-con ide-char)

lemma cong-append-ideI [intro]:
assumes seq T U
shows ide T ⟹ T @ U *∼* U and ide U ⟹ T @ U *∼* T
and ide T ⟹ U *∼* T @ U and ide U ⟹ T *∼* T @ U
proof −
  show 1: ide T ⟹ T @ U *∼* U
    using assms
    by (metis append-is-composite-of composite-ofE resid-arr-ide prfx-implies-con
        con-sym)
  show 2: ide U ⟹ T @ U *∼* T
    by (meson assms append-is-composite-of composite-ofE ide-backward-stable)
  show ide T ⟹ U *∼* T @ U
    using 1 cong-symmetric by auto
  show ide U ⟹ T *∼* T @ U
    using 2 cong-symmetric by auto
qed

lemma cong-cons-ideI [intro]:
assumes seq [t] U and R.ide t
shows t # U *∼* U and U *∼* t # U
```

```
  using assms cong-append-ideI [of [t] U]
  by (auto simp add: ide-char)

lemma prfx-decomp:
assumes [t] *≲* [u]
shows [t] @ [u \ t] *∼* [u]
proof

  show 1: [u] *≲* [t] @ [u \ t]
    using assms
    by (metis Con-imp-Arr-Resid Con-rec(3) Resid.simps(3) Resid-rec(3) R.con-sym
        append.left-neutral append-Cons arr-char cong-reflexive list.distinct(1))
  show [t] @ [u \ t] *≲* [u]
  proof −
    have ([t] @ [u \ t]) *\* [u] = ([t] *\* [u]) @ ([u \ t] *\* [u \ t])
      using assms
      by (metis Arr-Resid-single Con-Arr-self Con-appendI(1) Con-sym Resid-append(1)
          Resid-rec(1) con-char list.discI prfx-implies-con)
    moreover have Ide ...
      using assms
      by (metis 1 Con-sym append-Nil2 arr-append-imp-seq calculation cong-append-ideI(4)
          ide-backward-stable Con-implies-Arr(2) Resid-Arr-self con-char ide-char
          prfx-implies-con arr-resid-iff-con)
    ultimately show ?thesis
      using ide-char by presburger
  qed
qed

lemma composite-of-single-single:
assumes R.composite-of t u v
shows composite-of [t] [u] ([t] @ [u])
proof
  show [t] *≲* [t] @ [u]
  proof −
    have [t] *\* ([t] @ [u]) = ([t] *\* [t]) *\* [u]
      using assms by auto
    moreover have Ide ...
      by (metis (no-types, lifting) Con-implies-Arr(2) R.bounded-imp-con
          R.con-composite-of-iff R.con-prfx-composite-of(1) assms resid-ide-arr
          Con-rec(1) Resid.simps(3) Resid-Arr-self con-char ide-char)
    ultimately show ?thesis
      using ide-char by presburger
  qed
  show ([t] @ [u]) *\* [t] *∼* [u]
    using assms
    by (metis ‹prfx [t] ([t] @ [u])› append-is-composite-of arr-append-imp-seq
        composite-ofE con-def not-Cons-self2 Con-implies-Arr(2) arr-char null-char
        prfx-implies-con)
qed
```

end

2.4.4 Paths in a Weakly Extensional RTS

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

lemma *ex-un-Src*:
assumes *Arr T*
shows ∃!*a*. *a* ∈ *Srcs T*
using *assms*
by (*simp add*: *Srcs-simp*$_P$ *R.arr-has-un-source*)

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

lemma *Srcs-simp*$_{PWE}$:
assumes *Arr T*
shows *Srcs T* = {*Src T*}
proof −
have [*R.src* (*hd T*)] ∈ *sources T*
by (*metis Arr-imp-arr-hd Con-single-ide-ind Ide.simps*(*2*) *Srcs-simp*$_P$ *assms*
con-char ide-char in-sourcesI con-sym R.ide-src R.src-in-sources)
hence *R.src* (*hd T*) ∈ *Srcs T*
using *assms*
by (*metis Srcs.elims Arr-has-Src list.sel*(*1*) *R.arr-iff-has-source R.src-in-sources*)
thus *?thesis*
using *assms ex-un-Src* **by** *auto*
qed

lemma *ex-un-Trg*:
assumes *Arr T*
shows ∃!*b*. *b* ∈ *Trgs T*
using *assms*
apply (*induct T*)
apply *auto*[*1*]
by (*metis Con-Arr-self Ide-implies-Arr Resid-Arr-self Srcs-Resid ex-un-Src*)

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

lemma *Trg-simp* [*simp*]:
shows *T* ≠ [] ⟹ *Trg T* = *R.trg* (*last T*)
apply (*induct T*)

apply *auto*
by (*metis Trg.simps*(*3*) *list.exhaust-sel*)

lemma *Trgs-simp*$_{PWE}$ [*simp*]:
assumes *Arr T*
shows *Trgs T* = {*Trg T*}
using *assms*
by (*metis Arr-imp-arr-last Con-Arr-self Con-imp-Arr-Resid R.trg-in-targets Srcs.simps*(*1*) *Srcs-Resid Srcs-simp*$_{PWE}$ *Trg-simp insertE insert-absorb insert-not-empty Trgs-simp*$_{P}$)

lemma *Src-resid* [*simp*]:
assumes *T* $*\frown^*$ *U*
shows *Src* (*T* $*\backslash^*$ *U*) = *Trg U*
using *assms Con-imp-Arr-Resid Con-implies-Arr*(*2*) *Srcs-Resid Srcs-simp*$_{PWE}$ **by** *force*

lemma *Trg-resid-sym*:
assumes *T* $*\frown^*$ *U*
shows *Trg* (*T* $*\backslash^*$ *U*) = *Trg* (*U* $*\backslash^*$ *T*)
using *assms Con-imp-Arr-Resid Con-sym Trgs-Resid-sym* **by** *auto*

lemma *Src-append* [*simp*]:
assumes *seq T U*
shows *Src* (*T* @ *U*) = *Src T*
using *assms*
by (*metis Arr.simps*(*1*) *Src.simps hd-append seq-char*)

lemma *Trg-append* [*simp*]:
assumes *seq T U*
shows *Trg* (*T* @ *U*) = *Trg U*
using *assms*
by (*metis Ide.simps*(*1*) *Resid.simps*(*1*) *Trg-simp append-is-Nil-conv ide-char ide-trg last-appendR seqE trg-def*)

lemma *Arr-append-iff*$_{PWE}$:
assumes *T* ≠ [] **and** *U* ≠ []
shows *Arr* (*T* @ *U*) ⟷ *Arr T* ∧ *Arr U* ∧ *Trg T* = *Src U*
using *assms Arr-appendE*$_{P}$ *Srcs-simp*$_{PWE}$ **by** *auto*

lemma *Arr-consI*$_{PWE}$ [*intro*, *simp*]:
assumes *R.arr t* **and** *Arr U* **and** *R.trg t* = *Src U*
shows *Arr* (*t* # *U*)
using *assms*
by (*metis Arr.simps*(*2*) *Srcs-simp*$_{PWE}$ *Trg.simps*(*2*) *Trgs.simps*(*2*) *Trgs-simp*$_{PWE}$ *dual-order.eq-iff Arr-consI*$_{P}$)

lemma *Arr-consE* [*elim*]:
assumes *Arr* (*t* # *U*)
and ⟦*R.arr t*; *U* ≠ [] ⟹ *Arr U*; *U* ≠ [] ⟹ *R.trg t* = *Src U*⟧ ⟹ *thesis*

shows *thesis*
using *assms*
by (*metis Arr-append-iff*$_{PWE}$ *Trg.simps*(*2*) *append-Cons append-Nil list.distinct*(*1*) *Arr.simps*(*2*))

lemma *Arr-appendI*$_{PWE}$ [*intro*, *simp*]:
assumes *Arr T* **and** *Arr U* **and** *Trg T* = *Src U*
shows *Arr* (*T* @ *U*)
using *assms*
by (*metis Arr.simps*(*1*) *Arr-append-iff*$_{PWE}$)

lemma *Arr-appendE*$_{PWE}$ [*elim*]:
assumes *Arr* (*T* @ *U*) **and** *T* ≠ [] **and** *U* ≠ []
and ⟦*Arr T*; *Arr U*; *Trg T* = *Src U*⟧ ⟹ *thesis*
shows *thesis*
using *assms Arr-append-iff*$_{PWE}$ *seq-implies-Trgs-eq-Srcs* **by** *force*

lemma *Ide-append-iff*$_{PWE}$:
assumes *T* ≠ [] **and** *U* ≠ []
shows *Ide* (*T* @ *U*) ⟷ *Ide T* ∧ *Ide U* ∧ *Trg T* = *Src U*
using *assms Ide-char*
apply (*intro iffI*)
by *force auto*

lemma *Ide-appendI*$_{PWE}$ [*intro*, *simp*]:
assumes *Ide T* **and** *Ide U* **and** *Trg T* = *Src U*
shows *Ide* (*T* @ *U*)
using *assms*
by (*metis Ide.simps*(*1*) *Ide-append-iff*$_{PWE}$)

lemma *Ide-appendE* [*elim*]:
assumes *Ide* (*T* @ *U*) **and** *T* ≠ [] **and** *U* ≠ []
and ⟦*Ide T*; *Ide U*; *Trg T* = *Src U*⟧ ⟹ *thesis*
shows *thesis*
using *assms Ide-append-iff*$_{PWE}$ **by** *metis*

lemma *Ide-consI* [*intro*, *simp*]:
assumes *R.ide t* **and** *Ide U* **and** *R.trg t* = *Src U*
shows *Ide* (*t* # *U*)
using *assms*
by (*simp add*: *Ide-char*)

lemma *Ide-consE* [*elim*]:
assumes *Ide* (*t* # *U*)
and ⟦*R.ide t*; *U* ≠ [] ⟹ *Ide U*; *U* ≠ [] ⟹ *R.trg t* = *Src U*⟧ ⟹ *thesis*
shows *thesis*
using *assms*
by (*metis Con-rec*(*4*) *Ide.simps*(*2*) *Ide-imp-Ide-hd Ide-imp-Ide-tl R.trg-def R.trg-ide Resid-Arr-Ide-ind Trg.simps*(*2*) *ide-char list.sel*(*1*) *list.sel*(*3*) *list.simps*(*3*)

Src-resid ide-def)

lemma *Ide-imp-Src-eq-Trg*:
assumes *Ide T*
shows *Src T = Trg T*
using *assms*
by (*metis Ide.simps*(*1*) *Src-resid ide-char ide-def*)

end

2.4.5 Paths in a Confluent RTS

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

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

lemma *confluence-single*:
assumes $\bigwedge t\ u.\ R.coinitial\ t\ u \Longrightarrow t \frown u$
shows $[\![R.arr\ t;\ Arr\ U;\ R.sources\ t = Srcs\ U]\!] \Longrightarrow [t]\ {}^*\!\frown^*\ U$
proof (*induct U arbitrary*: *t*)
show $\bigwedge t.\ [\![R.arr\ t;\ Arr\ [];\ R.sources\ t = Srcs\ []]\!] \Longrightarrow [t]\ {}^*\!\frown^*\ []$
by *simp*
fix *t u U*
assume *ind*: $\bigwedge t.\ [\![R.arr\ t;\ Arr\ U;\ R.sources\ t = Srcs\ U]\!] \Longrightarrow [t]\ {}^*\!\frown^*\ U$
assume *t*: *R.arr t*
assume *uU*: *Arr* (*u # U*)
assume *coinitial*: *R.sources t = Srcs* (*u # U*)
hence *1*: *R.coinitial t u*
using *t uU*
by (*metis Arr.simps*(*2*) *Con-implies-Arr*(*1*) *Con-imp-eq-Srcs Con-initial-left*
Srcs.simps(*2*) *Con-Arr-self R.coinitial-iff*)
show $[t]\ {}^*\!\frown^*\ u\ \#\ U$
proof (*cases U = []*)
show $U = [] \Longrightarrow$ *?thesis*
using *assms t uU coinitial R.coinitial-iff* **by** *fastforce*
assume *U*: $U \neq []$
show *?thesis*
proof −
have *2*: $Arr\ [t \setminus u] \wedge Arr\ U \wedge Srcs\ [t \setminus u] = Srcs\ U$
using *assms 1 t uU U R.arr-resid-iff-con*
apply (*intro conjI*)
apply *simp*
apply (*metis Con-Arr-self Con-implies-Arr*(*2*) *Resid-cons*(*2*))
by (*metis* (*full-types*) *Con-cons*(*2*) *Srcs.simps*(*2*) *Srcs-Resid Trgs.simps*(*2*)
Con-Arr-self Con-imp-eq-Srcs list.simps(*3*) *R.sources-resid*)
have $[t]\ {}^*\!\frown^*\ u\ \#\ U \longleftrightarrow t \frown u \wedge [t \setminus u]\ {}^*\!\frown^*\ U$
using *U Con-rec*(*3*) [*of U t u*] **by** *simp*

```
      also have ... ⟷ True
        using assms t uU U 1 2 ind by force
      finally show ?thesis by blast
    qed
  qed
qed

lemma confluence-ind:
shows ⟦Arr T; Arr U; Srcs T = Srcs U⟧ ⟹ T *⌢* U
proof (induct T arbitrary: U)
  show ⋀U. ⟦Arr []; Arr U; Srcs [] = Srcs U⟧ ⟹ [] *⌢* U
    by simp
  fix t T U
  assume ind: ⋀U. ⟦Arr T; Arr U; Srcs T = Srcs U⟧ ⟹ T *⌢* U
  assume tT: Arr (t # T)
  assume U: Arr U
  assume coinitial: Srcs (t # T) = Srcs U
  show t # T *⌢* U
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      using U tT coinitial confluence-single [of t U] R.confluence by simp
    assume T: T ≠ []
    show ?thesis
    proof −
      have 1: [t] *⌢* U
        using tT U coinitial R.confluence
        by (metis R.arr-def Srcs.simps(2) T Con-Arr-self Con-imp-eq-Srcs
            Con-initial-right Con-rec(4) confluence-single)
      moreover have T *⌢* U *\* [t]
        using 1 tT U T coinitial ind [of U *\* [t]]
        by (metis (full-types) Con-imp-Arr-Resid Arr-iff-Con-self Con-implies-Arr(2)
            Con-imp-eq-Srcs Con-sym R.sources-resid Srcs.simps(2) Srcs-Resid
            Trgs.simps(2) Con-rec(4))
      ultimately show ?thesis
        using Con-cons(1) [of T U t] by fastforce
    qed
  qed
qed

lemma confluence_P:
assumes coinitial T U
shows con T U
  using assms confluence-ind sources-char_P coinitial-def con-char by auto

sublocale confluent-rts Resid
  apply (unfold-locales)
  using confluence_P by simp

lemma is-confluent-rts:
```

shows *confluent-rts Resid*
..

end

2.4.6 Simulations Lift to Paths

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

context *simulation*
begin

interpretation P_A: *paths-in-rts A*
..
interpretation P_B: *paths-in-rts B*
..

lemma *map-Resid-single*:
shows $P_A.con\ T\ [u] \Longrightarrow map\ F\ (P_A.Resid\ T\ [u]) = P_B.Resid\ (map\ F\ T)\ [F\ u]$
apply (*induct T arbitrary*: *u*)
apply *simp*
proof −
fix *t u T*
assume *ind*: $\bigwedge u.\ P_A.con\ T\ [u] \Longrightarrow map\ F\ (P_A.Resid\ T\ [u]) = P_B.Resid\ (map\ F\ T)\ [F\ u]$
assume *1*: $P_A.con\ (t\ \#\ T)\ [u]$
show $map\ F\ (P_A.Resid\ (t\ \#\ T)\ [u]) = P_B.Resid\ (map\ F\ (t\ \#\ T))\ [F\ u]$
proof (*cases* $T = []$)
show $T = [] \Longrightarrow$ *?thesis*
using *1* $P_A.null\text{-}char$ **by** *fastforce*
assume *T*: $T \neq []$
show *?thesis*
using *T 1 ind* $P_A.con\text{-}def\ P_A.null\text{-}char\ P_A.Con\text{-}rec(2)\ P_A.Resid\text{-}rec(2)\ P_B.Con\text{-}rec(2)$
$P_B.Resid\text{-}rec(2)$
apply *simp*
by (*metis A.con-sym Nil-is-map-conv preserves-con preserves-resid*)
qed
qed

lemma *map-Resid*:
shows $P_A.con\ T\ U \Longrightarrow map\ F\ (P_A.Resid\ T\ U) = P_B.Resid\ (map\ F\ T)\ (map\ F\ U)$
apply (*induct U arbitrary*: *T*)
using $P_A.Resid.simps(1)\ P_A.con\text{-}char\ P_A.con\text{-}sym$
apply *blast*
proof −
fix *u U T*
assume *ind*: $\bigwedge T.\ P_A.con\ T\ U \Longrightarrow$
$map\ F\ (P_A.Resid\ T\ U) = P_B.Resid\ (map\ F\ T)\ (map\ F\ U)$
assume *1*: $P_A.con\ T\ (u\ \#\ U)$

```
    show map F (P_A.Resid T (u # U)) = P_B.Resid (map F T) (map F (u # U))
    proof (cases U = [])
      show U = [] ⟹ ?thesis
        using 1 map-Resid-single by force
      assume U: U ≠ []
      have P_B.Resid (map F T) (map F (u # U)) =
           P_B.Resid (P_B.Resid (map F T) [F u]) (map F U)
        using U 1 P_B.Resid-cons(2)
        apply simp
       by (metis P_B.Arr.simps(1) P_B.Con-consI(2) P_B.Con-implies-Arr(1) list.map-disc-iff)
      also have ... = map F (P_A.Resid (P_A.Resid T [u]) U)
        using U 1 ind
        by (metis P_A.Con-initial-right P_A.Resid-cons(2) P_A.con-char map-Resid-single)
      also have ... = map F (P_A.Resid T (u # U))
        using 1 P_A.Resid-cons(2) P_A.con-char U by auto
      finally show ?thesis by simp
    qed
  qed

  lemma preserves-paths:
  shows P_A.Arr T ⟹ P_B.Arr (map F T)
    by (metis P_A.Con-Arr-self P_A.conI_P P_B.Arr-iff-Con-self map-Resid map-is-Nil-conv)

  interpretation Fx: simulation P_A.Resid P_B.Resid ‹λT. if P_A.Arr T then map F T else []›
  proof
    let ?Fx = λT. if P_A.Arr T then map F T else []
    show ⋀T. ¬ P_A.arr T ⟹ ?Fx T = P_B.null
      by (simp add: P_A.arr-char P_B.null-char)
    show ⋀T U. P_A.con T U ⟹ P_B.con (?Fx T) (?Fx U)
    using P_A.Con-implies-Arr(1) P_A.Con-implies-Arr(2) P_A.con-char map-Resid by fastforce
    show ⋀T U. P_A.con T U ⟹ ?Fx (P_A.Resid T U) = P_B.Resid (?Fx T) (?Fx U)
      by (simp add: P_A.Con-imp-Arr-Resid P_A.Con-implies-Arr(1) P_A.Con-implies-Arr(2)
          P_A.con-char map-Resid)
  qed

  lemma lifts-to-paths:
  shows simulation P_A.Resid P_B.Resid (λT. if P_A.Arr T then map F T else [])
    ..

end
```

2.4.7 Normal Sub-RTS's Lift to Paths

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

```
locale paths-in-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* ≡ (*Arr T* ∧ *set T* ⊆ $\mathfrak{N}$)

lemma *Ide-implies-NPath*:
assumes *Ide T*
shows *NPath T*
using *assms*
by (*metis Ball-Collect NPath-def Ide-implies-Arr N.ide-closed set-Ide-subset-ide subsetI*)

lemma *NPath-implies-Arr*:
assumes *NPath T*
shows *Arr T*
using *assms NPath-def* **by** *simp*

lemma *NPath-append*:
assumes *T* ≠ [] **and** *U* ≠ []
shows *NPath* (*T* @ *U*) ⟷ *NPath T* ∧ *NPath U* ∧ *Trgs T* ⊆ *Srcs U*
using *assms NPath-def* **by** *auto*

lemma *NPath-appendI* [*intro*, *simp*]:
assumes *NPath T* **and** *NPath U* **and** *Trgs T* ⊆ *Srcs U*
shows *NPath* (*T* @ *U*)
using *assms NPath-def* **by** *simp*

lemma *NPath-Resid-single-Arr*:
shows ⟦*t* ∈ $\mathfrak{N}$; *Arr U*; *R.sources t* = *Srcs U*⟧ ⟹ *NPath* (*Resid* [*t*] *U*)
proof (*induct U arbitrary*: *t*)
show ⋀*t*. ⟦*t* ∈ $\mathfrak{N}$; *Arr* []; *R.sources t* = *Srcs* []⟧ ⟹ *NPath* (*Resid* [*t*] [])
by *simp*
fix *t u U*
assume *ind*: ⋀*t*. ⟦*t* ∈ $\mathfrak{N}$; *Arr U*; *R.sources t* = *Srcs U*⟧ ⟹ *NPath* (*Resid* [*t*] *U*)
assume *t*: *t* ∈ $\mathfrak{N}$
assume *uU*: *Arr* (*u* # *U*)
assume *src*: *R.sources t* = *Srcs* (*u* # *U*)
show *NPath* (*Resid* [*t*] (*u* # *U*))
proof (*cases U* = [])
show *U* = [] ⟹ *?thesis*
using *NPath-def t src*
apply *simp*
by (*metis Arr.simps*(*2*) *R.arr-resid-iff-con R.coinitialI N.forward-stable N.elements-are-arr uU*)
assume *U*: *U* ≠ []
show *?thesis*
proof −

```
      have NPath (Resid [t] (u # U)) ⟷ NPath (Resid [t \ u] U)
        using t U uU src
       by (metis Arr.simps(2) Con-implies-Arr(1) Resid-rec(3) Con-rec(3) R.arr-resid-iff-con)
      also have ... ⟷ True
      proof −
        have t \ u ∈ ℜ
          using t U uU src N.forward-stable [of t u]
          by (metis Con-Arr-self Con-imp-eq-Srcs Con-initial-left
                    Srcs.simps(2) inf.idem Arr-has-Src R.coinitial-def)
        moreover have Arr U
          using U uU
          by (metis Arr.simps(3) neq-Nil-conv)
        moreover have R.sources (t \ u) = Srcs U
          using t uU src
          by (metis Con-Arr-self Srcs.simps(2) U calculation(1) Con-imp-eq-Srcs
                    Con-rec(4) N.elements-are-arr R.sources-resid R.arr-resid-iff-con)
        ultimately show ?thesis
          using ind [of t \ u] by simp
      qed
      finally show ?thesis by blast
    qed
  qed
qed

lemma NPath-Resid-Arr-single:
shows ⟦ NPath T; R.arr u; Srcs T = R.sources u ⟧ ⟹ NPath (Resid T [u])
proof (induct T arbitrary: u)
  show ⋀u. ⟦NPath []; R.arr u; Srcs [] = R.sources u⟧ ⟹ NPath (Resid [] [u])
    by simp
  fix t u T
  assume ind: ⋀u. ⟦NPath T; R.arr u; Srcs T = R.sources u⟧ ⟹ NPath (Resid T [u])
  assume tT: NPath (t # T)
  assume u: R.arr u
  assume src: Srcs (t # T) = R.sources u
  show NPath (Resid (t # T) [u])
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      using tT u src NPath-def
     by (metis Arr.simps(2) NPath-Resid-single-Arr Srcs.simps(2) list.set-intros(1) subsetD)
    assume T: T ≠ []
    have R.coinitial u t
      by (metis R.coinitialI Srcs.simps(3) T list.exhaust-sel src u)
    hence con: t ⌢ u
      using tT T u src R.con-sym NPath-def
      by (metis N.forward-stable N.elements-are-arr R.not-arr-null
          list.set-intros(1) R.conI subsetD)
    have 1: NPath (Resid (t # T) [u]) ⟷ NPath ((t \ u) # Resid T [u \ t])
    proof −
      have t # T *⌢* [u]
```

```
      proof −
        have 2: [t] *⌢* [u]
          by (simp add: Con-rec(1) con)
        moreover have T *⌢* Resid [u] [t]
        proof −
          have NPath T
            using tT T NPath-def
            by (metis NPath-append append-Cons append-Nil)
          moreover have 3: R.arr (u \ t)
            using con by (meson R.arr-resid-iff-con R.con-sym)
          moreover have Srcs T = R.sources (u \ t)
            using tT T u src con
            by (metis 3 Arr-iff-Con-self Con-cons(2) Con-imp-eq-Srcs
                R.sources-resid Srcs-Resid Trgs.simps(2) NPath-implies-Arr list.discI
                R.arr-resid-iff-con)
          ultimately show ?thesis
            using 2 ind [of u \ t] NPath-def by auto
        qed
        ultimately show ?thesis
          using tT T u src Con-cons(1) [of T [u] t] by simp
      qed
      thus ?thesis
        using tT T u src Resid-cons(1) [of T t [u]] Resid-rec(2) by presburger
    qed
    also have ... ⟷ True
    proof −
      have 2: t \ u ∈ ℜ ∧ R.arr (u \ t)
        using tT u src con NPath-def
        by (meson R.arr-resid-iff-con R.con-sym N.forward-stable ‹R.coinitial u t›
            list.set-intros(1) subsetD)
      moreover have 3: NPath (T *\* [u \ t])
        using tT ind [of u \ t] NPath-def
        by (metis Con-Arr-self Con-imp-eq-Srcs Con-rec(4) R.arr-resid-iff-con
            R.sources-resid Srcs.simps(2) T calculation insert-subset list.exhaust
            list.simps(15) Arr.simps(3))
      moreover have R.targets (t \ u) ⊆ Srcs (Resid T [u \ t])
        using tT T u src NPath-def
        by (metis 3 Arr.simps(1) R.targets-resid-sym Srcs-Resid-Arr-single con subset-refl)
      ultimately show ?thesis
        using NPath-def
        by (metis Arr-consI_P N.elements-are-arr insert-subset list.simps(15))
    qed
    finally show ?thesis by blast
  qed
qed

lemma NPath-Resid [simp]:
shows ⟦NPath T; Arr U; Srcs T = Srcs U⟧ ⟹ NPath (T *\* U)
proof (induct T arbitrary: U)
```

show $\bigwedge U.\ [\![NPath\ []; Arr\ U; Srcs\ [] = Srcs\ U]\!] \Longrightarrow NPath\ ([]\ {}^*\backslash^*\ U)$
by *simp*
fix *t T U*
assume *ind*: $\bigwedge U.\ [\![NPath\ T; Arr\ U; Srcs\ T = Srcs\ U]\!] \Longrightarrow NPath\ (T\ {}^*\backslash^*\ U)$
assume *tT*: *NPath* (*t* # *T*)
assume *U*: *Arr U*
assume *Coinitial*: *Srcs* (*t* # *T*) = *Srcs U*
show *NPath* ((*t* # *T*) $^*\backslash^*$ *U*)
proof (*cases T* = [])
show *T* = [] $\Longrightarrow$ *?thesis*
using *tT U Coinitial NPath-Resid-single-Arr* [*of t U*] *NPath-def* **by** *force*
assume *T*: *T* ≠ []
have *0*: *NPath* ((*t* # *T*) $^*\backslash^*$ *U*) $\longleftrightarrow$ *NPath* ([*t*] $^*\backslash^*$ *U* @ *T* $^*\backslash^*$ (*U* $^*\backslash^*$ [*t*]))
proof −
have *U* ≠ []
using *U* **by** *auto*
moreover have (*t* # *T*) $^*\frown^*$ *U*
proof −
have *t* ∈ $\mathfrak{N}$
using *tT NPath-def* **by** *auto*
moreover have *R.sources t* = *Srcs U*
using *Coinitial*
by (*metis Srcs.elims U list.sel*(*1*) *Arr-has-Src*)
ultimately have *1*: [*t*] $^*\frown^*$ *U*
using *U NPath-Resid-single-Arr* [*of t U*] *NPath-def* **by** *auto*
moreover have *T* $^*\frown^*$ (*U* $^*\backslash^*$ [*t*])
proof −
have *Srcs T* = *Srcs* (*U* $^*\backslash^*$ [*t*])
using *tT U Coinitial 1*
by (*metis Con-Arr-self Con-cons*(*2*) *Con-imp-eq-Srcs Con-sym Srcs-Resid-Arr-single*
T list.discI NPath-implies-Arr)
hence *NPath* (*T* $^*\backslash^*$ (*U* $^*\backslash^*$ [*t*]))
using *tT U Coinitial 1 Con-sym ind* [*of Resid U* [*t*]] *NPath-def*
by (*metis Con-imp-Arr-Resid Srcs.elims T insert-subset list.simps*(*15*)
Arr.simps(*3*))
thus *?thesis*
using *NPath-def* **by** *auto*
qed
ultimately show *?thesis*
using *Con-cons*(*1*) [*of T U t*] **by** *fastforce*
qed
ultimately show *?thesis*
using *tT U T Coinitial Resid-cons*(*1*) **by** *auto*
qed
also have ... $\longleftrightarrow$ *True*
proof (*intro iffI*, *simp-all*)
have *1*: *NPath* ([*t*] $^*\backslash^*$ *U*)
by (*metis Coinitial NPath-Resid-single-Arr Srcs-simp*$_P$ *U insert-subset*
list.sel(*1*) *list.simps*(*15*) *NPath-def tT*)

```
    moreover have 2: NPath (T *\* (U *\* [t]))
      by (metis 0 Arr.simps(1) Con-cons(1) Con-imp-eq-Srcs Con-implies-Arr(1−2)
          NPath-def T append-Nil2 calculation ind insert-subset list.simps(15) tT)
    moreover have Trgs ([t] *\* U) ⊆ Srcs (T *\* (U *\* [t]))
      by (metis Arr.simps(1) NPath-def Srcs-Resid Trgs-Resid-sym calculation(2)
          dual-order.refl)
    ultimately show NPath ([t] *\* U @ T *\* (U *\* [t]))
      using NPath-append [of T *\* (U *\* [t]) [t] *\* U] by fastforce
  qed
  finally show ?thesis by blast
 qed
qed

lemma Backward-stable-single:
shows ⟦NPath U; NPath ([t] *\* U)⟧ ⟹ NPath [t]
proof (induct U arbitrary: t)
 show ⋀t. ⟦NPath []; NPath ([t] *\* [])⟧ ⟹ NPath [t]
   using NPath-def by simp
 fix t u U
 assume ind: ⋀t. ⟦NPath U; NPath ([t] *\* U)⟧ ⟹ NPath [t]
 assume uU: NPath (u # U)
 assume resid: NPath ([t] *\* (u # U))
 show NPath [t]
   using uU ind NPath-def
   by (metis Arr.simps(1) Arr.simps(2) Con-implies-Arr(2) N.backward-stable
       N.elements-are-arr Resid-rec(1) Resid-rec(3) insert-subset list.simps(15) resid)
qed

lemma Backward-stable:
shows ⟦NPath U; NPath (T *\* U)⟧ ⟹ NPath T
proof (induct T arbitrary: U)
 show ⋀U. ⟦NPath U; NPath ([] *\* U)⟧ ⟹ NPath []
   by simp
 fix t T U
 assume ind: ⋀U. ⟦NPath U; NPath (T *\* U)⟧ ⟹ NPath T
 assume U: NPath U
 assume resid: NPath ((t # T) *\* U)
 show NPath (t # T)
 proof (cases T = [])
   show T = [] ⟹ ?thesis
     using U resid Backward-stable-single by blast
   assume T: T ≠ []
   have 1: NPath ([t] *\* U) ∧ NPath (T *\* (U *\* [t]))
     using T U NPath-append resid NPath-def
     by (metis Arr.simps(1) Con-cons(1) Resid-cons(1))
   have 2: t ∈ ℜ
     using 1 U Backward-stable-single NPath-def by simp
   moreover have NPath T
     using 1 U resid ind
```

by (*metis 2 Arr.simps(2) Con-imp-eq-Srcs NPath-Resid N.elements-are-arr*)
moreover have *R.targets t* ⊆ *Srcs T*
using *resid 1 Con-imp-eq-Srcs Con-sym Srcs-Resid-Arr-single NPath-def*
by (*metis Arr.simps(1) dual-order.eq-iff*)
ultimately show *?thesis*
using *NPath-def*
by (*simp add*: *N.elements-are-arr*)
qed
qed

sublocale *normal-sub-rts Resid* ‹*Collect NPath*›
using *Ide-implies-NPath NPath-implies-Arr arr-char ide-char coinitial-def*
sources-char$_P$ *append-is-composite-of*
apply *unfold-locales*
apply *auto*
using *Backward-stable*
by *metis+*

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

lemma *Resid-NPath-preserves-reflects-Con*:
assumes *NPath U* **and** *Srcs T* = *Srcs U*
shows *T* $*\backslash^*$ *U* $*\frown^*$ *T'* $*\backslash^*$ *U* ⟷ *T* $*\frown^*$ *T'*
using *assms NPath-def NPath-Resid con-char con-imp-coinitial resid-along-elem-preserves-con*
Con-implies-Arr(2) Con-sym Cube(1)
by (*metis Arr.simps(1) mem-Collect-eq*)

notation *Cong*$_0$ (**infix** ‹≈*$_0$› *50*)
notation *Cong* (**infix** ‹≈*› *50*)

lemma *Cong*$_0$*-cancel-left*$_{CS}$:
assumes *T* @ *U* ≈*$_0$ *T* @ *U'* **and** *T* ≠ [] **and** *U* ≠ [] **and** *U'* ≠ []
shows *U* ≈*$_0$ *U'*
using *assms Cong*$_0$*-cancel-left* [*of T U T* @ *U U' T* @ *U'*] *Cong*$_0$*-reflexive*
append-is-composite-of
by (*metis Cong*$_0$*-implies-Cong Cong-imp-arr(1) arr-append-imp-seq*)

lemma *Srcs-respects-Cong*:
assumes *T* ≈* *T'* **and** *a* ∈ *Srcs T* **and** *a'* ∈ *Srcs T'*
shows [*a*] ≈* [*a'*]
proof −
obtain *U U'* **where** *UU'*: *NPath U* ∧ *NPath U'* ∧ *T* $*\backslash^*$ *U* ≈*$_0$ *T'* $*\backslash^*$ *U'*
using *assms(1)* **by** *blast*
show *?thesis*
proof
show *U* ∈ *Collect NPath*

```
    using UU' by simp
  show U' ∈ Collect NPath
    using UU' by simp
  show [a] *\* U ≈*₀ [a'] *\* U'
  proof −
    have NPath ([a] *\* U) ∧ NPath ([a'] *\* U')
      by (metis Arr.simps(1) Con-imp-eq-Srcs Con-implies-Arr(1) Con-single-ide-ind
          NPath-implies-Arr N.ide-closed R.in-sourcesE Srcs.simps(2) Srcs-simp_P
          UU' assms(2−3) elements-are-arr not-arr-null null-char NPath-Resid-single-Arr)
    thus ?thesis
      using UU'
      by (metis Con-imp-eq-Srcs Cong₀-imp-con NPath-Resid Srcs-Resid
          con-char NPath-implies-Arr mem-Collect-eq arr-resid-iff-con con-implies-arr(2))
  qed
 qed
qed

lemma Trgs-respects-Cong:
assumes T ≈* T' and b ∈ Trgs T and b' ∈ Trgs T'
shows [b] ≈* [b']
proof −
  have [b] ∈ targets T ∧ [b'] ∈ targets T'
  proof −
    have 1: Ide [b] ∧ Ide [b']
      using assms
      by (metis Ball-Collect Trgs-are-ide Ide.simps(2))
    moreover have Srcs [b] = Trgs T
      using assms
      by (metis 1 Con-imp-Arr-Resid Con-imp-eq-Srcs Cong-imp-arr(1) Ide.simps(2)
          Srcs-Resid Con-single-ide-ind con-char arrE)
    moreover have Srcs [b'] = Trgs T'
      using assms
      by (metis Con-imp-Arr-Resid Con-imp-eq-Srcs Cong-imp-arr(2) Ide.simps(2)
          Srcs-Resid 1 Con-single-ide-ind con-char arrE)
    ultimately show ?thesis
      unfolding targets-char_P
      using assms Cong-imp-arr(2) arr-char by blast
  qed
  thus ?thesis
    using assms targets-char in-targets-respects-Cong [of T T' [b] [b']] by simp
qed

lemma Cong₀-append-resid-NPath:
assumes NPath (T *\* U)
shows Cong₀ (T @ (U *\* T)) U
proof (intro conjI)
  show 0: (T @ U *\* T) *\* U ∈ Collect NPath
  proof −
    have 1: (T @ U *\* T) *\* U = T *\* U @ (U *\* T) *\* (U *\* T)
```

```
      by (metis Arr.simps(1) NPath-implies-Arr assms Con-append(1) Con-implies-Arr(2)
          Con-sym Resid-append(1) con-imp-arr-resid null-char)
    moreover have NPath ...
      using assms
      by (metis 1 Arr-append-iff_P NPath-append NPath-implies-Arr Ide-implies-NPath
          Nil-is-append-conv Resid-Arr-self arr-char con-char arr-resid-iff-con
          self-append-conv)
    ultimately show ?thesis by simp
  qed
  show U *\* (T @ U *\* T) ∈ Collect NPath
    using assms 0
    by (metis Arr.simps(1) Con-implies-Arr(2) Cong_0-reflexive Resid-append(2)
        append.right-neutral arr-char Con-sym)
qed

end

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

  sublocale paths-in-rts-with-normal resid 𝔑 ..

  notation Cong_0  (infix ‹≈*_0› 50)
  notation Cong  (infix ‹≈*› 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 ⟹ ∃u. u ∈ 𝔑 ∧ R.sources u = Srcs U ∧ R.targets u = Trgs U ∧
                       (∀t. con [t] U ⟶ [t] *\* U ≈*_0 [t \ u])
  proof (induct U)
    show NPath [] ⟹ ∃u. u ∈ 𝔑 ∧ R.sources u = Srcs [] ∧ R.targets u = Trgs [] ∧
                         (∀t. con [t] [] ⟶ [t] *\* [] ≈*_0 [t \ u])
      using NPath-def by auto
    fix v U
    assume ind: NPath U ⟹ ∃u. u ∈ 𝔑 ∧ R.sources u = Srcs U ∧ R.targets u = Trgs U ∧
                              (∀t. con [t] U ⟶ [t] *\* U ≈*_0 [t \ u])
    assume vU: NPath (v # U)
    show ∃vU. vU ∈ 𝔑 ∧ R.sources vU = Srcs (v # U) ∧ R.targets vU = Trgs (v # U) ∧
              (∀t. con [t] (v # U) ⟶ [t] *\* (v # U) ≈*_0 [t \ vU])
    proof (cases U = [])
      show U = [] ⟹ ∃vU. vU ∈ 𝔑 ∧ R.sources vU = Srcs (v # U) ∧
                         R.targets vU = Trgs (v # U) ∧
                         (∀t. con [t] (v # U) ⟶ [t] *\* (v # U) ≈*_0 [t \ vU])
        using vU Resid-rec(1) con-char
        by (metis Cong_0-reflexive NPath-def Srcs.simps(2) Trgs.simps(2) arr-resid-iff-con
```

```
      insert-subset list.simps(15))
assume U ≠ []
hence U: NPath U
  using vU by (metis NPath-append append-Cons append-Nil)
obtain u where u: u ∈ ℜ ∧ R.sources u = Srcs U ∧ R.targets u = Trgs U ∧
                  (∀ t. con [t] U ⟶ [t] *\* U ≈*₀ [t \ u])
  using U ind by blast
have seq: R.seq v u
proof
  show R.arr u
    by (simp add: N.elements-are-arr u)
  show R.trg v ∼ R.src u
  proof −
    have R.targets v = R.sources u
      by (metis (full-types) NPath-implies-Arr R.sources-resid Srcs.simps(2)
          ‹U ≠ []› Con-Arr-self Con-imp-eq-Srcs Con-initial-right Con-rec(2)
          u vU)
    thus ?thesis
      using R.seqI(2) ‹R.arr u› by blast
  qed
qed
obtain vu where vu: R.composite-of v u vu
  using N.composite-closed-right seq u by presburger
have vu ∈ ℜ ∧ R.sources vu = Srcs (v # U) ∧ R.targets vu = Trgs (v # U) ∧
     (∀ t. con [t] (v # U) ⟶ [t] *\* (v # U) ≈*₀ [t \ vu])
proof (intro conjI allI)
  show vu ∈ ℜ
    by (meson NPath-def N.composite-closed list.set-intros(1) subsetD u vU vu)
  show R.sources vu = Srcs (v # U)
    by (metis Con-imp-eq-Srcs Con-initial-right NPath-implies-Arr
          R.sources-composite-of Srcs.simps(2) Arr-iff-Con-self vU vu)
  show R.targets vu = Trgs (v # U)
    by (metis R.targets-composite-of Trgs.simps(3) ‹U ≠ []› list.exhaust-sel u vu)
  fix t
  show con [t] (v # U) ⟶ [t] *\* (v # U) ≈*₀ [t \ vu]
  proof (intro impI)
    assume t: con [t] (v # U)
    have 1: [t] *\* (v # U) = [t \ v] *\* U
      using t Resid-rec(3) ‹U ≠ []› con-char by force
    also have ... ≈*₀ [(t \ v) \ u]
      using 1 t u by force
    also have [(t \ v) \ u] ≈*₀ [t \ vu]
    proof −
      have (t \ v) \ u ∼ t \ vu
        using vu R.resid-composite-of
      by (metis (no-types, lifting) N.Cong₀-composite-of-arr-normal N.Cong₀-subst-right(1)
          ‹U ≠ []› Con-rec(3) con-char R.con-sym t u)
      thus ?thesis
        using Ide.simps(2) R.prfx-implies-con Resid.simps(3) ide-char ide-closed
```

```
      by presburger
    qed
    finally show [t] *\* (v # U) ≈*₀ [t \ vu] by blast
   qed
  qed
  thus ?thesis by blast
 qed
qed
```

Coherence for single transitions extends inductively to paths.

```
lemma Coherent-single:
assumes R.arr t and NPath U and NPath U'
and R.sources t = Srcs U and Srcs U = Srcs U' and Trgs U = Trgs U'
shows [t] *\* U ≈*₀ [t] *\* U'
proof −
 have 1: con [t] U ∧ con [t] U'
  using assms
  by (metis Arr.simps(1−2) Arr-iff-Con-self Resid-NPath-preserves-reflects-Con
      Srcs.simps(2) con-char)
 obtain u where u: u ∈ 𝔑 ∧ R.sources u = Srcs U ∧ R.targets u = Trgs U ∧
                   (∀ t. con [t] U ⟶ [t] *\* U ≈*₀ [t \ u])
  using assms NPath-folding by metis
 obtain u' where u': u' ∈ 𝔑 ∧ R.sources u' = Srcs U' ∧ R.targets u' = Trgs U' ∧
                     (∀ t. con [t] U' ⟶ [t] *\* U' ≈*₀ [t \ u'])
  using assms NPath-folding by metis
 have [t] *\* U ≈*₀ [t \ u]
  using u 1 by blast
 also have [t \ u] ≈*₀ [t \ u']
  using assms(1,4−6) N.Cong₀-imp-con N.coherent u u' NPath-def by simp
 also have [t \ u'] ≈*₀ [t] *\* U'
  using u' 1 by simp
 finally show ?thesis by simp
qed

lemma Coherent:
shows ⟦ Arr T; NPath U; NPath U'; Srcs T = Srcs U;
        Srcs U = Srcs U'; Trgs U = Trgs U' ⟧
          ⟹ T *\* U ≈*₀ T *\* U'
proof (induct T arbitrary: U U')
 show ⋀U U'. ⟦ Arr []; NPath U; NPath U'; Srcs [] = Srcs U;
               Srcs U = Srcs U'; Trgs U = Trgs U' ⟧
                 ⟹ [] *\* U ≈*₀ [] *\* U'
  by (simp add: arr-char)
 fix t T U U'
 assume tT: Arr (t # T) and U: NPath U and U': NPath U'
 and Srcs1: Srcs (t # T) = Srcs U and Srcs2: Srcs U = Srcs U'
 and Trgs: Trgs U = Trgs U'
 and ind: ⋀U U'. ⟦ Arr T; NPath U; NPath U'; Srcs T = Srcs U;
                   Srcs U = Srcs U'; Trgs U = Trgs U' ⟧
```

$\Longrightarrow T \mathbin{*\backslash^*} U \approx^*_0 T \mathbin{*\backslash^*} U'$
have *t*: *R.arr t*
using *tT* **by** (*metis Arr.simps(2) Con-Arr-self Con-rec(4) R.arrI*)
show $(t \# T) \mathbin{*\backslash^*} U \approx^*_0 (t \# T) \mathbin{*\backslash^*} U'$
proof (*cases* $T = []$)
show $T = [] \Longrightarrow$ *?thesis*
by (*metis Srcs.simps(2) Srcs1 Srcs2 Trgs U U' Coherent-single Arr.simps(2) tT*)
assume *T*: $T \neq []$
let $?t = [t] \mathbin{*\backslash^*} U$ **and** $?t' = [t] \mathbin{*\backslash^*} U'$
let $?T = T \mathbin{*\backslash^*} (U \mathbin{*\backslash^*} [t])$
let $?T' = T \mathbin{*\backslash^*} (U' \mathbin{*\backslash^*} [t])$
have *0*: $(t \# T) \mathbin{*\backslash^*} U = ?t \mathbin{@} ?T \wedge (t \# T) \mathbin{*\backslash^*} U' = ?t' \mathbin{@} ?T'$
using *tT U U' Srcs1 Srcs2*
by (*metis Arr-has-Src Arr-iff-Con-self Resid-cons(1) Srcs.simps(1) Resid-NPath-preserves-reflects-Con*)
have *1*: $?t \approx^*_0 ?t'$
by (*metis Srcs1 Srcs2 Srcs-simp$_P$ Trgs U U' list.sel(1) Coherent-single t tT*)
have *A*: $?T \mathbin{*\backslash^*} (?t' \mathbin{*\backslash^*} ?t) = T \mathbin{*\backslash^*} ((U \mathbin{*\backslash^*} [t]) \mathbin{@} (?t' \mathbin{*\backslash^*} ?t))$
using *1 Arr.simps(1) Con-append(2) Con-sym Resid-append(2) Con-implies-Arr(1) NPath-def*
by (*metis arr-char elements-are-arr*)
have *B*: $?T' \mathbin{*\backslash^*} (?t \mathbin{*\backslash^*} ?t') = T \mathbin{*\backslash^*} ((U' \mathbin{*\backslash^*} [t]) \mathbin{@} (?t \mathbin{*\backslash^*} ?t'))$
by (*metis 1 Con-appendI(2) Con-sym Resid.simps(1) Resid-append(2) elements-are-arr not-arr-null null-char*)
have *E*: $?T \mathbin{*\backslash^*} (?t' \mathbin{*\backslash^*} ?t) \approx^*_0 ?T' \mathbin{*\backslash^*} (?t \mathbin{*\backslash^*} ?t')$
proof −
have *Arr T*
using *Arr.elims(1) T tT* **by** *blast*
moreover have *NPath* $(U \mathbin{*\backslash^*} [t] \mathbin{@} ([t] \mathbin{*\backslash^*} U') \mathbin{*\backslash^*} ([t] \mathbin{*\backslash^*} U))$
using *1 U t tT Srcs1 Srcs-simp$_P$*
apply (*intro NPath-appendI*)
apply *auto*
by (*metis Arr.simps(1) NPath-def Srcs-Resid Trgs-Resid-sym*)
moreover have *NPath* $(U' \mathbin{*\backslash^*} [t] \mathbin{@} ([t] \mathbin{*\backslash^*} U) \mathbin{*\backslash^*} ([t] \mathbin{*\backslash^*} U'))$
using *t U' 1 Con-imp-eq-Srcs Trgs-Resid-sym*
apply (*intro NPath-appendI*)
apply *auto*
apply (*metis Arr.simps(2) NPath-Resid Resid.simps(1)*)
by (*metis Arr.simps(1) NPath-def Srcs-Resid*)
moreover have *Srcs T* = *Srcs* $(U \mathbin{*\backslash^*} [t] \mathbin{@} ([t] \mathbin{*\backslash^*} U') \mathbin{*\backslash^*} ([t] \mathbin{*\backslash^*} U))$
using *A B*
by (*metis* (*full-types*) *0 1 Arr-has-Src Con-cons(1) Con-implies-Arr(1) Srcs.simps(1) Srcs-append T elements-are-arr not-arr-null null-char Con-imp-eq-Srcs*)
moreover have *Srcs* $(U \mathbin{*\backslash^*} [t] \mathbin{@} ([t] \mathbin{*\backslash^*} U') \mathbin{*\backslash^*} ([t] \mathbin{*\backslash^*} U)) =$
Srcs $(U' \mathbin{*\backslash^*} [t] \mathbin{@} ([t] \mathbin{*\backslash^*} U) \mathbin{*\backslash^*} ([t] \mathbin{*\backslash^*} U'))$
by (*metis 1 Con-implies-Arr(2) Con-sym Cong$_0$-imp-con Srcs-Resid Srcs-append arr-char con-char arr-resid-iff-con*)
moreover have *Trgs* $(U \mathbin{*\backslash^*} [t] \mathbin{@} ([t] \mathbin{*\backslash^*} U') \mathbin{*\backslash^*} ([t] \mathbin{*\backslash^*} U)) =$

```
            Trgs (U' *\* [t] @ ([t] *\* U) *\* ([t] *\* U'))
    using 1 Cong₀-imp-con con-char by force
  ultimately show ?thesis
    using A B ind [of (U *\* [t]) @ (?t' *\* ?t) (U' *\* [t]) @ (?t *\* ?t')]
    by simp
qed
have C: NPath ((?T *\* (?t' *\* ?t)) *\* (?T' *\* (?t *\* ?t')))
  using E by blast
have D: NPath ((?T' *\* (?t *\* ?t')) *\* (?T *\* (?t' *\* ?t)))
  using E by blast
show ?thesis
proof
  have 2: ((t # T) *\* U) *\* ((t # T) *\* U') =
        ((?t *\* ?t') *\* ?T') @ ((?T *\* (?t' *\* ?t)) *\* (?T' *\* (?t *\* ?t')))
  proof −
    have ((t # T) *\* U) *\* ((t # T) *\* U') = (?t @ ?T) *\* (?t' @ ?T')
      using 0 by fastforce
    also have ... = ((?t @ ?T) *\* ?t') *\* ?T'
      using tT T U U' Srcs1 Srcs2 0
      by (metis Con-appendI(2) Con-cons(1) Con-sym Resid.simps(1) Resid-append(2))
    also have ... = ((?t *\* ?t') @ (?T *\* (?t' *\* ?t))) *\* ?T'
      by (metis (no-types, lifting) Arr.simps(1) Con-appendI(1) Con-implies-Arr(1)
          D NPath-def Resid-append(1) null-is-zero(2))
    also have ... = ((?t *\* ?t') *\* ?T') @
                    ((?T *\* (?t' *\* ?t)) *\* (?T' *\* (?t *\* ?t')))
    proof −
      have ?t *\* ?t' @ ?T *\* (?t' *\* ?t) *⌢* ?T'
        using C D E Con-sym
        by (metis Con-append(2) Cong₀-imp-con con-char arr-resid-iff-con
            con-implies-arr(2))
      thus ?thesis
        using Resid-append(1)
        by (metis Con-sym append.right-neutral Resid.simps(1))
    qed
    finally show ?thesis by simp
  qed
  moreover have 3: NPath ...
  proof −
    have NPath ((?t *\* ?t') *\* ?T')
      using 0 1 E
      by (metis Con-imp-Arr-Resid Con-imp-eq-Srcs NPath-Resid Resid.simps(1)
          ex-un-null mem-Collect-eq)
    moreover have Trgs ((?t *\* ?t') *\* ?T') =
                  Srcs ((?T *\* (?t' *\* ?t)) *\* (?T' *\* (?t *\* ?t')))
      using C
      by (metis NPath-implies-Arr Srcs.simps(1) Srcs-Resid
          Trgs-Resid-sym Arr-has-Src)
    ultimately show ?thesis
      using C by blast
```

```
      qed
      ultimately show ((t # T) *\* U) *\* ((t # T) *\* U') ∈ Collect NPath
        by simp

      have 4: ((t # T) *\* U') *\* ((t # T) *\* U) =
            ((?t' *\* ?t) *\* ?T) @ ((?T' *\* (?t *\* ?t')) *\* (?T *\* (?t' *\* ?t)))
        by (metis 0 2 3 Arr.simps(1) Con-implies-Arr(1) Con-sym D NPath-def Resid-append2)
      moreover have NPath ...
      proof −
        have NPath ((?t' *\* ?t) *\* ?T)
          by (metis 1 CollectD Cong0-imp-con E con-imp-coinitial forward-stable
              arr-resid-iff-con con-implies-arr(2))
        moreover have NPath ((?T' *\* (?t *\* ?t')) *\* (?T *\* (?t' *\* ?t)))
          using U U' 1 D ind Coherent-single [of t U' U] by blast
        moreover have Trgs ((?t' *\* ?t) *\* ?T) =
                      Srcs ((?T' *\* (?t *\* ?t')) *\* (?T *\* (?t' *\* ?t)))
          by (metis Arr.simps(1) NPath-def Srcs-Resid Trgs-Resid-sym calculation(2))
        ultimately show ?thesis by blast
      qed
      ultimately show ((t # T) *\* U') *\* ((t # T) *\* U) ∈ Collect NPath
        by simp
    qed
  qed
qed

sublocale rts-with-composites Resid
  using is-rts-with-composites by simp

sublocale coherent-normal-sub-rts Resid ‹Collect NPath›
proof
  fix T U U'
  assume T: arr T and U: U ∈ Collect NPath and U': U' ∈ Collect NPath
  assume sources-UU': sources U = sources U' and targets-UU': targets U = targets U'
  and TU: sources T = sources U
  have Srcs T = Srcs U
    using TU sources-char_P T arr-iff-has-source by auto
  moreover have Srcs U = Srcs U'
    by (metis Con-imp-eq-Srcs T TU con-char con-imp-coinitial-ax con-sym in-sourcesE
        in-sourcesI arr-def sources-UU')
  moreover have Trgs U = Trgs U'
    using U U' targets-UU' targets-char
    by (metis (full-types) arr-iff-has-target composable-def composable-iff-seq
        composite-of-arr-target elements-are-arr equals0I seq-char)
  ultimately show T *\* U ≈*_0 T *\* U'
    using T U U' Coherent [of T U U'] arr-char by blast
qed

theorem coherent-normal-extends-to-paths:
shows coherent-normal-sub-rts Resid (Collect NPath)
```

..

lemma *Cong$_0$-append-Arr-NPath*:
assumes $T \neq []$ **and** *Arr* $(T \mathbin{@} U)$ **and** *NPath U*
shows *Cong$_0$* $(T \mathbin{@} U)$ T
using *assms*
by (*metis Arr.simps(1) Arr-appendE$_P$ NPath-implies-Arr append-is-composite-of arrI$_P$ arr-append-imp-seq composite-of-arr-normal mem-Collect-eq*)

lemma *Cong-append-NPath-Arr*:
assumes $T \neq []$ **and** *Arr* $(U \mathbin{@} T)$ **and** *NPath U*
shows $U \mathbin{@} T \approx^* T$
using *assms*
by (*metis* (*full-types*) *Arr.simps(1) Con-Arr-self Con-append(2) Con-implies-Arr(2) Con-imp-eq-Srcs composite-of-normal-arr Srcs-Resid append-is-composite-of arr-char NPath-implies-Arr mem-Collect-eq seq-char*)

Permutation Congruence

Here we show that $^*\sim^*$ 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 $T \mathbin{@} [b]$ 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
| $[\![$*PCong* T U; *PCong* U $V]\!] \Longrightarrow$ *PCong* T V
| $[\![$*seq* T U; *PCong* T T'; *PCong* U $U']\!] \Longrightarrow$ *PCong* $(T \mathbin{@} U)$ $(T' \mathbin{@} U')$
| $[\![$*seq* T $[b]$; *R.ide* $b]\!] \Longrightarrow$ *PCong* $(T \mathbin{@} [b])$ T
| $t \frown u \Longrightarrow$ *PCong* $[t,\ u \setminus t]$ $[u,\ t \setminus u]$

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

lemma *PCong-append-Ide*:
shows $[\![$*seq* T B; *Ide* $B]\!] \Longrightarrow$ *PCong* $(T \mathbin{@} B)$ T
proof (*induct B*)
show $[\![$*seq* T $[]$; *Ide* $[]]\!] \Longrightarrow$ *PCong* $(T \mathbin{@} [])$ T
by *auto*
fix b B T
assume *ind*: $[\![$*seq* T B; *Ide* $B]\!] \Longrightarrow$ *PCong* $(T \mathbin{@} B)$ T
assume *seq*: *seq* T $(b \mathbin{\#} B)$
assume *Ide*: *Ide* $(b \mathbin{\#} B)$
have $T \mathbin{@} (b \mathbin{\#} B) = (T \mathbin{@} [b]) \mathbin{@} B$
by *simp*
also have *PCong* ... $(T \mathbin{@} B)$
apply (*cases* $B = []$)
using *Ide PCong.intros(5) seq* **apply** *force*
using *seq Ide PCong.intros(4)* [*of* $T \mathbin{@} [b]$ B T B]
by (*metis Arr.simps(1) Ide-imp-Ide-hd PCong.intros(1) PCong.intros(5) append-is-Nil-conv arr-append arr-append-imp-seq arr-char calculation*

```
      list.distinct(1) list.sel(1) seq-char)
  also have PCong (T @ B) T
  proof (cases B = [])
    show B = [] ⟹ ?thesis
      using PCong.intros(1) seq seq-char by force
    assume B: B ≠ []
    have seq T B
      using B seq Ide
      by (metis Con-imp-eq-Srcs Ide-imp-Ide-hd Trgs-append ‹T @ b # B = (T @ [b]) @ B›
          append-is-Nil-conv arr-append arr-append-imp-seq arr-char cong-cons-ideI(2)
          list.distinct(1) list.sel(1) not-arr-null null-char seq-char ide-implies-arr)
    thus ?thesis
      using seq Ide ind
      by (metis Arr.simps(1) Ide.elims(3) Ide.simps(3) seq-char)
  qed
  finally show PCong (T @ (b # B)) T by blast
qed

lemma PCong-imp-Cong:
shows PCong T U ⟹ T *∼* U
proof (induct rule: PCong.induct)
  show ⋀T. Arr T ⟹ T *∼* T
    using cong-reflexive by blast
  show ⋀T U. ⟦PCong T U; T *∼* U⟧ ⟹ U *∼* T
    by blast
  show ⋀T U V. ⟦PCong T U; T *∼* U; PCong U V; U *∼* V⟧ ⟹ T *∼* V
    using cong-transitive by blast
  show ⋀T U U' T'. ⟦seq T U; PCong T T'; T *∼* T'; PCong U U'; U *∼* U'⟧
                    ⟹ T @ U *∼* T' @ U'
    using cong-append by simp
  show ⋀T b. ⟦seq T [b]; R.ide b⟧ ⟹ T @ [b] *∼* T
    using cong-append-ideI(4) ide-char by force
  show ⋀t u. t ⌢ u ⟹ [t, u \ t] *∼* [u, t \ u]
  proof −
    have ⋀t u. t ⌢ u ⟹ [t, u \ t] *≲* [u, t \ u]
    proof −
      fix t u
      assume con: t ⌢ u
      have ([t] @ [u \ t]) *\* ([u] @ [t \ u]) =
            [(t \ u) \ (t \ u), ((u \ t) \ (u \ t)) \ ((t \ u) \ (t \ u))]
        using con Resid-append2 [of [t] [u \ t] [u] [t \ u]]
        apply simp
        by (metis R.arr-resid-iff-con R.con-target R.conE R.con-sym
            R.prfx-implies-con R.prfx-reflexive R.cube)
      moreover have Ide ...
        using con
        by (metis Arr.simps(2) Arr.simps(3) Ide.simps(2) Ide.simps(3) R.arr-resid-iff-con
            R.con-sym R.resid-ide-arr R.prfx-reflexive calculation Con-imp-Arr-Resid)
      ultimately show[t, u \ t] *≲* [u, t \ u]
```

```
      using ide-char by auto
    qed
    thus ⋀t u. t ⌢ u ⟹ [t, u \ t] *∼* [u, t \ u]
      using R.con-sym by blast
  qed
qed

lemma PCong-permute-single:
shows [t] *⌢* U ⟹ PCong ([t] @ (U *\* [t])) (U @ ([t] *\* U))
proof (induct U arbitrary: t)
  show ⋀t. [t] *⌢* [] ⟹ PCong ([t] @ [] *\* [t]) ([] @ [t] *\* [])
    by auto
  fix t u U
  assume ind: ⋀t. [t] *\* U ≠ [] ⟹ PCong ([t] @( U *\* [t])) (U @ ([t] *\* U))
  assume con: [t] *⌢* u # U
  show PCong ([t] @ (u # U) *\* [t]) ((u # U) @ [t] *\* (u # U))
  proof (cases U = [])
    show U = [] ⟹ ?thesis
      by (metis PCong.intros(6) Resid.simps(3) append-Cons append-eq-append-conv2
          append-self-conv con-char con-def con con-sym-ax)
    assume U: U ≠ []
    show PCong ([t] @ ((u # U) *\* [t])) ((u # U) @ ([t] *\* (u # U)))
    proof −
      have [t] @ ((u # U) *\* [t]) = [t] @ ([u \ t] @ (U *\* [t \ u]))
        using Con-sym Resid-rec(2) U con by auto
      also have ... = ([t] @ [u \ t]) @ (U *\* [t \ u])
        by auto
      also have PCong ... (([u] @ [t \ u]) @ (U *\* [t \ u]))
      proof −
        have PCong ([t] @ [u \ t]) ([u] @ [t \ u])
          using con
          by (simp add: Con-rec(3) PCong.intros(6) U)
        thus ?thesis
          by (metis Arr-Resid-single Con-implies-Arr(1) Con-rec(2) Con-sym
              PCong.intros(1,4) Srcs-Resid U append-is-Nil-conv append-is-composite-of
              arr-append-imp-seq arr-char calculation composite-of-unq-upto-cong
              con not-arr-null null-char ide-implies-arr seq-char)
      qed
      also have ([u] @ [t \ u]) @ (U *\* [t \ u]) = [u] @ ([t \ u] @ (U *\* [t \ u]))
        by simp
      also have PCong ... ([u] @ (U @ ([t \ u] *\* U)))
      proof −
        have PCong ([t \ u] @ (U *\* [t \ u])) (U @ ([t \ u] *\* U))
          using ind
          by (metis Resid-rec(3) U con)
        moreover have seq [u] ([t \ u] @ U *\* [t \ u])
        proof
          show Arr [u]
            using Con-implies-Arr(2) Con-initial-right con by blast
```

```
      show Arr ([t \ u] @ U *\* [t \ u])
        using Con-implies-Arr(1) U con Con-imp-Arr-Resid Con-rec(3) Con-sym
        by fastforce
      show Trgs [u] ∩ Srcs ([t \ u] @ U *\* [t \ u]) ≠ {}
        by (metis Arr.simps(1) Arr.simps(2) Arr-has-Trg Con-implies-Arr(1)
            Int-absorb R.arr-resid-iff-con R.sources-resid Resid-rec(3)
            Srcs.simps(2) Srcs-append Trgs.simps(2) U ‹Arr [u]› con)
    qed
    moreover have PCong [u] [u]
      using PCong.intros(1) calculation(2) seq-char by force
    ultimately show ?thesis
      using U arr-append arr-char con seq-char
            PCong.intros(4) [of [u] [t \ u] @ (U *\* [t \ u])
                                [u] U @ ([t \ u] *\* U)]
      by blast
  qed
  also have ([u] @ (U @ ([t \ u] *\* U))) = ((u # U) @ [t] *\* (u # U))
    by (metis Resid-rec(3) U append-Cons append-Nil con)
  finally show ?thesis by blast
 qed
qed

lemma PCong-permute:
shows T *⌢* U ⟹ PCong (T @ (U *\* T)) (U @ (T *\* U))
proof (induct T arbitrary: U)
 show ⋀U. [] *\* U ≠ [] ⟹ PCong ([] @ U *\* []) (U @ [] *\* U)
   by simp
 fix t T U
 assume ind: ⋀U. T *⌢* U ⟹ PCong (T @ (U *\* T)) (U @ (T *\* U))
 assume con: t # T *⌢* U
 show PCong ((t # T) @ (U *\* (t # T))) (U @ ((t # T) *\* U))
 proof (cases T = [])
   assume T: T = []
   have (t # T) @ (U *\* (t # T)) = [t] @ (U *\* [t])
     using con T by simp
   also have PCong ... (U @ ([t] *\* U))
     using PCong-permute-single T con by blast
   finally show ?thesis
     using T by fastforce
   next
   assume T: T ≠ []
   have (t # T) @ (U *\* (t # T)) = [t] @ (T @ (U *\* (t # T)))
     by simp
   also have PCong ... ([t] @ (U *\* [t]) @ (T *\* (U *\* [t])))
     using ind [of U *\* [t]]
     by (metis Arr.simps(1) Con-imp-Arr-Resid Con-implies-Arr(2) Con-sym
         PCong.intros(1,4) Resid-cons(2) Srcs-Resid T arr-append arr-append-imp-seq
         calculation con not-arr-null null-char seq-char)
```

```
    also have [t] @ (U *\* [t]) @ (T *\* (U *\* [t])) =
              ([t] @ (U *\* [t])) @ (T *\* (U *\* [t]))
      by simp
    also have PCong (([t] @ (U *\* [t])) @ (T *\* (U *\* [t])))
                    ((U @ ([t] *\* U)) @ (T *\* (U *\* [t])))
      by (metis Arr.simps(1) Con-cons(1) Con-imp-Arr-Resid Con-implies-Arr(2)
          PCong.intros(1,4) PCong-permute-single Srcs-Resid T Trgs-append arr-append
          arr-char con seq-char)
    also have (U @ ([t] *\* U)) @ (T *\* (U *\* [t])) = U @ ((t # T) *\* U)
      by (metis Resid.simps(2) Resid-cons(1) append.assoc con)
    finally show ?thesis by blast
  qed
qed

lemma Cong-imp-PCong:
assumes T *~* U
shows PCong T U
proof -
  have PCong T (T @ (U *\* T))
    using assms PCong.intros(2) PCong-append-Ide
    by (metis Con-implies-Arr(1) Ide.simps(1) Srcs-Resid ide-char Con-imp-Arr-Resid
        seq-char)
  also have PCong (T @ (U *\* T)) (U @ (T *\* U))
    using PCong-permute assms con-char prfx-implies-con by presburger
  also have PCong (U @ (T *\* U)) U
    using assms PCong-append-Ide
    by (metis Con-imp-Arr-Resid Con-implies-Arr(1) Srcs-Resid arr-resid-iff-con
        ide-implies-arr con-char ide-char seq-char)
  finally show ?thesis by blast
qed

lemma Cong-iff-PCong:
shows T *~* U ⟷ PCong T U
  using PCong-imp-Cong Cong-imp-PCong by blast

end
```

2.5 Composite Completion

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

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

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

interpretation N: *coherent-normal-sub-rts R ‹Collect R.ide›*
  using *R.rts-axioms R.identities-form-coherent-normal-sub-rts* **by** *auto*
sublocale P: *paths-in-rts-with-coherent-normal R ‹Collect R.ide›* **..**
sublocale Q: *quotient-by-coherent-normal P.Resid ‹Collect P.NPath›* **..**

definition *resid* (**infix** ‹⦇**⦈› *70*)
where *resid* ≡ *Q.Resid*

sublocale *extensional-rts resid*
  unfolding *resid-def*
  using *Q.extensional-rts-axioms* **by** *simp*

notation *con* (**infix** ‹⦇*⌢*⦈› *50*)
notation *prfx* (**infix** ‹⦇*≾*⦈› *50*)

notation *P.Resid* (**infix** ‹**› *70*)
notation *P.Con* (**infix** ‹*⌢*› *50*)
notation *P.Cong* (**infix** ‹*≈*› *50*)
notation $P.Cong_0$ (**infix** ‹*≈₀*› *50*)
notation *P.Cong-class* (‹⦇-⦈›)

lemma *P-ide-iff-NPath*:
shows *P.ide T* ⟷ *P.NPath T*
  using *P.NPath-def P.ide-char P.ide-closed* **by** *auto*

lemma $Cong\text{-}eq\text{-}Cong_0$:
shows *T* ≈* *T′* ⟷ *T* ≈₀* *T′*
  by (*metis Collect-cong P.Cong-iff-cong P-ide-iff-NPath mem-Collect-eq*)

lemma *Srcs-respects-Cong*:
assumes *T* ≈* *T′*
shows *P.Srcs T* = *P.Srcs T′*
  using *assms*
  by (*meson P.Con-imp-eq-Srcs* $P.Cong_0$*-imp-con P.con-char* $Cong\text{-}eq\text{-}Cong_0$)

lemma *sources-respects-Cong*:
assumes *T* ≈* *T′*
shows *P.sources T* = *P.sources T′*
  using *assms*
  by (*meson* $P.Cong_0$*-imp-coinitial* $Cong\text{-}eq\text{-}Cong_0$)

lemma *Trgs-respects-Cong*:
assumes *T* ≈* *T′*
shows *P.Trgs T* = *P.Trgs T′*
  by (*metis* $Cong\text{-}eq\text{-}Cong_0$ *P.Srcs-Resid P.con-char P.cube P.ide-def*
      P-ide-iff-NPath assms mem-Collect-eq)

lemma *targets-respects-Cong*:

assumes $T\ {}^*\!\approx^*\ T'$
shows *P.targets* T = *P.targets* T'
using *assms P.Cong-imp-arr(1) P.Cong-imp-arr(2) P.arr-iff-has-target P.targets-char*$_P$ *Trgs-respects-Cong*
by *force*

lemma *ide-char*$_{CC}$:
shows *ide* $\mathcal{T} \longleftrightarrow$ *arr* $\mathcal{T} \wedge (\forall\, T.\ T \in \mathcal{T} \longrightarrow$ *P.Ide* $T)$
by (*metis Ball-Collect P.IdeI P.Ide-implies-NPath Q.ide-char' P.NPath-def resid-def*)

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
show *arr* $\mathcal{T} \wedge$ *arr* $\mathcal{U} \wedge$ *P.Cong-class-rep* $\mathcal{T}\ {}^*\!\frown^*$ *P.Cong-class-rep* $\mathcal{U} \Longrightarrow \mathcal{T}\ \{\!|{}^*\!\frown^*|\!\}\ \mathcal{U}$
by (*metis P.Cong-class-rep P.conI*$_P$ *Q.arr-char Q.quot.preserves-con resid-def*)
show $\mathcal{T}\ \{\!|{}^*\!\frown^*|\!\}\ \mathcal{U} \Longrightarrow$ *arr* $\mathcal{T} \wedge$ *arr* $\mathcal{U} \wedge$ *P.Cong-class-rep* $\mathcal{T}\ {}^*\!\frown^*$ *P.Cong-class-rep* $\mathcal{U}$
proof −
assume *con*: $\mathcal{T}\ \{\!|{}^*\!\frown^*|\!\}\ \mathcal{U}$
have *1*: *arr* $\mathcal{T} \wedge$ *arr* $\mathcal{U}$
using *con coinitial-iff con-imp-coinitial* **by** *blast*
moreover have *P.Cong-class-rep* $\mathcal{T}\ {}^*\!\frown^*$ *P.Cong-class-rep* $\mathcal{U}$
proof −
obtain $T\ U$ **where** *TU*: $T \in \mathcal{T} \wedge U \in \mathcal{U} \wedge$ *P.Con* $T\ U$
using *P.con-char Q.con-char*$_{QCN}$ *con resid-def* **by** *auto*
have $T\ {}^*\!\approx^*$ *P.Cong-class-rep* $\mathcal{T} \wedge U\ {}^*\!\approx^*$ *P.Cong-class-rep* $\mathcal{U}$
using *TU 1 P.Cong-class-memb-Cong-rep Q.arr-char resid-def* **by** *simp*
thus *?thesis*
using *TU P.Cong-subst(1)* [*of* T *P.Cong-class-rep* $\mathcal{T}$ U *P.Cong-class-rep* $\mathcal{U}$]
by (*metis P.coinitial-iff P.con-char P.con-imp-coinitial sources-respects-Cong*)
qed
ultimately show *?thesis* **by** *simp*
qed
qed

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
show *arr* $\mathcal{T} \wedge$ *arr* $\mathcal{U} \wedge (\forall\, T\ U.\ T \in \mathcal{T} \wedge U \in \mathcal{U} \longrightarrow T\ {}^*\!\frown^*\ U) \Longrightarrow \mathcal{T}\ \{\!|{}^*\!\frown^*|\!\}\ \mathcal{U}$
using *con-char*$_{CC}$ *P.rep-in-Cong-class Q.arr-char resid-def* **by** *simp*
show $\mathcal{T}\ \{\!|{}^*\!\frown^*|\!\}\ \mathcal{U} \Longrightarrow$ *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 (*intro conjI allI impI*)
assume *1*: $\mathcal{T}\ \{\!|{}^*\!\frown^*|\!\}\ \mathcal{U}$
show *arr* $\mathcal{T}$
using *1 con-implies-arr* **by** *simp*
show *arr* $\mathcal{U}$
using *1 con-implies-arr* **by** *simp*
fix $T\ U$
assume *2*: $T \in \mathcal{T} \wedge U \in \mathcal{U}$

```
    show T *⌢* U
    proof −
      have P.Cong T (P.Cong-class-rep T)
        using ‹arr T› by (simp add: 2 P.Cong-class-memb-Cong-rep Q.arr-char resid-def)
      moreover have P.con (P.Cong-class-rep T) (P.Cong-class-rep U)
        using 1 con-char_CC by blast
      moreover have P.Cong (P.Cong-class-rep U) U
        using ‹arr U› by (simp add: 2 P.Cong-class-membs-are-Cong P.rep-in-Cong-class
            Q.arr-char resid-def)
      ultimately show ?thesis
        by (meson Cong-eq-Cong_0 P.Cong_0-subst-Con P.con-char)
    qed
  qed
qed

lemma resid-char:
shows T ⟦*\*⟧ U =
      (if T ⟦*⌢*⟧ U then ⟦P.Cong-class-rep T *\* P.Cong-class-rep U⟧ else {})
  by (metis P.con-char P.rep-in-Cong-class Q.Resid-by-members Q.arr-char arr-resid-iff-con
      con-char_CC Q.is-Cong-class-Resid resid-def)

lemma resid-simp:
assumes T ⟦*⌢*⟧ U and T ∈ T and U ∈ U
shows T ⟦*\*⟧ U = ⟦P.Resid T U⟧
  using assms resid-char
  by (metis (no-types, lifting) P.con-char con-char_CC' Q.Resid-by-members
      Q.con-char_QCN resid-def)

lemma src-char':
shows src T = {A. arr T ∧ P.Ide A ∧ P.Srcs (P.Cong-class-rep T) = P.Srcs A}
proof (cases arr T)
  show ¬ arr T ⟹ ?thesis
    by (simp add: Q.null-char Q.src-def resid-def)
  assume T: arr T
  have 1: ∃A. P.Ide A ∧ P.Srcs (P.Cong-class-rep T) = P.Srcs A
    by (metis P.Con-imp-eq-Srcs P.con-char P.con-imp-coinitial-ax P.ide-char
        T con-char_CC arrE)
  let ?A = SOME A. P.Ide A ∧ P.Srcs (P.Cong-class-rep T) = P.Srcs A
  have A: P.Ide ?A ∧ P.Srcs (P.Cong-class-rep T) = P.Srcs ?A
    using 1 someI-ex [of λA. P.Ide A ∧ P.Srcs (P.Cong-class-rep T) = P.Srcs A] by simp
  have a: arr ⟦?A⟧
    using A P.ide-char P.is-Cong-classI Q.arr-char resid-def by auto
  have ide-a: ide ⟦?A⟧
  using a A P.Cong-class-def P.normal-is-Cong-closed P-ide-iff-NPath P.ide-char ide-char_CC
    by auto
  have sources T = {⟦?A⟧}
  proof −
```

```
    have 𝒯 ⦇*⌢*⦈ ⦇?A⦈
      by (metis (no-types, lifting) A P.Cong-class-rep P.conI_P Q.quot.preserves-con
          𝒯 con-arr-self con-char_CC P.Arr-iff-Con-self P.Con-IdeI(2) Q.arr-char resid-def)
    hence ⦇?A⦈ ∈ sources 𝒯
      using ide-a in-sourcesI by simp
    thus ?thesis
      using sources-char_WE by auto
  qed
  moreover have ⦇?A⦈ = {A. P.Ide A ∧ P.Srcs (P.Cong-class-rep 𝒯) = P.Srcs A}
  proof
    show {A. P.Ide A ∧ P.Srcs (P.Cong-class-rep 𝒯) = P.Srcs A} ⊆ ⦇?A⦈
      using A P.Cong-class-def P.Cong-closure-props(3) P.Ide-implies-Arr
            P.ide-closed P.ide-char
      by fastforce
    show ⦇?A⦈ ⊆ {A. P.Ide A ∧ P.Srcs (P.Cong-class-rep 𝒯) = P.Srcs A}
      using a A P.Cong-class-def Srcs-respects-Cong ide-a ide-char_CC by blast
  qed
  ultimately show ?thesis
    using 𝒯 src-in-sources sources-char_WE by auto
qed

lemma src-char:
shows src 𝒯 = {A. arr 𝒯 ∧ P.Ide A ∧ (∀ T. T ∈ 𝒯 ⟶ P.Srcs T = P.Srcs A)}
proof (cases arr 𝒯)
  show ¬ arr 𝒯 ⟹ ?thesis
    using src-char' by simp
  assume 𝒯: arr 𝒯
  have ⋀T. T ∈ 𝒯 ⟹ P.Srcs T = P.Srcs (P.Cong-class-rep 𝒯)
    using 𝒯 P.Cong-class-memb-Cong-rep Srcs-respects-Cong Q.arr-char resid-def by auto
  thus ?thesis
    using 𝒯 src-char' P.is-Cong-class-def Q.arr-char resid-def by force
qed

lemma trg-char':
shows trg 𝒯 = {B. arr 𝒯 ∧ P.Ide B ∧ P.Trgs (P.Cong-class-rep 𝒯) = P.Srcs B}
proof (cases arr 𝒯)
  show ¬ arr 𝒯 ⟹ ?thesis
    using resid-char resid-def Q.trg-char_QCN by auto
  assume 𝒯: arr 𝒯
  have 1: ∃B. P.Ide B ∧ P.Trgs (P.Cong-class-rep 𝒯) = P.Srcs B
    by (metis P.Con-implies-Arr(2) P.Resid-Arr-self P.Srcs-Resid 𝒯 con-char_CC arrE)
  define B where B = (SOME B. P.Ide B ∧ P.Trgs (P.Cong-class-rep 𝒯) = P.Srcs B)
  have B: P.Ide B ∧ P.Trgs (P.Cong-class-rep 𝒯) = P.Srcs B
    unfolding B-def
    using 1 someI-ex [of λB. P.Ide B ∧ P.Trgs (P.Cong-class-rep 𝒯) = P.Srcs B] by simp
  hence 2: P.Ide B ∧ P.Con (P.Resid (P.Cong-class-rep 𝒯) (P.Cong-class-rep 𝒯)) B
    using 𝒯
    by (metis (no-types, lifting) P.Con-Ide-iff P.Ide-implies-Arr P.Resid-Arr-self
        P.Srcs-Resid arrE P.Con-implies-Arr(2) con-char_CC)
```

have *b*: *arr* ⦇*B*⦈
by (*simp add*: *2 P.ide-char P.is-Cong-classI Q.arr-char resid-def*)
have *ide-b*: *ide* ⦇*B*⦈
by (*simp add*: *2 P.ide-char resid-def*)
have *targets* $\mathcal{T}$ = {⦇*B*⦈}
proof −
have *cong* ($\mathcal{T}$ ⦇**⦈ $\mathcal{T}$) ⦇*B*⦈
by (*metis 2 P.con-char Q.ide-imp-con-iff-cong Q.quot.preserves-con*
$\mathcal{T}$ *composite-completion.resid-char composite-completion-axioms*
cong-reflexive ide-b resid-def)
thus *?thesis*
using $\mathcal{T}$ *Q.targets-char*$_{QCN}$ [*of* $\mathcal{T}$] *cong-char resid-def* **by** *auto*
qed
moreover have ⦇*B*⦈ = {*B*. *P.Ide B* ∧ *P.Trgs* (*P.Cong-class-rep* $\mathcal{T}$) = *P.Srcs B*}
proof
show {*B*. *P.Ide B* ∧ *P.Trgs* (*P.Cong-class-rep* $\mathcal{T}$) = *P.Srcs B*} ⊆ ⦇*B*⦈
using *B P.Cong-class-def P.Cong-closure-props*(*3*) *P.Ide-implies-Arr*
P.ide-closed P.ide-char
by *force*
show ⦇*B*⦈ ⊆ {*B*. *P.Ide B* ∧ *P.Trgs* (*P.Cong-class-rep* $\mathcal{T}$) = *P.Srcs B*}
proof −
have ⋀*B'*. *P.Cong B' B* ⟹ *P.Ide B'* ∧ *P.Trgs* (*P.Cong-class-rep* $\mathcal{T}$) = *P.Srcs B'*
using *B P-ide-iff-NPath P.normal-is-Cong-closed Srcs-respects-Cong*
by (*metis P.Cong-closure-props*(*1*) *P.ide-char mem-Collect-eq*)
thus *?thesis*
using *P.Cong-class-def* **by** *blast*
qed
qed
ultimately show *?thesis*
using $\mathcal{T}$ *trg-in-targets targets-char* **by** *auto*
qed

lemma *trg-char*:
shows *trg* $\mathcal{T}$ = {*B*. *arr* $\mathcal{T}$ ∧ *P.Ide B* ∧ (∀ *T*. *T* ∈ $\mathcal{T}$ ⟶ *P.Trgs T* = *P.Srcs B*)}
proof (*cases arr* $\mathcal{T}$)
show ¬ *arr* $\mathcal{T}$ ⟹ *?thesis*
using *trg-char'* **by** *presburger*
assume $\mathcal{T}$: *arr* $\mathcal{T}$
have ⋀*T*. *T* ∈ $\mathcal{T}$ ⟹ *P.Trgs T* = *P.Trgs* (*P.Cong-class-rep* $\mathcal{T}$)
using $\mathcal{T}$
by (*metis P.Cong-class-memb-Cong-rep Trgs-respects-Cong Q.arr-char resid-def*)
thus *?thesis*
using $\mathcal{T}$ *trg-char' P.is-Cong-class-def Q.arr-char resid-def* **by** *force*
qed

lemma *prfx-char*:
shows $\mathcal{T}$ ⦇*≲*⦈ $\mathcal{U}$ ⟷ *arr* $\mathcal{T}$ ∧ *arr* $\mathcal{U}$ ∧ (∀ *T U*. *T* ∈ $\mathcal{T}$ ∧ *U* ∈ $\mathcal{U}$ ⟶ *P.prfx T U*)
proof
show $\mathcal{T}$ ⦇*≲*⦈ $\mathcal{U}$ ⟹ *arr* $\mathcal{T}$ ∧ *arr* $\mathcal{U}$ ∧ (∀ *T U*. *T* ∈ $\mathcal{T}$ ∧ *U* ∈ $\mathcal{U}$ ⟶ *P.prfx T U*)

by (*metis* (*mono-tags*, *lifting*) *Ball-Collect P.arr-in-Cong-class*
P.arr-resid-iff-con P.conI$_P$ *P-ide-iff-NPath Q.ide-char'*
con-char$_{CC}$*' prfx-implies-con resid-def resid-simp*)
show *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*$) \Longrightarrow \mathcal{T}\ \{\!|^*\lesssim^*|\!\}\ \mathcal{U}$
by (*metis P.Cong-class-rep P.rep-in-Cong-class Q.arr-char Q.quot.preserves-prfx*
resid-def)
qed

lemma *quotient-reflects-con*:
assumes *con* (*Q.quot t*) (*Q.quot u*)
shows *P.con t u*
using *assms P.arr-in-Cong-class P.con-char con-char*$_{CC}$*' resid-def* **by** *force*

lemma *is-extensional-rts-with-composites*:
shows *extensional-rts-with-composites resid*
proof
fix $\mathcal{T}\ \mathcal{U}$
assume *seq*: *seq* $\mathcal{T}\ \mathcal{U}$
obtain *T* **where** *T*: $\mathcal{T} = \{\!|T|\!\}$
by (*metis P.Cong-class-rep Q.arr-char Q.seqE*$_{WE}$ *resid-def seq*)
obtain *U* **where** *U*: $\mathcal{U} = \{\!|U|\!\}$
by (*metis P.Cong-class-rep Q.arr-char Q.seqE*$_{WE}$ *resid-def seq*)
have *1*: *P.Arr T* $\wedge$ *P.Arr U*
using *P.arr-char T U resid-def seq* **by** *auto*
have *2*: *P.Trgs T* = *P.Srcs U*
proof −
have *P.Trgs* (*P.Cong-class-rep* $\mathcal{T}$) = *P.Srcs* (*P.Cong-class-rep* $\mathcal{U}$)
by (*metis* (*mono-tags*, *lifting*) *P.Con-imp-eq-Srcs P.rep-in-Cong-class*
Q.arr-char con-arr-src(*2*) *con-char*$_{CC}$*' mem-Collect-eq resid-def*
seq seqE$_{WE}$ *trg-char'*)
thus *?thesis*
by (*metis 1 P.Cong-class-memb-Cong-rep P.arrI*$_P$ *P.arr-in-Cong-class*
P.is-Cong-classI Srcs-respects-Cong T Trgs-respects-Cong U)
qed
have *P.Arr* (*T* @ *U*)
using *1 2* **by** *simp*
moreover have *P.Ide* (*T* $^*\backslash^*$ (*T* @ *U*))
by (*metis 1 P.Con-append*(*2*) *P.Con-sym P.Resid-Arr-self P.Resid-Ide-Arr-ind*
P.Resid-append(*2*) *P.Trgs.simps*(*1*) *calculation P.Arr-has-Trg*)
moreover have (*T* @ *U*) $^*\backslash^*$ *T* $^*\approx^*$ *U*
by (*metis 1 P.Arr.simps*(*1*) *P.Con-implies-Arr*(*2*) *P.Cong*$_0$*-implies-Cong*
P.con-sym-ax P.null-char calculation(*2*) *P.Cong*$_0$*-append-resid-NPath*
P.Cong$_0$*-cancel-left*$_{CS}$ *P.Ide-implies-NPath*)
ultimately have *composite-of* $\mathcal{T}\ \mathcal{U}$ $\{\!|T\ @\ U|\!\}$
by (*metis 1 P.Arr.simps*(*1*) *P.append-is-composite-of P.arrI*$_P$ *P.arr-append-imp-seq*
Q.quot.preserves-composites T U resid-def)
thus *composable* $\mathcal{T}\ \mathcal{U}$
using *composable-def* **by** *auto*
qed

sublocale *extensional-rts-with-composites resid*
using *is-extensional-rts-with-composites* **by** *simp*

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

2.5.1 Inclusion Map

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

sublocale *incl*: *simulation R resid incl*
unfolding *incl-def resid-def*
using *P.incl-is-simulation Q.quotient-is-simulation composite-simulation.intro*
by *blast*
sublocale *incl*: *simulation-to-extensional-rts R resid incl* **..**

lemma *incl-is-simulation*:
shows *simulation R resid incl*
..

lemma *incl-simp* [*simp*]:
shows *incl t* = $\{\!|[t]|\!\}$
by (*metis* (*mono-tags*, *lifting*) *P.Arr.simps*(*2*) *P.Arr-iff-Con-self P.Ide.simps*(*1*)
P.cong-reflexive P.ide-char Q.quot.extensionality incl.extensionality incl-def
o-apply resid-def)

lemma *incl-reflects-con*:
assumes *incl t* $\{\!|^{*}\frown^{*}|\!\}$ *incl u*
shows *R.con t u*
by (*metis P.Con-rec*(*1*) *P.arr-in-Cong-class assms con-char*$_{CC}$*' incl-simp*
Q.quot.preserves-reflects-arr resid-def)

lemma *cong-iff-eq-incl*:
assumes *R.arr t* **and** *R.arr u*
shows *incl t* = *incl u* $\longleftrightarrow$ *R.cong t u*
by (*metis P.Ide.simps*(*2*) *P.arr-in-Cong-class assms*(*1*) *incl-reflects-con*
conE cong-char ide-char$_{CC}$ *incl.preserves-prfx incl.preserves-reflects-arr*
incl.preserves-resid incl-simp prfx-implies-con Q.quot.extensionality resid-def)

lemma *incl-cancel-left*:
assumes *transformation X R F G T* **and** *transformation X R F' G' T'*
and *extensional-rts R*
and *incl* $\circ$ *T* = *incl* $\circ$ *T'*
shows *T* = *T'*
proof −
interpret *R*: *extensional-rts R*
using *assms*(*3*) **by** *blast*
interpret *T*: *transformation X R F G T*

```
    using assms(1) by blast
  interpret T': transformation X R F' G' T'
    using assms(2) by blast
  show T = T'
  proof
    fix x
    show T x = T' x
    by (metis R.cong-char T'.extensionality T.A.cong-reflexive T.extensionality
      T.respects-cong assms(4) comp-apply cong-iff-eq-incl incl.preserves-reflects-arr)
  qed
qed
```

The inclusion is surjective on identities.

```
lemma img-incl-ide:
shows incl ‘ (Collect R.ide) = Collect ide
proof
  show incl ‘ Collect R.ide ⊆ Collect ide
    using incl.preserves-ide by force
  show Collect ide ⊆ incl ‘ Collect R.ide
  proof
    fix A
    assume A: A ∈ Collect ide
    obtain A where A: A ∈ A
      using A Q.ide-char resid-def by auto
    have P.NPath A
      by (metis A Ball-Collect A Q.ide-char' mem-Collect-eq resid-def)
    obtain a where a: a ∈ P.Srcs A
      using ‹P.NPath A›
      by (meson P.NPath-implies-Arr equals0I P.Arr-has-Src)
    have A = ⦇[a]⦈
    proof −
      have A *≈₀* [a]
        by (metis P.Con-Arr-self P.Con-imp-eq-Srcs P.Con-implies-Arr(1)
            P.Con-sym P.NPath-implies-Arr P.Resid-Arr-Src P.conI_P P.ideI
            P.ide-closed P.resid-ide-arr ‹P.NPath A› a mem-Collect-eq)
      thus ?thesis
        by (metis A CollectD P.Cong₀-imp-con P.Cong₀-subst-left(1)
            P.Cong-class-eqI P.Cong-closure-props(3) P.resid-arr-ide
            P-ide-iff-NPath Q.ideE A ‹P.NPath A› resid-def resid-simp)
    qed
    thus A ∈ incl ‘ Collect R.ide
      using P.Srcs-are-ide a by auto
  qed
qed

end
```

2.5.2 Composite Completion of a Weakly Extensional RTS

```
locale composite-completion-of-weakly-extensional-rts =
```

```
  R: weakly-extensional-rts R +
  composite-completion
begin

  sublocale P: paths-in-weakly-extensional-rts R ..
  sublocale incl: simulation-between-weakly-extensional-rts R resid incl ..

  notation comp (infixr ⦇*·*⦈ 55)

  lemma src-char_CCWE:
  shows src 𝒯 = (if arr 𝒯 then incl (P.Src (P.Cong-class-rep 𝒯)) else null)
  proof (cases arr 𝒯)
    show ¬ arr 𝒯 ⟹ ?thesis
      using src-def by auto
    assume 𝒯: arr 𝒯
    have src 𝒯 = {A. P.Ide A ∧ P.Srcs (P.Cong-class-rep 𝒯) = P.Srcs A}
      using 𝒯 src-char' [of 𝒯] by simp
    moreover have 1: ⋀A. P.Ide A ⟹
                        P.Srcs (P.Cong-class-rep 𝒯) = P.Srcs A ⟷
                        P.Src (P.Cong-class-rep 𝒯) = P.Src A
      by (metis 𝒯 P.Con-implies-Arr(2) P.Ide-implies-Arr con-arr-self con-char_CC
          insertCI P.Srcs-simp_PWE singletonD)
    ultimately have 2: src 𝒯 = {A. P.Ide A ∧ P.Src (P.Cong-class-rep 𝒯) = P.Src A}
      by blast
    also have ... = incl (P.Src (P.Cong-class-rep 𝒯))
    proof −
      have incl (P.Src (P.Cong-class-rep 𝒯)) = Q.quot [R.src (hd (P.Cong-class-rep 𝒯))]
        by auto
      also have ... = {A. P.Ide A ∧ P.Src (P.Cong-class-rep 𝒯) = P.Src A}
        apply auto[1]
          apply (metis Q.null-char R.ide-iff-src-self R.src-src empty-iff ide-char_CC
              incl.extensionality incl.preserves-ide incl-simp resid-def)
         apply (metis 1 CollectD P.Cong-class-def P.Ide.simps(2)
            P.paths-in-weakly-extensional-rts-axioms Q.null-char R.rts-axioms R.src-trg
            R.trg-src Srcs-respects-Cong calculation composite-completion.ide-char_CC
            composite-completion-axioms empty-iff incl.extensionality incl.preserves-ide
            list.sel(1) paths-in-weakly-extensional-rts.Src.elims resid-def rts.ide-src)
        by (metis (mono-tags, lifting) 2 P.Cong-class-def P.Ide.simps(2)
            P.Ide-imp-Ide-hd P.Src.elims P.is-Cong-classE Q.arr-char Q.src-src R.src-ide
            list.sel(1) mem-Collect-eq resid-def src-char')
      finally show ?thesis by blast
    qed
    finally show ?thesis
      using 𝒯 by auto
  qed

  lemma trg-char_CCWE:
  shows trg 𝒯 = (if arr 𝒯 then incl (P.Trg (P.Cong-class-rep 𝒯)) else null)
  proof (cases arr 𝒯)
```

show $\neg$ *arr* $\mathcal{T} \Longrightarrow$ *?thesis*
using *trg-def* **by** *auto*
assume $\mathcal{T}$: *arr* $\mathcal{T}$
have *trg* $\mathcal{T} = \{A.\ P.Ide\ A \wedge P.Trg\ (P.Cong\text{-}class\text{-}rep\ \mathcal{T}) = P.Src\ A\}$
proof −
have *trg* $\mathcal{T} = \{B.\ P.Ide\ B \wedge P.Trgs\ (P.Cong\text{-}class\text{-}rep\ \mathcal{T}) = P.Srcs\ B\}$
using $\mathcal{T}$ *trg-char'* [*of* $\mathcal{T}$] **by** *simp*
moreover have $\bigwedge B.\ P.Ide\ B \Longrightarrow$
$P.Trgs\ (P.Cong\text{-}class\text{-}rep\ \mathcal{T}) = P.Srcs\ B \longleftrightarrow$
$P.Trg\ (P.Cong\text{-}class\text{-}rep\ \mathcal{T}) = P.Src\ B$
by (*metis P.Con-implies-Arr*(*1*) *P.Ide-implies-Arr* $\mathcal{T}$ *con-arr-self con-char*$_{CC}$
P.Srcs-simp$_{PWE}$ *P.Trgs-simp*$_{PWE}$ *singleton-inject*)
ultimately show *?thesis* **by** *blast*
qed
also have ... = *incl* (*P.Trg* (*P.Cong-class-rep* $\mathcal{T}$))
using *incl.preserves-trg*
by (*metis* (*mono-tags*, *lifting*) *P.rep-in-Cong-class* $\mathcal{T}$ *arr-trg-iff-arr*
calculation mem-Collect-eq Q.arr-char resid-def src-char$_{CCWE}$ *src-trg*)
finally show *?thesis*
using $\mathcal{T}$ **by** *auto*
qed

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

lemma *incl-injective-on-ide*:
shows *inj-on incl* (*Collect R.ide*)
by (*metis R.con-ide-are-eq ideE incl.preserves-ide incl-reflects-con inj-onCI*
mem-Collect-eq)

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

lemma *incl-bijective-on-ide*:
shows *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-betw* $(\lambda\mathcal{A}.\ P.Src\ (P.Cong\text{-}class\text{-}rep\ \mathcal{A}))$ (*Collect ide*) (*Collect R.ide*)
proof −
show *1*: *incl* $\in$ *Collect R.ide* $\rightarrow$ *Collect ide*
using *img-incl-ide* **by** *auto*
show *2*: $(\lambda\mathcal{A}.\ P.Src\ (P.Cong\text{-}class\text{-}rep\ \mathcal{A})) \in$ *Collect ide* $\rightarrow$ *Collect R.ide*
by (*metis* (*no-types*, *lifting*) *P.Src.elims Pi-I' R.ide-iff-trg-self*
R.trg-src ide-implies-arr incl.preserves-reflects-arr mem-Collect-eq
src-char$_{CCWE}$ *src-ide*)
show *3*: $\bigwedge a.\ R.ide\ a \Longrightarrow (\lambda\mathcal{A}.\ P.Src\ (P.Cong\text{-}class\text{-}rep\ \mathcal{A}))\ (incl\ a) = a$
by (*metis R.ide-backward-stable R.weak-extensionality cong-iff-eq-incl*
incl.preserves-ide incl.preserves-reflects-arr ide-implies-arr
src-char$_{CCWE}$ *src-ide*)

show *4*: $\bigwedge \mathcal{A}$. *ide* $\mathcal{A} \Longrightarrow$ *incl* ((λ$\mathcal{A}$. *P.Src* (*P.Cong-class-rep* $\mathcal{A}$)) $\mathcal{A}$) = $\mathcal{A}$
using *src-char*$_{CCWE}$ *src-ide* **by** *auto*
show *bij-betw incl* (*Collect R.ide*) (*Collect ide*)
using *incl-injective-on-ide img-incl-ide bij-betw-def* **by** *blast*
show *bij-betw* (λ$\mathcal{A}$. *P.Src* (*P.Cong-class-rep* $\mathcal{A}$)) (*Collect ide*) (*Collect R.ide*)
using *1 2 3 4 bij-betwI* **by** *force*
qed

end

2.5.3 Composite Completion of an Extensional RTS

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

sublocale *composite-completion-of-weakly-extensional-rts* ..
sublocale *incl*: *simulation-between-extensional-rts R resid incl* ..

end

2.5.4 Freeness of Composite Completion

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

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

locale *rts-with-chosen-composites* =
rts +
fixes *comp* :: *'a comp* (**infixr** · *55*)
assumes *comp-extensionality-ax*: $\bigwedge t\ u$ :: *'a*. ¬ *seq t u* ⟹ *t* · *u* = *null*
and *composite-of-comp-ax*: $\bigwedge t\ u\ v$ :: *'a*. *seq t u* ⟹ *composite-of t u* (*t* · *u*)
and *comp-assoc-ax*: $\bigwedge t\ u\ v$ :: *'a*. ⟦*seq t u*; *seq u v*⟧ ⟹ (*t* · *u*) · *v* = *t* · (*u* · *v*)
and *resid-comp-right-ax*: *t* · *u* ⌢ *w* ⟹ *w* \ (*t* · *u*) = (*w* \ *t*) \ *u*
and *resid-comp-left-ax*: (*t* · *u*) \ *w* = (*t* \ *w*) · (*u* \ (*w* \ *t*))
begin

lemma *comp-assoc*$_{CC}$:
shows *t* · *u* · *v* = (*t* · *u*) · *v*
using *comp-extensionality-ax comp-assoc-ax*
by (*metis* (*mono-tags*, *lifting*) *composite-of-comp-ax not-arr-null seq-def*
sources-composite-of targets-composite-of)

lemma *comp-null*$_{CC}$:
shows *t* · *null* = *null* **and** *null* · *t* = *null*
using *comp-extensionality-ax not-arr-null* **apply** *blast*
by (*simp add*: *comp-extensionality-ax seq-def*)

lemma *composable-iff-arr-comp*$_{CC}$:
shows *composable t u* $\longleftrightarrow$ *arr* (*t* · *u*)
by (*metis arr-composite-of comp-extensionality-ax composable-def composable-imp-seq composite-of-comp-ax not-arr-null*)

lemma *composable-iff-comp-not-null*$_{CC}$:
shows *composable t u* $\longleftrightarrow$ *t* · *u* ≠ *null*
by (*metis comp-extensionality-ax composable-def composable-iff-arr-comp*$_{CC}$ *composite-of-comp-ax not-arr-null*)

lemma *con-comp-iff*$_{CC}$:
shows *w* ⌢ *t* · *u* $\longleftrightarrow$ *composable t u* ∧ *w* \ *t* ⌢ *u*
by (*metis composable-iff-comp-not-null*$_{CC}$ *composable-imp-seq composite-of-comp-ax con-composite-of-iff not-con-null*(*2*))

lemma *con-compI*$_{CC}$ [*intro*]:
assumes *composable t u* **and** *w* \ *t* ⌢ *u*
shows *w* ⌢ *t* · *u* **and** *t* · *u* ⌢ *w*
using *assms con-comp-iff*$_{CC}$ *con-sym* **by** *blast*+

sublocale *rts-with-composites resid*
using *composite-of-comp-ax composable-def*
by *unfold-locales auto*

end

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

abbreviation *Comp*
where *Comp T U* ≡ *if seq T U then T* @ *U else null*

sublocale *rts-with-chosen-composites Resid Comp*
proof
show ⋀*T U*. ¬ *seq T U* ⟹ *Comp T U* = *null*
by *argo*
show ⋀*t u v*. *seq t u* ⟹ *composite-of t u* (*Comp t u*)
by (*simp add*: *append-is-composite-of*)
show ⋀*t u v*. ⟦*seq t u*; *seq u v*⟧ ⟹ *Comp* (*Comp t u*) *v* = *Comp t* (*Comp u v*)
by (*simp add*: *seq-def*)
show ⋀*t u w*. *con* (*Comp t u*) *w* ⟹ *w* ** *Comp t u* = (*w* ** *t*) ** *u*
by (*metis* (*full-types*) *Arr.simps*(*1*) *Con-append*(*2*) *con-implies-arr*(*1*) *not-arr-null Resid.simps*(*1*) *Resid-append*(*2*) *paths-in-rts.seq-char paths-in-rts-axioms*)
show ⋀*t u w*. *Comp t u* ** *w* = *Comp* (*t* ** *w*) (*u* ** (*w* ** *t*))
proof −
fix *t u w*
have ⟦*Arr t*; *Arr u*; {*Trg t*} = *Srcs u*; (*t* @ *u*) ** *w* ≠ []⟧

```
            ⟹ Trg (t *\* w) ∈ Srcs (u *\* (w *\* t))
        by (metis Arr.simps(1) Con-imp-Arr-Resid Con-implies-Arr(2) Resid-append-ind
            Srcs-Resid Trgs-Resid-sym Trgs-simp_PWE insertI1)
      thus Comp t u *\* w = Comp (t *\* w) (u *\* (w *\* t))
        using seq-char con-char null-char Con-implies-Arr
        apply auto[1]
        by (metis Arr-has-Src Con-append(1) Resid-append(1) Src-resid Srcs.simps(1)
            Srcs-simp_PWE Trg-resid-sym con-imp-arr-resid singleton-iff Con-imp-eq-Srcs)+

    qed
  qed

  lemma extends-to-rts-with-chosen-composites:
  shows rts-with-chosen-composites Resid Comp
    ..

end

context extensional-rts-with-composites
begin

  lemma extends-to-rts-with-chosen-composites:
  shows rts-with-chosen-composites resid comp
  using composable-iff-comp-not-null comp-is-composite-of(1) comp-assoc_EC
        resid-comp(1−2) comp-null(2) conI con-comp-iff_EC(2) mediating-transition
    apply unfold-locales
      apply auto[4]
    by (metis comp-null(2) composable-imp-seq resid-comp(2))

  sublocale rts-with-chosen-composites resid comp
    using extends-to-rts-with-chosen-composites by blast

end

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

  notation Resid    (infix ‹*\_A*› 70)
  notation Resid1x  (infix ‹1\_A*› 70)
  notation Residx1  (infix ‹*\_A1› 70)
  notation Con      (infix ‹*⌢_A*› 70)
```

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

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

lemma *map-o-incl-eq*:
shows *map* (*incl t*) = *F t*
by (*simp add*: *null-char F.extensionality*)

lemma *extensionality*:
shows ¬ *arr T* ⟹ *map T* = *B.null*
using *F.extensionality arr-char*
by (*metis Arr.simps*(*2*) *map.elims*)

lemma *preserves-comp*:
shows ⟦*T* ≠ []; *U* ≠ []; *Arr* (*T* @ *U*)⟧ ⟹ *map* (*T* @ *U*) = *map T* $\cdot_B$ *map U*
proof (*induct T arbitrary*: *U*)
show ⋀*U*. [] ≠ [] ⟹ *map* ([] @ *U*) = *map* [] $\cdot_B$ *map U*
by *simp*
fix *t* **and** *T U* :: *'a list*
assume *ind*: ⋀*U*. ⟦*T* ≠ []; *U* ≠ []; *Arr* (*T* @ *U*)⟧
⟹ *map* (*T* @ *U*) = *map T* $\cdot_B$ *map U*
assume *U*: *U* ≠ []
assume *Arr*: *Arr* ((*t* # *T*) @ *U*)
hence *1*: *Arr* (*t* # (*T* @ *U*))
by *simp*
have *2*: *Arr* (*t* # *T*)
by (*metis Con-Arr-self Con-append*(*1*) *Con-implies-Arr*(*1*) *Arr U append-is-Nil-conv*
list.distinct(*1*))
show *map* ((*t* # *T*) @ *U*) = *comp*$_B$ (*map* (*t* # *T*)) (*map U*)
proof (*cases T* = [])
show *T* = [] ⟹ *?thesis*
by (*metis* (*full-types*) *1 arr-char U append-Cons append-Nil list.exhaust*
map.simps(*2*) *map.simps*(*3*))
assume *T*: *T* ≠ []
have *map* ((*t* # *T*) @ *U*) = *map* (*t* # (*T* @ *U*))
by *simp*
also have ... = *F t* $\cdot_B$ *map* (*T* @ *U*)
using *T 1*
by (*metis arr-char Nil-is-append-conv list.exhaust map.simps*(*3*))
also have ... = *F t* $\cdot_B$ (*map T* $\cdot_B$ *map U*)
using *ind*
by (*metis 1 Con-Arr-self Con-implies-Arr*(*1*) *Con-rec*(*4*) *T U append-is-Nil-conv*)
also have ... = (*F t* $\cdot_B$ *map T*) $\cdot_B$ *map U*
using *B.comp-assoc*$_{CC}$ **by** *blast*
also have ... = *map* (*t* # *T*) $\cdot_B$ *map U*
using *T 2*

```
      by (metis arr-char list.exhaust map.simps(3))
    finally show map ((t # T) @ U) = map (t # T) ·B map U by simp
  qed
qed

lemma preserves-arr-ind:
shows ⟦arr T; a ∈ Srcs T⟧ ⟹ B.arr (map T) ∧ F a ∈ B.sources (map T)
proof (induct T arbitrary: a)
  show ⋀a. ⟦arr []; a ∈ Srcs []⟧ ⟹ B.arr (map []) ∧ F a ∈ B.sources (map [])
    using arr-char by simp
  fix a t T
  assume a: a ∈ Srcs (t # T)
  assume tT: arr (t # T)
  assume ind: ⋀a. ⟦arr T; a ∈ Srcs T⟧ ⟹ B.arr (map T) ∧ F a ∈ B.sources (map T)
  have 1: a ∈ A.sources t
    using a tT Con-imp-eq-Srcs Con-initial-right Srcs.simps(2) con-char
    by blast
  show B.arr (map (t # T)) ∧ F a ∈ B.sources (map (t # T))
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      using 1arr-char tT by auto
    assume T: T ≠ []
    obtain a' where a': a' ∈ A.targets t
      using tT 1 A.resid-source-in-targets by auto
    have 2: a' ∈ Srcs T
      using a' tT
      by (metis Con-Arr-self A.sources-resid Srcs.simps(2) arr-char T
          Con-imp-eq-Srcs Con-rec(4))
    have B.arr (map (t # T)) ⟷ B.arr (F t ·B map T)
      using tT T by (metis map.simps(3) neq-Nil-conv)
    also have ... ⟷ True
    proof −
      have B.arr (F t ·B map T)
      proof −
        have B.seq (F t) (map T)
        proof
          show B.arr (map T)
            by (metis 2 A.rts-axioms Con-implies-Arr(2) Resid-cons(2)
                T arrE arr-resid ind not-arr-null null-char
                paths-in-rts.arr-char paths-in-rts.intro tT)
          show B.trg (F t) ∼B B.src (map T)
          proof (intro B.coinitial-ide-are-cong)
            show B.ide (B.trg (F t))
              by (meson 1 A.in-sourcesE A.residuation-axioms B.ide-trg
                  F.preserves-reflects-arr residuation.con-implies-arr(1))
            show B.ide (B.src (map T))
              by (simp add: ‹B.arr (map T)›)
            show B.coinitial (B.trg (F t)) (B.src (map T))
              using a' ind extensionality
```

```
            by (metis 1 2 A.con-implies-arr(1) A.in-sourcesE A.in-targetsE
                B.con-sym B.cong-implies-coinitial B.in-sourcesE B.not-arr-null
                B.sources-con-closed B.src-congI F.preserves-con F.preserves-trg
                ‹B.arr (map T)› ‹B.ide (B.trg (F t))›)
          qed
        qed
        thus ?thesis
          using B.composable-iff-arr-comp_CC by blast
      qed
      thus ?thesis by blast
    qed
    finally have B.arr (map (t # T)) by blast
    moreover have F a ∈ B.sources (map (t # T))
    proof −
      have F a ∈ B.sources (F t)
        using 1 tT F.preserves-sources by blast
      moreover have B.sources (F t) = B.sources (map (t # T))
      proof −
        have B.sources (F t) = B.sources (F t ⋅_B map T)
          by (metis B.comp-extensionality-ax B.composite-of-comp-ax
              B.not-arr-null B.sources-composite-of ‹B.arr (F t ⋅_B map T) = True›)
        also have ... = B.sources (map (t # T))
          by (metis T list.exhaust map.simps(3) tT)
        finally show ?thesis by blast
      qed
      ultimately show ?thesis by blast
    qed
    ultimately show ?thesis by blast
  qed
qed

lemma preserves-arr:
shows arr T ⟹ B.arr (map T)
  using preserves-arr-ind arr-char Arr-has-Src by blast

lemma preserves-sources:
assumes arr T and a ∈ Srcs T
shows F a ∈ B.sources (map T)
  using assms preserves-arr-ind by simp

lemma preserves-targets:
shows ⟦arr T; b ∈ Trgs T⟧ ⟹ F b ∈ B.targets (map T)
proof (induct T)
  show ⟦arr []; b ∈ Trgs []⟧ ⟹ F b ∈ B.targets (map [])
    by simp
  fix t T
  assume tT: arr (t # T)
  assume b: b ∈ Trgs (t # T)
  assume ind: ⟦arr T; b ∈ Trgs T⟧ ⟹ F b ∈ B.targets (map T)
```

```
  show F b ∈ B.targets (map (t # T))
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      using tT b arr-char by auto
    assume T: T ≠ []
    show ?thesis
    proof −
      have F b ∈ B.targets (map T)
        by (metis Resid-cons(2) T Trgs.simps(3) arrE b con-char
            con-implies-arr(2) ind neq-Nil-conv tT)
      moreover have B.targets (map T) = B.targets (map (t # T))
      proof −
        have B.targets (map T) = B.targets (F t ·B map T)
          by (metis B.comp-extensionality-ax B.composite-of-comp-ax
              B.not-arr-null B.targets-composite-of T append-Cons append-Nil
              arr-char preserves-arr list.distinct(1) map.simps(2)
              preserves-comp tT)
        also have ... = B.targets (map (t # T))
          by (metis T list.exhaust map.simps(3) tT)
        finally show ?thesis by blast
      qed
      ultimately show ?thesis by blast
    qed
  qed
qed

lemma preserves-Resid1x-ind:
shows t 1\A* U ≠ A.null ⟹ F t ⌢B map U ∧ F (t 1\A* U) = F t \B map U
proof (induct U arbitrary: t)
  show ⋀t. t 1\A* [] ≠ A.null ⟹ F t ⌢B map [] ∧ F (t 1\A* []) = F t \B map []
    by simp
  fix t u U
  assume uU: t 1\A* (u # U) ≠ A.null
  assume ind: ⋀t. t 1\A* U ≠ A.null
                ⟹ F t ⌢B map U ∧ F (t 1\A* U) = F t \B map U
  show F t ⌢B map (u # U) ∧ F (t 1\A* (u # U)) = F t \B map (u # U)
  proof
    show 1: F t ⌢B map (u # U)
    proof (cases U = [])
      show U = [] ⟹ ?thesis
        using Resid1x.simps(2) map.simps(2) F.preserves-con uU by fastforce
      assume U: U ≠ []
      have 3: [t] *\A* [u] ≠ [] ∧ ([t] *\A* [u]) *\A* U ≠ []
        using Con-cons(2) [of [t] U u]
        by (meson Resid1x-as-Resid′ U not-Cons-self2 uU)
      hence 2: F t ⌢B F u ∧ F t \B F u ⌢B map U
        by (metis Con-rec(1) Con-sym Con-sym1 Residx1-as-Resid Resid-rec(1)
            F.preserves-con F.preserves-resid ind)
      moreover have B.seq (F u) (map U)
```

```
      by (metis B.coinitialE B.con-imp-coinitial calculation B.seqI(1)
          B.sources-resid)
    ultimately have F t ⌢_B map ([u] @ U)
      by (metis 3 B.composite-of-comp-ax B.con-composite-of-iff Con-consI(2)
          Con-implies-Arr(2) append-Cons list.simps(3) map.simps(2) preserves-comp
          self-append-conv2)
    thus ?thesis by simp
  qed
  show F (t ^1\_A^* (u # U)) = F t \_B map (u # U)
  proof (cases U = [])
    show U = [] ⟹ ?thesis
      using Resid1x.simps(2) F.preserves-resid map.simps(2) uU by fastforce
    assume U: U ≠ []
    have F (t ^1\_A^* (u # U)) = F ((t \_A u) ^1\_A^* U)
      using Resid1x-as-Resid' Resid-rec(3) U uU by metis
    also have ... = F (t \_A u) \_B map U
      using uU U ind Con-rec(3) Resid1x-as-Resid [of t \_A u U]
      by (metis Resid1x.simps(3) list.exhaust)
    also have ... = (F t \_B F u) \_B map U
      using uU U
      by (metis Resid1x-as-Resid' F.preserves-resid Con-rec(3))
    also have ... = F t \_B (F u ·_B map U)
    proof −
      have F u ·_B map U ⌢_B F t
      proof
        show B.composable (F u) (map U)
          by (metis 1 B.composable-iff-comp-not-null_CC B.not-con-null(2)
              U append.left-neutral append-Cons arr-char extensionality
              map.simps(2) not-Cons-self preserves-comp)
        show F t \_B F u ⌢_B map U
          by (metis B.arr-resid-iff-con Resid1x.simps(3) U calculation
              ind list.exhaust uU)
      qed
      thus ?thesis
        using B.resid-comp-right-ax [of F u map U F t] by argo
    qed
    also have ... = F t \_B map (u # U)
      by (metis Resid1x-as-Resid' con-char U map.simps(3) neq-Nil-conv
          con-implies-arr(2) uU)
    finally show ?thesis by simp
  qed
 qed
qed

lemma preserves-Residx1-ind:
shows U *\_A^1 t ≠ [] ⟹ map U ⌢_B F t ∧ map (U *\_A^1 t) = map U \_B F t
proof (induct U arbitrary: t)
  show ⋀t. [] *\_A^1 t ≠ [] ⟹ map [] ⌢_B F t ∧ map ([] *\_A^1 t) = map [] \_B F t
    by simp
```

```
  fix t u U
  assume ind: ⋀t. U *\_A^1 t ≠ [] ⟹ map U ⌢_B F t ∧ map (U *\_A^1 t) = map U \_B F t
  assume uU: (u # U) *\_A^1 t ≠ []
  show map (u # U) ⌢_B F t ∧ map ((u # U) *\_A^1 t) = map (u # U) \_B F t
  proof (cases U = [])
    show U = [] ⟹ ?thesis
      using Residx1.simps(2) F.preserves-con F.preserves-resid map.simps(2) uU
      by presburger
    assume U: U ≠ []
    show ?thesis
    proof
      show map (u # U) ⌢_B F t
        using uU U Con-sym1 B.con-sym preserves-Resid1x-ind by blast
      show map ((u # U) *\_A^1 t) = map (u # U) \_B F t
      proof −
        have map ((u # U) *\_A^1 t) = map ((u \_A t) # U *\_A^1 (t \_A u))
          using uU U Residx1-as-Resid Resid-rec(2) by fastforce
        also have ... = F (u \_A t) ·_B map (U *\_A^1 (t \_A u))
          by (metis Residx1-as-Resid arr-char U Con-imp-Arr-Resid
              Con-rec(2) Resid-rec(2) list.exhaust map.simps(3) uU)
        also have ... = F (u \_A t) ·_B map U \_B F (t \_A u)
          using uU U ind Con-rec(2) Residx1-as-Resid by force
        also have ... = (F u \_B F t) ·_B map U \_B (F t \_B F u)
          using uU U
          by (metis Con-initial-right Con-rec(1) Con-sym1 Resid1x-as-Resid'
              Residx1-as-Resid F.preserves-resid)
        also have ... = (F u ·_B map U) \_B F t
          using B.resid-comp-left-ax by auto
        also have ... = map (u # U) \_B F t
          by (metis Con-implies-Arr(2) Con-sym Residx1-as-Resid U
              arr-char map.simps(3) neq-Nil-conv uU)
        finally show ?thesis by simp
      qed
    qed
  qed
qed

lemma preserves-resid-ind:
shows con T U ⟹ map T ⌢_B map U ∧ map (T *\_A^* U) = map T \_B map U
proof (induct T arbitrary: U)
  show ⋀U. con [] U ⟹ map [] ⌢_B map U ∧ map ([] *\_A^* U) = map [] \_B map U
    using con-char Resid.simps(1) by blast
  fix t T U
  assume tT: con (t # T) U
  assume ind: ⋀U. con T U ⟹
                map T ⌢_B map U ∧ map (T *\_A^* U) = map T \_B map U
  show map (t # T) ⌢_B map U ∧ map ((t # T) *\_A^* U) = map (t # T) \_B map U
  proof (cases T = [])
    assume T: T = []
```

```
    show ?thesis
      using T tT
      apply simp
      by (metis Resid1x-as-Resid Residx1-as-Resid con-char
          Con-sym Con-sym1 map.simps(2) preserves-Resid1x-ind)
    next
    assume T: T ≠ []
    have 1: map (t # T) = F t ·B map T
      using tT T
      by (metis con-implies-arr(1) list.exhaust map.simps(3))
    show ?thesis
    proof
      show 2: B.con (map (t # T)) (map U)
        using T tT
        by (metis 1 B.composable-iff-comp-not-null_CC B.con-compI_CC(2) B.con-sym
            B.not-arr-null Con-cons(1) Residx1-as-Resid con-char con-implies-arr(1−2)
            preserves-arr ind not-arr-null null-char preserves-Residx1-ind)
      show map ((t # T) *\A* U) = map (t # T) \B map U
      proof −
        have map ((t # T) *\A* U) = map (([t] *\A* U) @ (T *\A* (U *\A* [t])))
          by (metis Resid.simps(1) Resid-cons(1) con-char ex-un-null tT)
        also have ... = map ([t] *\A* U) ·B map (T *\A* (U *\A* [t]))
          by (metis Arr.simps(1) Con-imp-Arr-Resid Con-implies-Arr(2) Con-sym
              Resid-cons(1−2) con-char T preserves-comp tT)
        also have ... = (map [t] \B map U) ·B map (T *\A* (U *\A* [t]))
          by (metis Con-initial-right Con-sym Resid1x-as-Resid
              Residx1-as-Resid con-char Con-sym1 map.simps(2)
              preserves-Resid1x-ind tT)
        also have ... = (map [t] \B map U) ·B (map T \B map (U *\A* [t]))
          using tT T ind
          by (metis Con-cons(1) Con-sym Resid.simps(1) con-char)
        also have ... = (map [t] \B map U) ·B (map T \B (map U \B map [t]))
          using tT T
          by (metis Con-cons(1) Con-sym Resid.simps(2) Residx1-as-Resid
                con-char map.simps(2) preserves-Residx1-ind)
        also have ... = (F t \B map U) ·B (map T \B (map U \B F t))
          using tT T by simp
        also have ... = map (t # T) \B map U
          using 1 B.resid-comp-left-ax by auto
        finally show ?thesis by simp
      qed
    qed
  qed
qed

lemma preserves-con:
assumes con T U
shows map T ⌢B map U
  using assms preserves-resid-ind by simp
```

lemma *preserves-resid*:
assumes *con T U*
shows *map* (T *\$\backslash_A^*$ U) = *map T* $\backslash_B$ *map U*
  using *assms preserves-resid-ind* **by** *simp*

sublocale *simulation Resid B map*
  using *con-char preserves-con preserves-resid extensionality*
  by *unfold-locales auto*

lemma *is-extension*:
shows *map* ∘ *incl* = *F*
  using *map-o-incl-eq* **by** *auto*

lemma *is-universal*:
shows *simulation Resid B map* **and** *map* ∘ *incl* = *F*
and ⋀*F'*. ⟦*simulation Resid B F'*; *F' o incl* = *F*⟧
          ⟹ ∀ *T*. *arr T* ⟶ *B.cong* (*F' T*) (*map T*)
proof −
  show *simulation Resid B map* **and** *map* ∘ *incl* = *F*
    using *map-o-incl-eq simulation-axioms* **by** *auto*
  show ⋀*F'*. ⟦*simulation Resid B F'*; *F' o incl* = *F*⟧ ⟹ ∀ *T*. *arr T* ⟶ *F' T* ∼$_B$ *map T*
  proof (*intro allI impI*)
    fix *F' T*
    assume *F'*: *simulation Resid B F'*
    assume *1*: *F' o incl* = *F*
    interpret *F'*: *simulation Resid B F'*
      using *F'* **by** *simp*
    show *arr T* ⟹ *B.cong* (*F' T*) (*map T*)
    proof (*induct T*)
      show *arr* [] ⟹ *F'* [] ∼$_B$ *map* []
        by (*simp add*: *arr-char F'.extensionality*)
      fix *t T*
      assume *ind*: *arr T* ⟹ *F' T* ∼$_B$ *map T*
      assume *arr*: *arr* (*t* # *T*)
      show *F'* (*t* # *T*) ∼$_B$ *map* (*t* # *T*)
      proof (*cases Arr* (*t* # *T*))
        show ¬ *Arr* (*t* # *T*) ⟹ *?thesis*
          using *arr arr-char* **by** *blast*
        assume *tT*: *Arr* (*t* # *T*)
        show *?thesis*
        proof (*cases T* = [])
          show *2*: *T* = [] ⟹ *?thesis*
            using *F' 1 tT B.prfx-reflexive arr map.simps*(*2*) **by** *force*
          assume *T*: *T* ≠ []
          have *F'* (*t* # *T*) ∼$_B$ *F'* [*t*] ·$_B$ *map T*
          proof −
            have *F'* (*t* # *T*) = *F'* ([*t*] @ *T*)
              by *simp*

```
also have ... ∼B F′ [t] ·B F′ T
proof −
  have composite-of [t] T ([t] @ T)
    using T tT
    by (metis (full-types) Arr.simps(2) Con-Arr-self
        append-is-composite-of Con-implies-Arr(1) Con-imp-eq-Srcs
        Con-rec(4) Resid-rec(1) Srcs-Resid seq-char A.arrI)
  thus ?thesis
    using F′.preserves-composites B.composite-of-comp-ax
    by (meson B.comp-extensionality-ax B.composable-def
        B.composite-of-unq-upto-cong B.composable-iff-comp-not-null_CC)
qed
also have F′ [t] ·B F′ T ∼B F′ [t] ·B map T
proof
  show 0: F′ [t] ·B F′ T ≲B F′ [t] ·B map T
  proof −
    have F′ [t] ·B F′ T ⌢B F′ [t] ·B map T
    proof
      show 1: B.composable (F′ [t]) (F′ T)
        using B.composable-iff-comp-not-null_CC calculation by force
      show (F′ [t] ·B map T) \B F′ [t] ⌢B F′ T
        by (meson 1 B.composableD(1−2) B.composableE B.composable-def
            B.con-compI_CC(1) B.cong-respects-seq B.cong-subst-left(1)
            B.has-composites B.prfx-implies-con B.prfx-reflexive
            B.resid-composite-of(2) B.rts-axioms F′.preserves-reflects-arr
            ind rts.composite-ofE)
    qed
    thus ?thesis
      by (metis B.con-implies-arr(2) B.con-sym B.not-arr-null
          B.prfx-implies-con B.prfx-transitive B.resid-comp-right-ax
          F′.extensionality calculation ind)
  qed
  show F′ [t] ·B map T ≲B F′ [t] ·B F′ T
  proof −
    have 1: F′ [t] ·B map T ⌢B F′ [t] ·B F′ T
    proof
      show B.composable (F′ [t]) (map T)
        using 0 B.composable-iff-comp-not-null_CC
        by force
      show (F′ [t] ·B F′ T) \B F′ [t] ⌢B map T
        by (meson B.con-comp-iff_CC B.prfx-implies-con
            ‹F′ [t] ·B F′ T ≲B F′ [t] ·B map T›)
    qed
    hence (F′ [t] ·B map T) \B (F′ [t] ·B F′ T) =
          ((F′ [t] ·B map T) \B F′ [t]) \B F′ T
      using B.resid-comp-right-ax B.con-sym by blast
    thus ?thesis
      by (metis 1 B.con-arr-self B.con-implies-arr(1) B.cong-reflexive
          B.not-ide-null B.null-is-zero(2) B.prfx-transitive
```

```
                B.resid-comp-right-ax extensionality ind)
            qed
          qed
          finally show ?thesis by blast
        qed
        also have F' [t] ·_B map T = (F' ∘ incl) t ·_B map T
          using tT
          by (simp add: arr-char null-char F'.extensionality)
        also have ... = F t ·_B map T
          using F' 1 by simp
        also have ... = map (t # T)
          using T tT
          by (metis arr-char list.exhaust map.simps(3))
        finally show ?thesis by simp
      qed
    qed
  qed
qed

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 ⋀t u. rts.composable B t u ⟹ 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 −
  interpret A: rts A
    using assms(3) simulation-def simulation-axioms-def by blast
  interpret B: rts-with-chosen-composites B comp_B
    using assms(1) by blast
  interpret C: rts-with-chosen-composites C comp_C
    using assms(2) by blast
  interpret F: simulation A B F
    using assms(3) by blast
  interpret G: simulation B C G
    using assms(4) by blast
  interpret GoF: composite-simulation A B C F G ..
  interpret Ap: paths-in-rts A ..
  interpret Fx: extension-to-paths A B comp_B F ..
  interpret G-o-Fx: composite-simulation Ap.Resid B C Fx.map G ..
  interpret GoF-x: extension-to-paths A C comp_C ‹G ∘ F› ..
  show GoF-x.map = G-o-Fx.map
  proof
    fix T
    show GoF-x.map T = G-o-Fx.map T
```

```
    proof (cases Ap.arr T)
      show ¬ Ap.arr T ⟹ ?thesis
        using G-o-Fx.extensionality GoF-x.extensionality by presburger
      assume T: Ap.arr T
      show ?thesis
      proof (induct T rule: Ap.Arr-induct)
        show Ap.Arr T
          using T Ap.arr-char by simp
        show ⋀t. Ap.Arr [t] ⟹ GoF-x.map [t] = G-o-Fx.map [t]
          by auto
        show ⋀t U. ⟦Ap.Arr (t # U); U ≠ []; GoF-x.map U = G-o-Fx.map U⟧
                   ⟹ GoF-x.map (t # U) = G-o-Fx.map (t # U)
        proof −
          fix t U
          assume t: Ap.Arr (t # U) and U: U ≠ []
          assume ind: GoF-x.map U = G-o-Fx.map U
          show GoF-x.map (t # U) = G-o-Fx.map (t # U)
          proof −
            have GoF-x.map (t # U) = comp_C (GoF-x.map [t]) (GoF-x.map U)
              by (metis GoF-x.preserves-comp U append-Cons append-Nil
                  list.distinct(1) t)
            also have ... = comp_C (G (Fx.map [t])) (G (Fx.map U))
              using ind by simp
            also have ... = G (comp_B (Fx.map [t]) (Fx.map U))
              by (metis B.composable-iff-comp-not-null_CC B.not-arr-null
                  Fx.extension-to-paths-axioms Fx.preserves-comp U append-Cons
                  append-Nil assms(5) extension-to-paths.preserves-arr
                  extension-to-paths-def not-Cons-self2 paths-in-rts.arr-char t)
            also have ... = G (Fx.map ([t] @ U))
              by (metis Fx.preserves-comp U append.left-neutral append-Cons
                  not-Cons-self2 t)
            also have ... = G-o-Fx.map (t # U)
              by simp
            finally show ?thesis by blast
          qed
        qed
      qed
    qed
  qed
qed

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

interpretation *N*: *coherent-normal-sub-rts A ‹Collect A.ide›*
using *A.rts-axioms A.identities-form-coherent-normal-sub-rts* **by** *auto*
sublocale *P*: *paths-in-rts-with-coherent-normal A ‹Collect A.ide›* **..**
sublocale *Q*: *quotient-by-coherent-normal P.Resid ‹Collect P.NPath›* **..**
sublocale *Ac*: *composite-completion A* **..**

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* **..**

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

sublocale *simulation Ac.resid B map*
unfolding *map-def Ac.resid-def*
using *Q.ext-to-quotient-props* [*of B F-ext.map*] *F-ext.simulation-axioms*
F-ext.preserves-ide B.extensional-rts-axioms P.ide-char Ac.P-ide-iff-NPath
by *blast*

lemma *is-simulation*:
shows *simulation Ac.resid B map*
..

lemma *is-extension*:
shows *map ∘ Ac.incl = F*
proof −
have *map ∘ Ac.incl = map ∘ Q.quot ∘ P.incl*
using *Ac.incl-def* **by** *auto*
also have *... = F-ext.map ∘ P.incl*
using *Q.ext-to-quotient-props* [*of B F-ext.map*]
by (*simp add*: *B.extensional-rts-axioms F-ext.simulation-axioms*
Ac.P-ide-iff-NPath P.ide-char map-def)
also have *... = F*
by (*simp add*: *F-ext.is-extension*)
finally show *?thesis* **by** *blast*
qed

lemma *is-universal*:
shows ∃!*F′*. *simulation Ac.resid B F′ ∧ F′ ∘ Ac.incl = F*
proof
show *0*: *simulation Ac.resid B map ∧ map ∘ Ac.incl = F*
using *simulation-axioms is-extension* **by** *auto*
fix *F′*

```
    assume F': simulation Ac.resid B F' ∧ F' ∘ Ac.incl = F
    interpret F': simulation Ac.resid B F'
      using F' by blast
    show F' = map
    proof −
      have F' ∘ Q.quot = F-ext.map
      proof −
        interpret F'-o-quot: simulation P.Resid B ‹F' ∘ Q.quot›
          using F' Q.quotient-is-simulation Ac.resid-def by auto
        interpret incl: simulation A P.Resid P.incl
          using P.incl-is-simulation by blast
        interpret F'-o-quot-o-incl: composite-simulation A P.Resid B P.incl
                                    ‹F' ∘ Q.quot›
          ..
        have (F' ∘ Q.quot) ∘ P.incl = F
          using F' Ac.incl-def by auto
        hence ∀ T. P.arr T ⟶ (F' ∘ Q.quot) T ∼_B F-ext.map T
          using F-ext.is-universal(3) F'-o-quot.simulation-axioms by blast
        hence ∀ T. P.arr T ⟶ (F' ∘ Q.quot) T = F-ext.map T
          using B.cong-char by blast
        thus ?thesis
        proof −
          have ∀ as. (F' ∘ Q.quot) as = F-ext.map as ∨ ¬ P.arr as
            using ‹∀ T. P.arr T ⟶ (F' ∘ Q.quot) T = F-ext.map T› by blast
          then show ?thesis
            using F'-o-quot.extensionality F-ext.extensionality by fastforce
        qed
      qed
      thus ?thesis
        by (metis (no-types, lifting) 0 B.extensional-rts-axioms F'
            F-ext.preserves-ide F-ext.simulation-axioms Ac.P-ide-iff-NPath
            Q.ext-to-quotient-props(2) Q.is-couniversal map-def mem-Collect-eq
            Ac.resid-def)
    qed
  qed

end

context composite-completion
begin

  lemma arrows-factor-as-paths:
  assumes arr 𝒯
  shows ∃ T. P.arr T ∧ extension-to-paths.map R resid comp incl T = 𝒯
  proof −
    interpret inclx: extension-to-paths R resid comp incl ..
    let ?T = P.Cong-class-rep 𝒯
    have P.arr ?T
      by (metis P.Cong-class-memb-is-arr P.rep-in-Cong-class
```

```
        Q.quotient-by-coherent-normal-axioms assms
        quotient-by-coherent-normal.arr-char resid-def)
    moreover have inclx.map ?T = 𝒯
    proof −
      have ⋀T. P.arr T ⟹ inclx.map T = ⦇T⦈
      proof −
        fix T
        show P.arr T ⟹ inclx.map T = ⦇T⦈
        proof (induct T)
          show P.arr [] ⟹ inclx.map [] = Q.quot []
            using P.not-arr-null P.null-char by auto
          fix a U
          assume ind: P.arr U ⟹ inclx.map U = Q.quot U
          assume aU: P.arr (a # U)
          show inclx.map (a # U) = Q.quot (a # U)
            using Q.quotient-is-simulation aU cong-char incl-def
                  inclx.is-universal(3) resid-def
            by force
        qed
      qed
      thus ?thesis
        using Q.arr-char assms calculation resid-def by force
    qed
    ultimately show ?thesis by blast
  qed

end

lemma extension-to-composite-completion-comp:
assumes extensional-rts-with-composites B
and extensional-rts-with-composites C
and simulation A B F and simulation B C G
shows extension-to-composite-completion.map A C (G ∘ F) =
      G ∘ extension-to-composite-completion.map A B F
proof −
  interpret B: extensional-rts-with-composites B
    using assms(1) by blast
  interpret C: extensional-rts-with-composites C
    using assms(2) by blast
  interpret F: simulation A B F
    using assms(3) by blast
  interpret G: simulation B C G
    using assms(4) by blast
  interpret GoF: composite-simulation A B C F G ..
  interpret Ac: composite-completion A ..
  interpret Fc: extension-to-composite-completion A B F ..
  interpret GoFc: extension-to-composite-completion A C GoF.map ..
  show GoFc.map = G ∘ Fc.map
  proof −
```

```
    have G ∘ Fc.map ∘ Ac.incl = GoFc.map ∘ Ac.incl
      using GoFc.is-extension Fc.is-extension comp-assoc by metis
    thus ?thesis
      using GoFc.is-extension GoFc.is-universal GoFc.simulation-axioms
            G.simulation-axioms Fc.simulation-axioms simulation-comp
      by metis
  qed
qed


lemma composite-completion-of-rts:
assumes rts A
shows ∃(A' :: 'a list set resid) I.
          extensional-rts-with-composites A' ∧ simulation A A' I ∧
        (∀ B (J :: 'a ⇒ 'c). extensional-rts-with-composites B ∧ simulation A B J
                              ⟶ (∃!J'. simulation A' B J' ∧ J' o I = J))
proof (intro exI conjI)
  interpret A: rts A
    using assms by auto
  interpret A': composite-completion A ..
  show extensional-rts-with-composites A'.resid
    ..
  show simulation A A'.resid A'.incl
    using A'.incl-is-simulation by simp
  show ∀ B (J :: 'a ⇒ 'c). extensional-rts-with-composites B ∧ simulation A B J
                            ⟶ (∃!J'. simulation A'.resid B J' ∧ J' o A'.incl = J)
  proof (intro allI impI)
    fix B :: 'c resid and J
    assume 1: extensional-rts-with-composites B ∧ simulation A B J
    interpret B: extensional-rts-with-composites B
      using 1 by simp
    interpret J: simulation A B J
      using 1 by simp
    interpret J: extension-to-composite-completion A B J
      ..
    show ∃!J'. simulation A'.resid B J' ∧ J' ∘ A'.incl = J
      using J.is-universal by auto
  qed
qed
```

2.6 Constructions on RTS's

2.6.1 Products of RTS's

```
locale product-rts =
  A: rts A +
  B: rts B
for A :: 'a resid      (infix ‹\A› 70)
and B :: 'b resid      (infix ‹\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*
by *unfold-locales*
(*metis A.con-implies-arr*(*1*−*2*) *A.not-arr-null fst-conv resid-def*)

lemma *is-partial-magma*:
shows *partial-magma resid*
..

lemma *null-char* [*simp*]:
shows *null* = *Null*
by (*metis B.null-is-zero*(*1*) *ex-un-null null-is-zero*(*1*) *resid-def B.conE snd-conv*)

sublocale *residuation resid*
proof
show $\bigwedge t\ u.\ t \setminus u \neq null \Longrightarrow u \setminus t \neq null$
by (*metis A.con-def A.con-sym null-char prod.inject resid-def B.con-sym*)
show $\bigwedge t\ u.\ t \setminus u \neq null \Longrightarrow (t \setminus u) \setminus (t \setminus u) \neq null$
by (*metis* (*no-types*, *lifting*) *A.arrE B.con-def B.con-imp-arr-resid fst-conv null-char*
resid-def A.arr-resid snd-conv)
show $\bigwedge v\ t\ u.\ (v \setminus t) \setminus (u \setminus t) \neq null \Longrightarrow (v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$
proof −
fix *t u v*
assume *1*: $(v \setminus t) \setminus (u \setminus t) \neq null$
have (*fst v* $\backslash_A$ *fst t*) $\backslash_A$ (*fst u* $\backslash_A$ *fst t*) $\neq A.null$
by (*metis 1 A.not-arr-null fst-conv null-char null-is-zero*(*1*−*2*)
resid-def A.arr-resid)
moreover have (*snd v* $\backslash_B$ *snd t*) $\backslash_B$ (*snd u* $\backslash_B$ *snd t*) $\neq B.null$

by (*metis 1 B.not-arr-null snd-conv null-char null-is-zero*(*1*−*2*)
resid-def B.arr-resid)
ultimately show $(v \setminus t) \setminus (u \setminus t) = (v \setminus u) \setminus (t \setminus u)$
using *resid-def null-char A.con-def B.con-def A.cube B.cube*
apply *simp*
by (*metis* (*no-types*, *lifting*) *A.conI A.con-sym-ax A.resid-reflects-con*
B.con-sym-ax B.null-is-zero(*1*))
qed
qed

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

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

lemma *arr-char* [*iff*]:
shows $arr\ t \longleftrightarrow A.arr\ (fst\ t) \wedge B.arr\ (snd\ t)$
by (*metis* (*no-types*, *lifting*) *A.arr-def B.arr-def B.conE null-char resid-def*
arr-def con-def snd-eqD)

lemma *ide-char* [*iff*]:
shows $ide\ t \longleftrightarrow A.ide\ (fst\ t) \wedge B.ide\ (snd\ t)$
by (*metis* (*no-types*, *lifting*) *A.residuation-axioms B.residuation-axioms*
arr-char arr-def fst-conv null-char prod.collapse resid-def residuation.conE
residuation.ide-def residuation.ide-implies-arr residuation-axioms snd-conv)

lemma *con-char* [*iff*]:
shows $t \frown u \longleftrightarrow fst\ t \frown_A fst\ u \wedge snd\ t \frown_B snd\ u$
by (*simp add*: *con-def resid-def B.con-def*)

lemma *trg-char*:
shows $trg\ t = (if\ arr\ t\ then\ (A.trg\ (fst\ t), B.trg\ (snd\ t))\ else\ Null)$
using *A.trg-def B.trg-def resid-def trg-def* **by** *auto*

sublocale *rts resid*
proof
show $\bigwedge t.\ arr\ t \Longrightarrow ide\ (trg\ t)$
by (*simp add*: *trg-char*)
show *1*: $\bigwedge a\ t.\ [\![ide\ a;\ t \frown a]\!] \Longrightarrow t \setminus a = t$
by (*simp add*: *A.resid-arr-ide B.resid-arr-ide resid-def*)
thus $\bigwedge a\ t.\ [\![ide\ a;\ a \frown t]\!] \Longrightarrow ide\ (a \setminus t)$
using *arr-resid cube*
apply (*elim ideE*, *intro ideI*)
apply *auto*
by (*metis 1 conI con-sym-ax ideI null-is-zero*(*2*))
show $\bigwedge t\ u.\ t \frown u \Longrightarrow \exists a.\ ide\ a \wedge a \frown t \wedge a \frown u$
proof −
fix *t u*

```
    assume tu: t ⌢ u
    obtain a1 where a1: a1 ∈ A.sources (fst t) ∩ A.sources (fst u)
      by (meson A.con-imp-common-source all-not-in-conv con-char tu)
    obtain a2 where a2: a2 ∈ B.sources (snd t) ∩ B.sources (snd u)
      by (meson B.con-imp-common-source all-not-in-conv con-char tu)
    have ide (a1, a2) ∧ (a1, a2) ⌢ t ∧ (a1, a2) ⌢ u
      using a1 a2 ide-char con-char
      by (metis A.con-imp-common-source A.in-sourcesE A.sources-eqI
          B.con-imp-common-source B.in-sourcesE B.sources-eqI con-sym
          fst-conv inf-idem snd-conv tu)
    thus ∃a. ide a ∧ a ⌢ t ∧ a ⌢ u by blast
  qed
  show ⋀t u v. ⟦ide (t \ u); u ⌢ v⟧ ⟹ t \ u ⌢ v \ u
  proof −
    fix t u v
    assume tu: ide (t \ u)
    assume uv: u ⌢ v
    have A.ide (fst t \_A fst u) ∧ B.ide (snd t \_B snd u)
      using tu ide-char
      by (metis conI con-char fst-eqD ide-implies-arr not-arr-null resid-def snd-conv)
    moreover have fst u ⌢_A fst v ∧ snd u ⌢_B snd v
      using uv con-char by blast
    ultimately show t \ u ⌢ v \ u
      by (simp add: A.con-target A.con-sym A.prfx-implies-con
          B.con-target B.con-sym B.prfx-implies-con resid-def)
  qed
qed

lemma is-rts:
shows rts resid
  ..

notation prfx    (infix ‹≲› 50)
notation cong    (infix ‹∼› 50)

lemma sources-char:
shows sources t = A.sources (fst t) × B.sources (snd t)
  by force

lemma targets-char:
shows targets t = A.targets (fst t) × B.targets (snd t)
proof
  show targets t ⊆ A.targets (fst t) × B.targets (snd t)
    using targets-def ide-char con-char resid-def trg-char trg-def by auto
  show A.targets (fst t) × B.targets (snd t) ⊆ targets t
  proof
    fix a
    assume a: a ∈ A.targets (fst t) × B.targets (snd t)
    show a ∈ targets t
```

```
    proof
      show ide a
        using a ide-char by auto
      show trg t ⌢ a
        using a trg-char con-char [of trg t a]
        by (metis (no-types, lifting) SigmaE arr-char con-char con-implies-arr(1)
            fst-conv A.in-targetsE B.in-targetsE A.arr-resid-iff-con
            B.arr-resid-iff-con A.trg-def B.trg-def snd-conv)
    qed
  qed
qed

lemma prfx-char:
shows t ≲ u ⟷ fst t ≲_A fst u ∧ snd t ≲_B snd u
  using A.prfx-implies-con B.prfx-implies-con resid-def by auto

lemma cong-char:
shows t ∼ u ⟷ fst t ∼_A fst u ∧ snd t ∼_B snd u
  using prfx-char by auto

lemma join-of-char:
shows join-of t u v ⟷ A.join-of (fst t) (fst u) (fst v) ∧ B.join-of (snd t) (snd u) (snd v)
and joinable t u ⟷ A.joinable (fst t) (fst u) ∧ B.joinable (snd t) (snd u)
proof −
  show ⋀v. join-of t u v ⟷
             A.join-of (fst t) (fst u) (fst v) ∧ B.join-of (snd t) (snd u) (snd v)
  proof
    fix v
    show join-of t u v ⟹
           A.join-of (fst t) (fst u) (fst v) ∧ B.join-of (snd t) (snd u) (snd v)
    proof −
      assume 1: join-of t u v
      have 2: t ⌢ u ∧ t ⌢ v ∧ u ⌢ v ∧ u ⌢ t ∧ v ⌢ t ∧ v ⌢ u
        by (meson 1 bounded-imp-con con-prfx-composite-of(1) join-ofE con-sym)
      show A.join-of (fst t) (fst u) (fst v) ∧ B.join-of (snd t) (snd u) (snd v)
        using 1 2 prfx-char resid-def
        by (elim conjE join-ofE composite-ofE congE conE,
            intro conjI A.join-ofI B.join-ofI A.composite-ofI B.composite-ofI)
           auto
    qed
    show A.join-of (fst t) (fst u) (fst v) ∧ B.join-of (snd t) (snd u) (snd v)
           ⟹ join-of t u v
      using cong-char resid-def
      by (elim conjE A.join-ofE B.join-ofE A.composite-ofE B.composite-ofE,
           intro join-ofI composite-ofI)
         auto
  qed
  thus joinable t u ⟷ A.joinable (fst t) (fst u) ∧ B.joinable (snd t) (snd u)
    using joinable-def A.joinable-def B.joinable-def by simp
```

```
  qed

end

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

  sublocale weakly-extensional-rts resid
  proof
    show ⋀t u. ⟦t ∼ u; ide t; ide u⟧ ⟹ t = u
     by (metis cong-char ide-char prod.exhaust-sel A.weak-extensionality B.weak-extensionality)
  qed

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

  lemma src-char:
  shows src t = (if arr t then (A.src (fst t), B.src (snd t)) else null)
  proof (cases arr t)
    show ¬ arr t ⟹ ?thesis
      using src-def by presburger
    assume t: arr t
    show ?thesis
      using t con-char arr-char
      by (intro src-eqI) auto
  qed

end

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

  sublocale extensional-rts resid
  proof
    show ⋀t u. t ∼ u ⟹ t = u
      by (metis A.extensionality B.extensionality cong-char prod.collapse)
  qed

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

end

Product Simulations

locale *product-simulation* =
  A1: *rts A1* +
  A0: *rts A0* +
  B1: *rts B1* +
  B0: *rts B0* +
  A1xA0: *product-rts A1 A0* +
  B1xB0: *product-rts B1 B0* +
  F1: *simulation A1 B1 F1* +
  F0: *simulation A0 B0 F0*
for *A1* :: *'a1 resid* (**infix** ‹$\backslash_{A1}$› *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* = (λ*a*. *if A1xA0.arr a then* (*F1* (*fst a*), *F0* (*snd a*))
                else (*F1 A1.null*, *F0 A0.null*))

  lemma *map-simp* [*simp*]:
  assumes *A1.arr a1* **and** *A0.arr a0*
  shows *map* (*a1*, *a0*) = (*F1 a1*, *F0 a0*)
    using *assms map-def* **by** *auto*

  sublocale *simulation A1xA0.resid B1xB0.resid map*
  proof
    show ⋀*t*. ¬ *A1xA0.arr t* ⟹ *map t* = *B1xB0.null*
      using *map-def F1.extensionality F0.extensionality* **by** *auto*
    show ⋀*t u*. *A1xA0.con t u* ⟹ *B1xB0.con* (*map t*) (*map u*)
      using *A1xA0.con-char B1xB0.con-char A1.con-implies-arr A0.con-implies-arr* **by** *auto*
    show ⋀*t u*. *A1xA0.con t u* ⟹ *map* (*A1xA0.resid t u*) = *B1xB0.resid* (*map t*) (*map u*)
      using *A1xA0.resid-def B1xB0.resid-def A1.con-implies-arr A0.con-implies-arr*
      by *auto*
  qed

  lemma *is-simulation*:
  shows *simulation A1xA0.resid B1xB0.resid map*
    ..

end

Binary Simulations

locale *binary-simulation* =

```
  A1: rts A1 +
  A0: rts A0 +
  A: product-rts A1 A0 +
  B: rts B +
  simulation A.resid B F
for A1 :: 'a1 resid    (infix ‹\A1› 70)
and A0 :: 'a0 resid    (infix ‹\A0› 70)
and B :: 'b resid      (infix ‹\B› 70)
and F :: 'a1 * 'a0 ⇒ 'b
begin

  lemma fixing-ide-gives-simulation-1:
  assumes A1.ide a1
  shows simulation A0 B (λt0. F (a1, t0))
  proof
    show ⋀t0. ¬ A0.arr t0 ⟹ F (a1, t0) = B.null
      using assms extensionality A.arr-char by simp
    show ⋀t0 u0. A0.con t0 u0 ⟹ B.con (F (a1, t0)) (F (a1, u0))
      using assms A.con-char preserves-con by auto
    show ⋀t0 u0. A0.con t0 u0 ⟹ F (a1, t0 \A0 u0) = F (a1, t0) \B F (a1, u0)
      using assms A.con-char A.resid-def preserves-resid
      by (metis A1.ideE fst-conv snd-conv)
  qed

  lemma fixing-ide-gives-simulation-0:
  assumes A0.ide a0
  shows simulation A1 B (λt1. F (t1, a0))
  proof
    show ⋀t1. ¬ A1.arr t1 ⟹ F (t1, a0) = B.null
      using assms extensionality A.arr-char by simp
    show ⋀t1 u1. A1.con t1 u1 ⟹ B.con (F (t1, a0)) (F (u1, a0))
      using assms A.con-char preserves-con by auto
    show ⋀t1 u1. A1.con t1 u1 ⟹ F (t1 \A1 u1, a0) = F (t1, a0) \B F (u1, a0)
      using assms A.con-char A.resid-def preserves-resid
      by (metis A0.ideE fst-conv snd-conv)
  qed

end
```

2.6.2 Sub-RTS's

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

```
locale sub-rts =
  R: rts R
for R :: 'a resid  (infix ‹\R› 70)
and Arr :: 'a ⇒ bool +
assumes inclusion: Arr t ⟹ R.arr t
```

and *resid-closed*: $[\![Arr\ t;\ Arr\ u;\ R.con\ t\ u]\!] \Longrightarrow Arr\ (t\ \backslash_R\ u)$
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*
using *R.not-con-null*(*2*) *R.null-is-zero*(*1*) *resid-def*
by *unfold-locales metis*

lemma *is-partial-magma*:
shows *partial-magma resid*
..

lemma *null-char*:
shows $null = R.null$
by (*metis R.not-arr-null inclusion null-eqI resid-def*)

sublocale *residuation resid*
using *R.conE R.con-sym R.not-con-null*(*1*) *null-is-zero*(*1*) *resid-def*
apply *unfold-locales*
apply *metis*
apply (*metis R.con-def R.con-imp-arr-resid resid-closed*)
by (*metis* (*no-types*, *lifting*) *R.con-def R.cube resid-closed*)

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

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

lemma *arr-char*:
shows $arr\ t \longleftrightarrow Arr\ t$
by (*metis R.con-arr-self R.con-def R.not-arr-null arrE con-def inclusion*
null-is-zero(*2*) *resid-def residuation.con-implies-arr*(*1*) *residuation-axioms*)

lemma *ide-char*:
shows $ide\ t \longleftrightarrow Arr\ t \wedge R.ide\ t$
by (*metis R.ide-def arrI arr-char con-def ide-def not-arr-null resid-def*)

lemma *con-char*:
shows $con\ t\ u \longleftrightarrow Arr\ t \wedge Arr\ u \wedge R.con\ t\ u$
by (*metis R.conE arr-char con-def not-arr-null null-is-zero*(*1*) *resid-def*)

lemma *trg-char*:
shows *trg* = (λ*t*. *if arr t then R.trg t else null*)
using *arr-char trg-def R.trg-def resid-def* **by** *fastforce*

sublocale *rts resid*
using *arr-char ide-char con-char trg-char resid-def resid-closed inclusion*
apply *unfold-locales*
using *R.prfx-reflexive trg-def* **apply** *force*
apply (*simp add*: *R.resid-arr-ide*)
apply *simp*
apply (*meson R.con-sym R.in-sourcesE enough-sources*)
by (*metis* (*no-types*, *lifting*) *R.con-target arr-resid-iff-con con-sym-ax null-char*)

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

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

lemma *sources-subset*:
shows *sources t* ⊆ {*a*. *Arr t* ∧ *a* ∈ *R.sources t*}
using *con-char ide-char* **by** *fastforce*

lemma *targets-subset*:
shows *targets t* ⊆ {*b*. *Arr t* ∧ *b* ∈ *R.targets t*}
proof
fix *b*
assume *b*: *b* ∈ *targets t*
show *b* ∈ {*b*. *Arr t* ∧ *b* ∈ *R.targets t*}
by (*metis CollectI R.rts-axioms arr-char arr-iff-has-target b con-char*
emptyE ide-char in-targetsE rts.in-targetsI trg-char)
qed

lemma *prfx-char*$_{SRTS}$:
shows *prfx t u* ⟷ *Arr t* ∧ *Arr u* ∧ *R.prfx t u*
using *arr-char con-char ide-char*
by (*metis R.prfx-implies-con prfx-implies-con resid-closed resid-def*)

lemma *cong-char*$_{SRTS}$:
shows *t* ∼ *u* ⟷ *Arr t* ∧ *Arr u* ∧ *t* ∼$_R$ *u*
using *prfx-char*$_{SRTS}$ **by** *force*

lemma *composite-of-char*:
shows *composite-of t u v* ⟷ *Arr t* ∧ *Arr u* ∧ *Arr v* ∧ *R.composite-of t u v*
proof
show *composite-of t u v* ⟹ *Arr t* ∧ *Arr u* ∧ *Arr v* ∧ *R.composite-of t u v*
by (*metis R.composite-of-def R.con-sym composite-ofE con-char prfx-char*$_{SRTS}$

```
      resid-def rts.prfx-implies-con rts-axioms)
  show Arr t ∧ Arr u ∧ Arr v ∧ R.composite-of t u v ⟹ composite-of t u v
    using composite-of-def resid-closed resid-def rts.composite-ofE ide-char
    by fastforce
qed

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

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

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

abbreviation incl
where incl t ≡ if arr t then t else null

sublocale Incl: simulation resid R incl
  using resid-closed resid-def
  by unfold-locales (auto simp add: null-char arr-char con-char)

lemma inclusion-is-simulation:
shows simulation resid R incl
  ..

lemma incl-cancel-left:
assumes transformation X resid F G T and transformation X resid F' G' T'
and incl ∘ T = incl ∘ T'
shows T = T'
proof
  fix x
  interpret T: transformation X resid F G T
    using assms(1) by blast
  interpret T': transformation X resid F' G' T'
    using assms(2) by blast
  show T x = T' x
  proof −
    have T x = (incl ∘ T) x
      using T.extensionality T.A.prfx-reflexive T.respects-cong arr-char prfx-char_SRTS
      by auto
```

```
    also have ... = (incl ∘ T') x
      using assms(3) by auto
    also have ... = T' x
      using T'.extensionality T.A.prfx-reflexive T'.respects-cong arr-char prfx-char_SRTS
      by fastforce
    finally show ?thesis by blast
  qed
qed

lemma incl-reflects-con:
assumes R.con (incl t) (incl u)
shows con t u
  by (metis (full-types) R.not-con-null(1) R.not-con-null(2) arr-char
      assms con-char null-char)

lemma corestriction-of-simulation:
assumes simulation X R F
and ⋀x. residuation.arr X x ⟹ Arr (F x)
shows simulation X resid F and incl ∘ F = F
proof −
  interpret X: rts X
    using assms(1) simulation-def by blast
  interpret F: simulation X R F
    using assms(1) by blast
  interpret F': simulation X resid F
    using assms(2) con-char resid-def F.extensionality null-char
          X.con-implies-arr(1−2)
    by unfold-locales auto
  show 1: simulation X resid F ..
  show incl ∘ F = F
    using F.extensionality null-char by fastforce
qed

lemma corestriction-of-transformation:
assumes simulation X resid F and simulation X resid G
and transformation X R F G T
and ⋀x. residuation.arr X x ⟹ Arr (T x)
shows transformation X resid F G T and incl ∘ T = T
proof −
  interpret X: rts X
    using assms(3) transformation-def by blast
  interpret R: weakly-extensional-rts R
    using assms(3) transformation-def by blast
  interpret S: weakly-extensional-rts resid
    by (simp add: R.weakly-extensional-rts-axioms preserves-weakly-extensional-rts)
  interpret F: simulation X resid F
    using assms(1) transformation-def by blast
  interpret G: simulation X resid G
    using assms(2) transformation-def by blast
```

```
    interpret T: transformation X R F G T
      using assms(3) by blast
    interpret T': transformation X resid F G T
    proof
      show ⋀f. ¬ X.arr f ⟹ T f = null
        by (simp add: T.extensionality null-char)
      show ⋀x x'. ⟦X.ide x; X.cong x x'⟧ ⟹ T x = T x'
        using T.respects-cong-ide by blast
      show ⋀f. X.ide f ⟹ src (T f) = F f
        by (metis F.preserves-ide F.preserves-reflects-arr R.arr-resid-iff-con
            R.arr-src-iff-arr R.ide-implies-arr R.resid-arr-src S.con-imp-eq-src
            S.src-ide T.F.preserves-ide T.preserves-src X.con-implies-arr(2)
            X.ideE arr-char assms(4) con-char)
      show ⋀f. X.ide f ⟹ trg (T f) = G f
        by (simp add: T.preserves-trg arr-char assms(4) trg-char)
      show ⋀a f. a ∈ X.sources f ⟹ T a \ F f = T (X a f)
        by (metis F.preserves-reflects-arr R.residuation-axioms T.naturality1-ax
            X.arr-iff-has-source X.ex-un-null X.ide-implies-arr X.in-sourcesE
            X.not-arr-null X.null-eqI X.source-is-prfx arr-char assms(4) resid-def
            residuation.conI)
      show ⋀a f. a ∈ X.sources f ⟹ F f \ T a = G f
        by (metis F.preserves-reflects-arr R.arr-resid-iff-con
            T.G.preserves-reflects-arr T.naturality2-ax X.in-sourcesE
            X.residuation-axioms arr-char assms(4) resid-def
            residuation.con-implies-arr(1) residuation.ide-implies-arr)
      show ⋀a f. a ∈ X.sources f ⟹ join-of (T a) (F f) (T f)
        by (meson F.preserves-reflects-arr T.naturality3 X.con-implies-arr(1)
            X.ide-implies-arr X.in-sourcesE arr-char assms(4) join-of-char)
    qed
    show 1: transformation X resid F G T ..
    show incl ∘ T = T
      using T.extensionality arr-char assms(4) null-char by fastforce
  qed

end

locale source-replete-sub-rts =
  R: rts R
for R :: 'a resid  (infix \_R 70)
and Arr :: 'a ⇒ bool +
assumes inclusion: Arr t ⟹ R.arr t
and resid-closed: ⟦Arr t; Arr u; R.con t u⟧ ⟹ Arr (t \_R u)
and source-replete: Arr t ⟹ R.sources t ⊆ Collect Arr
begin

  sublocale sub-rts
    using inclusion resid-closed source-replete
    apply unfold-locales
      apply auto[2]
```

by (*metis Collect-mem-eq Collect-mono-iff R.con-imp-common-source R.sources-eqI R.src-in-sources*)

lemma *is-sub-rts*:
shows *sub-rts R Arr*
..

lemma *sources-char*$_{SRTS}$:
shows *sources t* = {*a*. *Arr t* ∧ *a* ∈ *R.sources t*}
using *source-replete sources-subset*
apply *auto*[*1*]
by (*metis Ball-Collect R.in-sourcesE con-char ide-char in-sourcesI*)

lemma *targets-char*$_{SRTS}$:
shows *targets t* = {*b*. *Arr t* ∧ *b* ∈ *R.targets t*}
proof
show *targets t* ⊆ {*b*. *Arr t* ∧ *b* ∈ *R.targets t*}
using *targets-subset* **by** *blast*
show {*b*. *Arr t* ∧ *b* ∈ *R.targets t*} ⊆ *targets t*
proof
fix *b*
assume *b*: *b* ∈ {*b*. *Arr t* ∧ *b* ∈ *R.targets t*}
show *b* ∈ *targets t*
by (*metis* (*no-types*, *lifting*) *R.in-targetsE R.rts-axioms arr-char b con-arr-self mem-Collect-eq rts.in-sourcesI sources-char*$_{SRTS}$ *sources-resid trg-char trg-def trg-in-targets*)
qed
qed

interpretation P_R: *paths-in-rts R*
..
interpretation *P*: *paths-in-rts resid*
..

lemma *path-reflection*:
shows ⟦P_R.*Arr T*; *set T* ⊆ *Collect Arr*⟧ ⟹ *P.Arr T*
proof (*induct T*, *simp*)
fix *t T*
assume *ind*: ⟦P_R.*Arr T*; *set T* ⊆ *Collect Arr*⟧ ⟹ *P.Arr T*
assume *tT*: P_R.*Arr* (*t* # *T*)
assume *set*: *set* (*t* # *T*) ⊆ *Collect Arr*
have *1*: *R.arr t*
using *tT*
by (*metis* P_R.*Arr-imp-arr-hd list.sel*(*1*))
show *P.Arr* (*t* # *T*)
proof (*cases T* = [])
show *T* = [] ⟹ *?thesis*

```
      using 1 set arr-char by simp
    assume T: T ≠ []
    show ?thesis
    proof
      show arr t
        using 1 arr-char set by simp
      show P.Arr T
        using T tT P_R.Arr-imp-Arr-tl
        by (metis ind insert-subset list.sel(3) list.simps(15) set)
      show targets t ⊆ P.Srcs T
      proof −
        have targets t ⊆ R.targets t
          using targets-subset by blast
        also have ... ⊆ R.sources (hd T)
          using T tT
          by (metis P_R.Arr.simps(3) P_R.Srcs-simp_P list.collapse)
        also have ... ⊆ P.Srcs T
          using P.Arr-imp-arr-hd P.Srcs-simp_P ‹P.Arr T› sources-char_SRTS arr-char
          by force
        finally show ?thesis by blast
      qed
    qed
  qed
qed

end

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

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

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

  lemma src-char:
  shows src = (λt. if arr t then R.src t else null)
  proof
    fix t
    show src t = (if arr t then R.src t else null)
      by (metis R.src-eqI con-arr-src(2) con-char ide-char ide-src src-def)
  qed
```

```
  lemma targets-char:
  assumes arr t
  shows targets t = {R.trg t}
    using assms trg-char trg-in-targets arr-has-un-target by auto

end

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

  sublocale sub-rts-of-weakly-extensional-rts ..

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

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

end
```

Here we justify the terminology "normal sub-RTS", which was introduced earlier, by showing that a normal sub-RTS really is a sub-RTS.

```
lemma (in normal-sub-rts) is-sub-rts:
shows source-replete-sub-rts resid (λt. t ∈ 𝔑)
  using elements-are-arr ide-closed
  apply unfold-locales
     apply blast
    apply (meson R.con-def R.con-imp-coinitial R.con-sym-ax forward-stable)
  by blast

end
```

Chapter 3

The Lambda Calculus

In this second part of the article, we apply the residuated transition system framework developed in the first part to the theory of reductions in Church's 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*
using *assms*
by (*metis in-measure wf-iff-no-infinite-down-chain wf-measure*)

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*

by (*induct rule*: *subterm.induct*) *auto*

abbreviation *subterm-rel*
where *subterm-rel* ≡ {(*t*, *u*). *subterm t u*}

lemma *wf-subterm-rel*:
shows *wf subterm-rel*
using *subterm-implies-smaller wf-if-img-lt*
by (*metis case-prod-conv mem-Collect-eq*)

abbreviation *subterm-pair-rel*
where *subterm-pair-rel* ≡ {((*t1*, *t2*), *u1*, *u2*). *subterm t1 u1* ∧ *subterm t2 u2*}

lemma *wf-subterm-pair-rel*:
shows *wf subterm-pair-rel*
using *subterm-implies-smaller*
wf-if-img-lt [*of subterm-pair-rel* λ(*t1*, *t2*). *max* (*size t1*) (*size t2*)]
by *fastforce*

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*
using *subterm-implies-smaller*
wf-if-img-lt [*of subterm-triple-rel*
λ(*t1*, *t2*, *t3*). *max* (*max* (*size t1*) (*size t2*)) (*size t3*)]
by *fastforce*

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*)
by (*metis subterm.simps*)+

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 ≠ ♯
  using assms by auto

fun Ide
where Ide ♯ = False
  | Ide «-» = True
  | Ide λ[t] = Ide t
  | Ide (t ∘ u) = (Ide t ∧ Ide u)
  | Ide (λ[t] • u) = False

lemma Ide-implies-Arr:
shows Ide t ⟹ Arr t
  by (induct t) auto

lemma ArrE [elim]:
assumes Arr t
and ⋀i. t = «i» ⟹ T
and ⋀u. t = λ[u] ⟹ T
and ⋀u v. t = u ∘ v ⟹ T
and ⋀u v. t = λ[u] • v ⟹ T
shows T
  using assms
  by (cases t) auto
```

3.1.3 Raising Indices

For substitution, we need to be able to raise the indices of all free variables in a subterm by a specified amount. To do this recursively, we need to keep track of the depth of nesting of 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 - - ♯ = ♯
  | Raise d n «i» = (if i ≥ d then «i+n» else «i»)
  | Raise d n λ[t] = λ[Raise (Suc d) n t]
  | Raise d n (t ∘ u) = Raise d n t ∘ Raise d n u
  | Raise d n (λ[t] • u) = λ[Raise (Suc d) n t] • Raise d n u
```

Ultimately, the definition of substitution will only directly involve the function that raises all indices of variables that are free in the outermost context; in a term, so we introduce an abbreviation for this special case.

abbreviation *raise*
where *raise* == *Raise 0*

lemma *size-Raise*:
shows $\bigwedge d.\ size\ (Raise\ d\ n\ t) = size\ t$
by (*induct t*) *auto*

lemma *Raise-not-Nil*:
assumes $t \neq \sharp$
shows $Raise\ d\ n\ t \neq \sharp$
using *assms*
by (*cases t*) *auto*

lemma *FV-Raise*:
shows $FV\ (Raise\ d\ n\ t) = (\lambda x.\ if\ x \geq d\ then\ x + n\ else\ x)\ `\ FV\ t$
apply (*induct t arbitrary*: *d n*)
apply *auto*[3]
apply *force*
apply *force*
apply *force*
apply *force*
apply *force*
proof −
fix *t u d n*
assume *ind1*: $\bigwedge d\ n.\ FV\ (Raise\ d\ n\ t) = (\lambda x.\ if\ d \leq x\ then\ x + n\ else\ x)\ `\ FV\ t$
assume *ind2*: $\bigwedge d\ n.\ FV\ (Raise\ d\ n\ u) = (\lambda x.\ if\ d \leq x\ then\ x + n\ else\ x)\ `\ FV\ u$
have $FV\ (Raise\ d\ n\ (\boldsymbol{\lambda}[t] \bullet u)) =$
$(\lambda x.\ x - Suc\ 0)\ `\ ((\lambda x.\ x + n)\ `$
$(FV\ t \cap \{x.\ Suc\ d \leq x\}) \cup FV\ t \cap \{x.\ \neg\ Suc\ d \leq x\} - \{0\}) \cup$
$((\lambda x.\ x + n)\ `\ (FV\ u \cap \{x.\ d \leq x\}) \cup FV\ u \cap \{x.\ \neg\ d \leq x\})$
using *ind1 ind2* **by** *simp*
also have ... $= (\lambda x.\ if\ d \leq x\ then\ x + n\ else\ x)\ `\ FV\ (\boldsymbol{\lambda}[t] \bullet u)$
by *auto force*+
finally show $FV\ (Raise\ d\ n\ (\boldsymbol{\lambda}[t] \bullet u)) =$
$(\lambda x.\ if\ d \leq x\ then\ x + n\ else\ x)\ `\ FV\ (\boldsymbol{\lambda}[t] \bullet u)$
by *blast*
qed

lemma *Arr-Raise*:
shows $Arr\ t \longleftrightarrow Arr\ (Raise\ d\ n\ t)$
using *FV-Raise*
by (*induct t arbitrary*: *d n*) *auto*

lemma *Ide-Raise*:
shows $Ide\ t \longleftrightarrow Ide\ (Raise\ d\ n\ t)$
by (*induct t arbitrary*: *d n*) *auto*

lemma *Raise-0*:
shows $Raise\ d\ 0\ t = t$

by (*induct t arbitrary*: *d*) *auto*

lemma *Raise-Suc*:
shows *Raise d* (*Suc n*) *t* = *Raise d 1* (*Raise d n t*)
by (*induct t arbitrary*: *d n*) *auto*

lemma *Raise-Var*:
shows *Raise d n «i»* = *«if i* < *d then i else i* + *n»*
by *auto*

The following development of the properties of raising indices, substitution, and residuation has benefited greatly from the previous work by Huet [7]. In particular, it was very helpful to have correct statements of various lemmas available, rather than having to reconstruct them.

lemma *Raise-plus*:
shows *Raise d* (*m* + *n*) *t* = *Raise* (*d* + *m*) *n* (*Raise d m t*)
by (*induct t arbitrary*: *d m n*) *auto*

lemma *Raise-plus′*:
shows ⟦*d′* ≤ *d* + *n*; *d* ≤ *d′*⟧ ⟹ *Raise d* (*m* + *n*) *t* = *Raise d′ m* (*Raise d n t*)
by (*induct t arbitrary*: *n m d d′*) *auto*

lemma *Raise-Raise*:
shows *i* ≤ *n* ⟹ *Raise i p* (*Raise n k t*) = *Raise* (*p* + *n*) *k* (*Raise i p t*)
by (*induct t arbitrary*: *i k n p*) *auto*

lemma *raise-plus*:
shows *d* ≤ *n* ⟹ *raise* (*m* + *n*) *t* = *Raise d m* (*raise n t*)
using *Raise-plus′* **by** *auto*

lemma *raise-Raise*:
shows *raise p* (*Raise n k t*) = *Raise* (*p* + *n*) *k* (*raise p t*)
by (*simp add*: *Raise-Raise*)

lemma *Raise-inj*:
shows *Raise d n t* = *Raise d n u* ⟹ *t* = *u*
proof (*induct t arbitrary*: *d n u*)
show ⋀*d n u*. *Raise d n ♯* = *Raise d n u* ⟹ *♯* = *u*
by (*metis Raise.simps*(*1*) *Raise-not-Nil*)
show ⋀*x d n*. *Raise d n «x»* = *Raise d n u* ⟹ *«x»* = *u* **for** *u*
using *Raise-Var*
apply (*cases u*, *auto*)
by (*metis add-lessD1 add-right-imp-eq*)
show ⋀*t d n*. ⟦⋀*d n u′*. *Raise d n t* = *Raise d n u′* ⟹ *t* = *u′*;
Raise d n **λ**[*t*] = *Raise d n u*⟧
⟹ **λ**[*t*] = *u*
for *u*
apply (*cases u*, *auto*)
by (*metis lambda.distinct*(*9*))

```
  show ⋀t1 t2 d n. ⟦⋀d n u'. Raise d n t1 = Raise d n u' ⟹ t1 = u';
                    ⋀d n u'. Raise d n t2 = Raise d n u' ⟹ t2 = u';
                    Raise d n (t1 ∘ t2) = Raise d n u⟧
                      ⟹ t1 ∘ t2 = u
    for u
    apply (cases u, auto)
    by (metis lambda.distinct(11))
  show ⋀t1 t2 d n. ⟦⋀d n u'. Raise d n t1 = Raise d n u' ⟹ t1 = u';
                    ⋀d n u'. Raise d n t2 = Raise d n u' ⟹ t2 = u';
                    Raise d n (λ[t1] • t2) = Raise d n u⟧
                      ⟹ λ[t1] • t2 = u
    for u
    apply (cases u, auto)
    by (metis lambda.distinct(13))
qed
```

3.1.4 Substitution

Following [7], we now define a generalized substitution operation with adjustment of indices. The ultimate goal is to define the result of contraction of a marked redex *Beta t u* to be *subst u t*. However, to be able to give a proper recursive definition of *subst*, we need to introduce a parameter *n* to keep track of the depth of nesting of *Lam*'s as we descend into the the term *t*. So, instead of *subst u t* simply substituting *u* for occurrences of *Var 0*, *Subst n u t* will be substituting for occurrences of *Var n*, and the term *u* will have the indices of its free variables raised by *n* before replacing *Var n*. In addition, any variables in *t* that have indices greater than *n* will have these indices lowered by one, to account for the outermost *Lam* that is being removed by the contraction. We can then define *subst u t* to be *Subst 0 u t*.

```
fun Subst
where Subst - - ♯ = ♯
    | Subst n v «i» = (if n < i then «i−1» else if n = i then raise n v else «i»)
    | Subst n v λ[t] = λ[Subst (Suc n) v t]
    | Subst n v (t ∘ u) = Subst n v t ∘ Subst n v u
    | Subst n v (λ[t] • u) = λ[Subst (Suc n) v t] • Subst n v u

abbreviation subst
where subst ≡ Subst 0

lemma Subst-Nil:
shows Subst n v ♯ = ♯
  by (cases v = ♯) auto

lemma Subst-not-Nil:
assumes v ≠ ♯ and t ≠ ♯
shows t ≠ ♯ ⟹ Subst n v t ≠ ♯
  using assms Raise-not-Nil
  by (induct t) auto
```

The following expression summarizes how the set of free variables of a term *Subst d u t*, obtained by substituting *u* into *t* at depth *d*, relates to the sets of free variables of *t* and *u*. This expression is not used in the subsequent formal development, but it has been left here as an aid to understanding.

abbreviation *FVS*
where $FVS\ d\ v\ t \equiv (FV\ t \cap \{x.\ x < d\}) \cup$
$(\lambda x.\ x - 1)\ `\ \{x.\ x > d \wedge x \in FV\ t\} \cup$
$(\lambda x.\ x + d)\ `\ \{x.\ d \in FV\ t \wedge x \in FV\ v\}$

lemma *FV-Subst*:
shows $FV\ (Subst\ d\ v\ t) = FVS\ d\ v\ t$
proof (*induct t arbitrary*: *d v*)
have $\bigwedge d\ t\ v.\ (\lambda x.\ x - 1)\ `\ (FVS\ (Suc\ d)\ v\ t - \{0\}) = FVS\ d\ v\ \boldsymbol{\lambda}[t]$
proof −
fix *d t v*
have $FVS\ d\ v\ \boldsymbol{\lambda}[t] =$
$(\lambda x.\ x - Suc\ 0)\ `\ (FV\ t - \{0\}) \cap \{x.\ x < d\} \cup$
$(\lambda x.\ x - Suc\ 0)\ `\ \{x.\ d < x \wedge x \in (\lambda x.\ x - Suc\ 0)\ `\ (FV\ t - \{0\})\} \cup$
$(\lambda x.\ x + d)\ `\ \{x.\ d \in (\lambda x.\ x - Suc\ 0)\ `\ (FV\ t - \{0\}) \wedge x \in FV\ v\}$
by *simp*
also have $... = (\lambda x.\ x - 1)\ `\ (FVS\ (Suc\ d)\ v\ t - \{0\})$
by *auto force+*
finally show $(\lambda x.\ x - 1)\ `\ (FVS\ (Suc\ d)\ v\ t - \{0\}) = FVS\ d\ v\ \boldsymbol{\lambda}[t]$
by *metis*
qed
thus $\bigwedge d\ t\ v.\ (\bigwedge d\ v.\ FV\ (Subst\ d\ v\ t) = FVS\ d\ v\ t)$
$\Longrightarrow FV\ (Subst\ d\ v\ \boldsymbol{\lambda}[t]) = FVS\ d\ v\ \boldsymbol{\lambda}[t]$
by *simp*
have $\bigwedge t\ u\ v\ d.\ (\lambda x.\ x - 1)\ `\ (FVS\ (Suc\ d)\ v\ t - \{0\}) \cup FVS\ d\ v\ u = FVS\ d\ v\ (\boldsymbol{\lambda}[t] \bullet u)$
proof −
fix *t u v d*
have $FVS\ d\ v\ (\boldsymbol{\lambda}[t] \bullet u) =$
$((\lambda x.\ x - Suc\ 0)\ `\ (FV\ t - \{0\}) \cup FV\ u) \cap \{x.\ x < d\} \cup$
$(\lambda x.\ x - Suc\ 0)\ `\ \{x.\ d < x \wedge (x \in (\lambda x.\ x - Suc\ 0)\ `\ (FV\ t - \{0\}) \vee x \in FV\ u)\} \cup$
$(\lambda x.\ x + d)\ `\ \{x.\ (d \in (\lambda x.\ x - Suc\ 0)\ `\ (FV\ t - \{0\}) \vee d \in FV\ u) \wedge x \in FV\ v\}$
by *simp*
also have $... = (\lambda x.\ x - 1)\ `\ (FVS\ (Suc\ d)\ v\ t - \{0\}) \cup FVS\ d\ v\ u$
by *force*
finally show $(\lambda x.\ x - 1)\ `\ (FVS\ (Suc\ d)\ v\ t - \{0\}) \cup FVS\ d\ v\ u = FVS\ d\ v\ (\boldsymbol{\lambda}[t] \bullet u)$
by *metis*
qed
thus $\bigwedge t\ u\ v\ d.\ [\![\bigwedge d\ v.\ FV\ (Subst\ d\ v\ t) = FVS\ d\ v\ t;$
$\bigwedge d\ v.\ FV\ (Subst\ d\ v\ u) = FVS\ d\ v\ u]\!]$
$\Longrightarrow FV\ (Subst\ d\ v\ (\boldsymbol{\lambda}[t] \bullet u)) = FVS\ d\ v\ (\boldsymbol{\lambda}[t] \bullet u)$
by *simp*
qed (*auto simp add*: *FV-Raise*)

lemma *Arr-Subst*:
assumes *Arr v*

shows *Arr t* $\Longrightarrow$ *Arr* (*Subst n v t*)
using *assms Arr-Raise FV-Subst*
by (*induct t arbitrary*: *n*) *auto*

lemma *vacuous-Subst*:
shows $[\![$*Arr v*; *i* $\notin$ *FV t*$]\!]$ $\Longrightarrow$ *Raise i 1* (*Subst i v t*) = *t*
apply (*induct t arbitrary*: *i v*, *auto*)
by *force*+

lemma *Ide-Subst-iff*:
shows *Ide* (*Subst n v t*) $\longleftrightarrow$ *Ide t* $\wedge$ (*n* $\in$ *FV t* $\longrightarrow$ *Ide v*)
using *Ide-Raise vacuous-Subst*
apply (*induct t arbitrary*: *n*)
apply *auto*[5]
apply *fastforce*
by (*metis Diff-empty Diff-insert0 One-nat-def diff-Suc-1 image-iff insertE*
insert-Diff nat.distinct(*1*))

lemma *Ide-Subst*:
shows $[\![$*Ide t*; *Ide v*$]\!]$ $\Longrightarrow$ *Ide* (*Subst n v t*)
using *Ide-Raise*
by (*induct t arbitrary*: *n*) *auto*

lemma *Raise-Subst*:
shows *Raise* (*p* + *n*) *k* (*Subst p v t*) = *Subst p* (*Raise n k v*) (*Raise* (*Suc* (*p* + *n*)) *k t*)
using *raise-Raise*
apply (*induct t arbitrary*: *v k n p*, *auto*)
by (*metis add-Suc*)+

lemma *Raise-Subst'*:
assumes *t* $\neq$ $\sharp$
shows $[\![$*v* $\neq$ $\sharp$; *k* $\leq$ *n*$]\!]$ $\Longrightarrow$ *Raise k p* (*Subst n v t*) = *Subst* (*p* + *n*) *v* (*Raise k p t*)
using *assms raise-plus*
apply (*induct t arbitrary*: *v k n p*, *auto*)
apply (*metis Raise.simps*(*1*) *Subst-Nil Suc-le-mono add-Suc-right*)
apply *fastforce*
apply *fastforce*
apply (*metis Raise.simps*(*1*) *Subst-Nil Suc-le-mono add-Suc-right*)
by *fastforce*

lemma *Raise-subst*:
shows *Raise n k* (*subst v t*) = *subst* (*Raise n k v*) (*Raise* (*Suc n*) *k t*)
using *Raise-0*
apply (*induct t arbitrary*: *v k n*, *auto*)
by (*metis One-nat-def Raise-Subst plus-1-eq-Suc*)+

lemma *raise-Subst*:
assumes *t* $\neq$ $\sharp$
shows *v* $\neq$ $\sharp$ $\Longrightarrow$ *raise p* (*Subst n v t*) = *Subst* (*p* + *n*) *v* (*raise p t*)

using *assms Raise-plus raise-Raise Raise-Subst′*
apply (*induct t arbitrary*: *v n p*)
by (*meson zero-le*)+

lemma *Subst-Raise*:
shows ⟦$v \neq \sharp$; $d \leq m$; $m \leq n + d$⟧ ⟹ *Subst m v* (*Raise d* (*Suc n*) *t*) = *Raise d n t*
by (*induct t arbitrary*: *v m n d*) *auto*

lemma *Subst-raise*:
shows ⟦$v \neq \sharp$; $m \leq n$⟧ ⟹ *Subst m v* (*raise* (*Suc n*) *t*) = *raise n t*
using *Subst-Raise*
by (*induct t arbitrary*: *v m n*) *auto*

lemma *Subst-Subst*:
shows ⟦$v \neq \sharp$; $w \neq \sharp$⟧ ⟹
Subst (*m* + *n*) *w* (*Subst m v t*) = *Subst m* (*Subst n w v*) (*Subst* (*Suc* (*m* + *n*)) *w t*)
using *Raise-0 raise-Subst Subst-not-Nil Subst-raise*
apply (*induct t arbitrary*: *v w m n*, *auto*)
by (*metis add-Suc*)+

The Substitution Lemma, as given by Huet [7].

lemma *substitution-lemma*:
shows ⟦$v \neq \sharp$; $w \neq \sharp$⟧ ⟹ *Subst n v* (*subst w t*) = *subst* (*Subst n v w*) (*Subst* (*Suc n*) *v t*)
by (*metis Subst-Subst add-0*)

3.2 Lambda-Calculus as an RTS

3.2.1 Residuation

We now define residuation on terms. Residuation is an operation which, when defined for terms t and u, produces terms $t \setminus u$ and $u \setminus t$ that represent, respectively, what remains of the reductions of t after performing the reductions in u, and what remains of the reductions of u after performing the reductions in t.

The definition ensures that, if residuation is defined for two terms, then those terms in must be arrows that are *coinitial* (*i.e.* they are the same after erasing marks on redexes). The residual $t \setminus u$ then has marked redexes at positions corresponding to redexes that were originally marked in t and that were not contracted by any of the reductions of u.

This definition has also benefited from the presentation in [7].

fun *resid* (**infix** ‹\› *70*)
where *«i»* \ *«i′»* = (*if i* = *i′ then «i» else* ♯)
| **λ**[*t*] \ **λ**[*t′*] = (*if t* \ *t′* = ♯ *then* ♯ *else* **λ**[*t* \ *t′*])
| (*t* ∘ *u*) \ (*t′*∘ *u′*) = (*if t* \ *t′* = ♯ ∨ *u* \ *u′* = ♯ *then* ♯ *else* (*t* \ *t′*) ∘ (*u* \ *u′*))
| (**λ**[*t*] • *u*) \ (**λ**[*t′*] • *u′*) = (*if t* \ *t′* = ♯ ∨ *u* \ *u′* = ♯ *then* ♯ *else subst* (*u* \ *u′*) (*t* \ *t′*))
| (**λ**[*t*] ∘ *u*) \ (**λ**[*t′*] • *u′*) = (*if t* \ *t′* = ♯ ∨ *u* \ *u′* = ♯ *then* ♯ *else subst* (*u* \ *u′*) (*t* \ *t′*))
| (**λ**[*t*] • *u*) \ (**λ**[*t′*] ∘ *u′*) = (*if t* \ *t′* = ♯ ∨ *u* \ *u′* = ♯ *then* ♯ *else* **λ**[*t* \ *t′*] • (*u* \ *u′*))
| *resid* - - = ♯

Terms t and u are *consistent* if residuation is defined for them.

abbreviation *Con* (**infix** ‹⌢› *50*)
where *Con t u* ≡ *resid t u* ≠ ♯

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 $\bigwedge u\ v\ u'\ v'.\ [\![t = u \circ v;\ t' = u' \circ v';\ u \frown u';\ v \frown v';$
$t \setminus t' = (u \setminus u') \circ (v \setminus v')]\!] \Longrightarrow T$
and $\bigwedge u\ v\ u'\ v'.\ [\![t = \boldsymbol{\lambda}[u] \bullet v;\ t' = \boldsymbol{\lambda}[u'] \bullet v';\ u \frown u';\ v \frown v';$
$t \setminus t' = subst\ (v \setminus v')\ (u \setminus u')]\!] \Longrightarrow T$
and $\bigwedge u\ v\ u'\ v'.\ [\![t = \boldsymbol{\lambda}[u] \circ v;\ t' = Beta\ u'\ v';\ u \frown u';\ v \frown v';$
$t \setminus t' = subst\ (v \setminus v')\ (u \setminus u')]\!] \Longrightarrow T$
and $\bigwedge u\ v\ u'\ v'.\ [\![t = \boldsymbol{\lambda}[u] \bullet v;\ t' = \boldsymbol{\lambda}[u'] \circ v';\ u \frown u';\ v \frown v';$
$t \setminus t' = \boldsymbol{\lambda}[u \setminus u'] \bullet (v \setminus v')]\!] \Longrightarrow T$
shows *T*
using *assms*
apply (*cases t*; *cases t′*)
apply *simp-all*
apply *metis*
apply *metis*
apply *metis*
apply (*cases un-App1 t*, *simp-all*)
apply *metis*
apply (*cases un-App1 t′*, *simp-all*)
apply *metis*
by *metis*

A term can only be consistent with another if both terms are "arrows".

lemma *Con-implies-Arr1*:
shows $t \frown u \Longrightarrow Arr\ t$
proof (*induct t arbitrary*: *u*)
fix *u v t′*
assume *ind1*: $\bigwedge u'.\ u \frown u' \Longrightarrow Arr\ u$
assume *ind2*: $\bigwedge v'.\ v \frown v' \Longrightarrow Arr\ v$
show $u \circ v \frown t' \Longrightarrow Arr\ (u \circ v)$
using *ind1 ind2*
apply (*cases t′*, *simp-all*)
apply *metis*
apply (*cases u*, *simp-all*)
by (*metis lambda.distinct*(*3*) *resid.simps*(*2*))
show $\boldsymbol{\lambda}[u] \bullet v \frown t' \Longrightarrow Arr\ (\boldsymbol{\lambda}[u] \bullet v)$
using *ind1 ind2*
apply (*cases t′*, *simp-all*)
apply (*cases un-App1 t′*, *simp-all*)
by *metis+*
qed *auto*

lemma *Con-implies-Arr2*:

shows $t \frown u \Longrightarrow Arr\ u$
proof (*induct u arbitrary*: *t*)
fix $u'\ v'\ t$
assume *ind1*: $\bigwedge u.\ u \frown u' \Longrightarrow Arr\ u'$
assume *ind2*: $\bigwedge v.\ v \frown v' \Longrightarrow Arr\ v'$
show $t \frown u' \circ v' \Longrightarrow Arr\ (u' \circ v')$
using *ind1 ind2*
apply (*cases t*, *simp-all*)
apply *metis*
apply (*cases u'*, *simp-all*)
by (*metis lambda.distinct*(*3*) *resid.simps*(*2*))
show $t \frown (\boldsymbol{\lambda}[u'] \bullet v') \Longrightarrow Arr\ (\boldsymbol{\lambda}[u'] \bullet v')$
using *ind1 ind2*
apply (*cases t*, *simp-all*)
apply (*cases un-App1 t*, *simp-all*)
by *metis+*
qed *auto*

lemma *ConD*:
shows $t \circ u \frown t' \circ u' \Longrightarrow t \frown t' \wedge u \frown u'$
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 $t \circ u \frown \boldsymbol{\lambda}[v'] \bullet u' \Longrightarrow t \frown \boldsymbol{\lambda}[v'] \wedge u \frown u'$
by *auto*

Residuation on consistent terms preserves arrows.

lemma *Arr-resid*:
shows $t \frown u \Longrightarrow Arr\ (t \setminus u)$
proof (*induct t arbitrary*: *u*)
fix *t1 t2 u*
assume *ind1*: $\bigwedge u.\ t1 \frown u \Longrightarrow Arr\ (t1 \setminus u)$
assume *ind2*: $\bigwedge u.\ t2 \frown u \Longrightarrow Arr\ (t2 \setminus u)$
show $t1 \circ t2 \frown u \Longrightarrow Arr\ ((t1 \circ t2) \setminus u)$
using *ind1 ind2 Arr-Subst*
apply (*cases u*, *auto*)
apply (*cases t1*, *auto*)
by (*metis Arr.simps*(*3*) *ConD*(*2*) *resid.simps*(*2*) *resid.simps*(*4*))
show $\boldsymbol{\lambda}[t1] \bullet t2 \frown u \Longrightarrow Arr\ ((\boldsymbol{\lambda}[t1] \bullet t2) \setminus u)$
using *ind1 ind2 Arr-Subst*
by (*cases u*) *auto*
qed *auto*

3.2.2 Source and Target

Here we give syntactic versions of the *source* and *target* of a term. These will later be shown to agree (on arrows) with the versions derived from the residuation. The underlying idea here is that a term stands for a reduction sequence in which all marked redexes (corresponding to instances of the constructor *Beta*) are contracted in a bottom-up fashion. A term without any marked redexes stands for an empty reduction sequence;

such terms will be shown to be the identities derived from the residuation. The source of term is the identity obtained by erasing all markings; that is, by replacing all subterms of the form *Beta t u* by *App* (*Lam t*) *u*. The target of a term is the identity that is the result of contracting all the marked redexes.

fun *Src*
where *Src* ♯ = ♯
| *Src «i»* = *«i»*
| *Src* **λ**[*t*] = **λ**[*Src t*]
| *Src* (*t* ∘ *u*) = *Src t* ∘ *Src u*
| *Src* (**λ**[*t*] • *u*) = **λ**[*Src t*] ∘ *Src u*

fun *Trg*
where *Trg «i»* = *«i»*
| *Trg* **λ**[*t*] = **λ**[*Trg t*]
| *Trg* (*t* ∘ *u*) = *Trg t* ∘ *Trg u*
| *Trg* (**λ**[*t*] • *u*) = *subst* (*Trg u*) (*Trg t*)
| *Trg* - = ♯

lemma *Ide-Src*:
shows *Arr t* ⟹ *Ide* (*Src t*)
by (*induct t*) *auto*

lemma *Ide-iff-Src-self*:
assumes *Arr t*
shows *Ide t* ⟷ *Src t* = *t*
using *assms Ide-Src*
by (*induct t*) *auto*

lemma *Arr-Src* [*simp*]:
assumes *Arr t*
shows *Arr* (*Src t*)
using *assms Ide-Src Ide-implies-Arr* **by** *blast*

lemma *Con-Src*:
shows ⟦*size t* + *size u* ≤ *n*; *t* ⌢ *u*⟧ ⟹ *Src t* ⌢ *Src u*
by (*induct n arbitrary*: *t u*) *auto*

lemma *Src-eq-iff*:
shows *Src «i»* = *Src «i′»* ⟷ *i* = *i′*
and *Src* (*t* ∘ *u*) = *Src* (*t′* ∘ *u′*) ⟷ *Src t* = *Src t′* ∧ *Src u* = *Src u′*
and *Src* (**λ**[*t*] • *u*) = *Src* (**λ**[*t′*] • *u′*) ⟷ *Src t* = *Src t′* ∧ *Src u* = *Src u′*
and *Src* (**λ**[*t*] ∘ *u*) = *Src* (**λ**[*t′*] • *u′*) ⟷ *Src t* = *Src t′* ∧ *Src u* = *Src u′*
by *auto*

lemma *Src-Raise*:
shows *Src* (*Raise d n t*) = *Raise d n* (*Src t*)
by (*induct t arbitrary*: *d*) *auto*

lemma *Src-Subst* [*simp*]:

shows $[\![Arr\ t;\ Arr\ u]\!] \Longrightarrow Src\ (Subst\ d\ t\ u) = Subst\ d\ (Src\ t)\ (Src\ u)$
using *Src-Raise*
by (*induct u arbitrary*: *d X*) *auto*

lemma *Ide-Trg*:
shows $Arr\ t \Longrightarrow Ide\ (Trg\ t)$
using *Ide-Subst*
by (*induct t*) *auto*

lemma *Ide-iff-Trg-self*:
shows $Arr\ t \Longrightarrow Ide\ t \longleftrightarrow Trg\ t = t$
apply (*induct t*)
apply *auto*
by (*metis Ide.simps*(5) *Ide-Subst Ide-Trg*)+

lemma *Arr-Trg* [*simp*]:
assumes *Arr X*
shows *Arr* (*Trg X*)
using *assms Ide-Trg Ide-implies-Arr* **by** *blast*

lemma *Src-Src* [*simp*]:
assumes *Arr t*
shows *Src* (*Src t*) = *Src t*
using *assms Ide-Src Ide-iff-Src-self Ide-implies-Arr* **by** *blast*

lemma *Trg-Src* [*simp*]:
assumes *Arr t*
shows *Trg* (*Src t*) = *Src t*
using *assms Ide-Src Ide-iff-Trg-self Ide-implies-Arr* **by** *blast*

lemma *Trg-Trg* [*simp*]:
assumes *Arr t*
shows *Trg* (*Trg t*) = *Trg t*
using *assms Ide-Trg Ide-iff-Trg-self Ide-implies-Arr* **by** *blast*

lemma *Src-Trg* [*simp*]:
assumes *Arr t*
shows *Src* (*Trg t*) = *Trg t*
using *assms Ide-Trg Ide-iff-Src-self Ide-implies-Arr* **by** *blast*

Two terms are syntactically *coinitial* if they are arrows with the same source; that is, they represent two reductions from the same starting term.

abbreviation *Coinitial*
where $Coinitial\ t\ u \equiv Arr\ t \wedge Arr\ u \wedge Src\ t = Src\ u$

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 $(t = \sharp \wedge t' = \sharp) \vee$

$(\exists x.\ t = \langle\!\langle x \rangle\!\rangle \wedge t' = \langle\!\langle x \rangle\!\rangle) \vee$
$(\exists u\ u'.\ t = \boldsymbol{\lambda}[u] \wedge t' = \boldsymbol{\lambda}[u']) \vee$
$(\exists u\ v\ u'\ v'.\ t = u \circ v \wedge t' = u' \circ v') \vee$
$(\exists u\ v\ u'\ v'.\ t = \boldsymbol{\lambda}[u] \bullet v \wedge t' = \boldsymbol{\lambda}[u'] \bullet v') \vee$
$(\exists u\ v\ u'\ v'.\ t = \boldsymbol{\lambda}[u] \circ v \wedge t' = \boldsymbol{\lambda}[u'] \bullet v') \vee$
$(\exists u\ v\ u'\ v'.\ t = \boldsymbol{\lambda}[u] \bullet v \wedge t' = \boldsymbol{\lambda}[u'] \circ v')$
using *assms*
by (*cases t*; *cases t'*) *auto*

lemma *Con-implies-Coinitial-ind*:
shows $[\![size\ t + size\ u \leq n;\ t \frown u]\!] \Longrightarrow Coinitial\ t\ u$
using *Con-implies-Arr1 Con-implies-Arr2*
by (*induct n arbitrary*: *t u*) *auto*

lemma *Coinitial-implies-Con-ind*:
shows $[\![size\ (Src\ t) \leq n;\ Coinitial\ t\ u]\!] \Longrightarrow t \frown u$
proof (*induct n arbitrary*: *t u*)
show $\bigwedge t\ u.\ [\![size\ (Src\ t) \leq 0;\ Coinitial\ t\ u]\!] \Longrightarrow t \frown u$
by *auto*
fix *n t u*
assume *Coinitial*: *Coinitial t u*
assume *n*: *size* (*Src t*) ≤ *Suc n*
assume *ind*: $\bigwedge t\ u.\ [\![size\ (Src\ t) \leq n;\ Coinitial\ t\ u]\!] \Longrightarrow t \frown u$
show $t \frown u$
using *n ind Coinitial Coinitial-cases* [*of t u*] *Subst-not-Nil* **by** *auto*
qed

lemma *Coinitial-iff-Con*:
shows $Coinitial\ t\ u \longleftrightarrow t \frown u$
using *Coinitial-implies-Con-ind Con-implies-Coinitial-ind* **by** *blast*

lemma *Coinitial-Raise-Raise*:
shows $Coinitial\ t\ u \Longrightarrow Coinitial\ (Raise\ d\ n\ t)\ (Raise\ d\ n\ u)$
using *Arr-Raise Src-Raise*
apply (*induct t arbitrary*: *d n u*, *auto*)
by (*metis Raise.simps*(*3*−*4*))

lemma *Con-sym*:
shows $t \frown u \longleftrightarrow u \frown t$
by (*metis Coinitial-iff-Con*)

lemma *ConI* [*intro*, *simp*]:
assumes *Arr t* **and** *Arr u* **and** *Src t* = *Src u*
shows *Con t u*
using *assms Coinitial-iff-Con* **by** *blast*

lemma *Con-Arr-Src* [*simp*]:
assumes *Arr t*
shows $t \frown Src\ t$ **and** $Src\ t \frown t$

using *assms*
by (*auto simp add*: *Ide-Src Ide-implies-Arr*)

lemma *resid-Arr-self*:
shows *Arr t* $\Longrightarrow$ *t* $\setminus$ *t* = *Trg t*
by (*induct t*) *auto*

The following result is not used in the formal development that follows, but it requires some proof and might eventually be useful.

lemma *finite-branching*:
shows *Ide a* $\Longrightarrow$ *finite* {*t*. *Arr t* $\wedge$ *Src t* = *a*}
proof (*induct a*)
show *Ide* ♯ $\Longrightarrow$ *finite* {*t*. *Arr t* $\wedge$ *Src t* = ♯}
by *simp*
fix *x*
have $\bigwedge$*t*. *Src t* = «*x*» $\longleftrightarrow$ *t* = «*x*»
using *Src.elims* **by** *blast*
thus *finite* {*t*. *Arr t* $\wedge$ *Src t* = «*x*»}
by *simp*
next
fix *a*
assume *a*: *Ide* **λ**[*a*]
assume *ind*: *Ide a* $\Longrightarrow$ *finite* {*t*. *Arr t* $\wedge$ *Src t* = *a*}
have {*t*. *Arr t* $\wedge$ *Src t* = **λ**[*a*]} = *Lam* ‘ {*t*. *Arr t* $\wedge$ *Src t* = *a*}
using *Coinitial-cases* **by** *fastforce*
thus *finite* {*t*. *Arr t* $\wedge$ *Src t* = **λ**[*a*]}
using *a ind* **by** *simp*
next
fix *a1 a2*
assume *ind1*: *Ide a1* $\Longrightarrow$ *finite* {*t*. *Arr t* $\wedge$ *Src t* = *a1*}
assume *ind2*: *Ide a2* $\Longrightarrow$ *finite* {*t*. *Arr t* $\wedge$ *Src t* = *a2*}
assume *a*: *Ide* (*a1* ∘ *a2*)
show *finite* {*t*. *Arr t* $\wedge$ *Src t* = **λ**[*a1*] • *a2*}
using *a ind1 ind2* **by** *simp*
next
fix *a1 a2*
assume *ind1*: *Ide a1* $\Longrightarrow$ *finite* {*t*. *Arr t* $\wedge$ *Src t* = *a1*}
assume *ind2*: *Ide a2* $\Longrightarrow$ *finite* {*t*. *Arr t* $\wedge$ *Src t* = *a2*}
assume *a*: *Ide* (*a1* ∘ *a2*)
have {*t*. *Arr t* $\wedge$ *Src t* = *a1* ∘ *a2*} =
({*t*. *is-App t*} $\cap$ ({*t*. *Arr t* $\wedge$ *Src* (*un-App1 t*) = *a1* $\wedge$ *Src* (*un-App2 t*) = *a2*})) $\cup$
({*t*. *is-Beta t* $\wedge$ *is-Lam a1*} $\cap$
({*t*. *Arr t* $\wedge$ *is-Lam a1* $\wedge$ *Src* (*un-Beta1 t*) = *un-Lam a1* $\wedge$ *Src* (*un-Beta2 t*) = *a2*}))
by *fastforce*
have {*t*. *Arr t* $\wedge$ *Src t* = *a1* ∘ *a2*} =
(λ(*t1*, *t2*). *t1* ∘ *t2*) ‘ ({*t1*. *Arr t1* $\wedge$ *Src t1* = *a1*} × {*t2*. *Arr t2* $\wedge$ *Src t2* = *a2*}) $\cup$
(λ(*t1*, *t2*). **λ**[*t1*] • *t2*) ‘
({*t1t2*. *is-Lam a1*} $\cap$
{*t1*. *Arr t1* $\wedge$ *Src t1* = *un-Lam a1*} × {*t2*. *Arr t2* $\wedge$ *Src t2* = *a2*})

proof
show $(\lambda(t1,\ t2).\ t1 \circ t2)\ `\ (\{t1.\ Arr\ t1 \wedge Src\ t1 = a1\} \times \{t2.\ Arr\ t2 \wedge Src\ t2 = a2\}) \cup$
$(\lambda(t1,\ t2).\ \boldsymbol{\lambda}[t1] \bullet t2)\ `$
$(\{t1t2.\ is\text{-}Lam\ a1\} \cap$
$\{t1.\ Arr\ t1 \wedge Src\ t1 = un\text{-}Lam\ a1\} \times \{t2.\ Arr\ t2 \wedge Src\ t2 = a2\})$
$\subseteq \{t.\ Arr\ t \wedge Src\ t = a1 \circ a2\}$
by *auto*
show $\{t.\ Arr\ t \wedge Src\ t = a1 \circ a2\}$
$\subseteq (\lambda(t1,\ t2).\ t1 \circ t2)\ `$
$(\{t1.\ Arr\ t1 \wedge Src\ t1 = a1\} \times \{t2.\ Arr\ t2 \wedge Src\ t2 = a2\}) \cup$
$(\lambda(t1,\ t2).\ \boldsymbol{\lambda}[t1] \bullet t2)\ `$
$(\{t1t2.\ is\text{-}Lam\ a1\} \cap$
$\{t1.\ Arr\ t1 \wedge Src\ t1 = un\text{-}Lam\ a1\} \times \{t2.\ Arr\ t2 \wedge Src\ t2 = a2\})$
proof
fix t
assume t: $t \in \{t.\ Arr\ t \wedge Src\ t = a1 \circ a2\}$
have $is\text{-}App\ t \vee is\text{-}Beta\ t$
using t **by** *auto*
moreover have $is\text{-}App\ t \Longrightarrow t \in (\lambda(t1,\ t2).\ t1 \circ t2)\ `$
$(\{t1.\ Arr\ t1 \wedge Src\ t1 = a1\} \times \{t2.\ Arr\ t2 \wedge Src\ t2 = a2\})$
using t *image-iff is-App-def* **by** *fastforce*
moreover have $is\text{-}Beta\ t \Longrightarrow$
$t \in (\lambda(t1,\ t2).\ \boldsymbol{\lambda}[t1] \bullet t2)\ `$
$(\{t1t2.\ is\text{-}Lam\ a1\} \cap$
$\{t1.\ Arr\ t1 \wedge Src\ t1 = un\text{-}Lam\ a1\} \times \{t2.\ Arr\ t2 \wedge Src\ t2 = a2\})$
using t *is-Beta-def* **by** *fastforce*
ultimately show $t \in (\lambda(t1,\ t2).\ t1 \circ t2)\ `$
$(\{t1.\ Arr\ t1 \wedge Src\ t1 = a1\} \times \{t2.\ Arr\ t2 \wedge Src\ t2 = a2\}) \cup$
$(\lambda(t1,\ t2).\ \boldsymbol{\lambda}[t1] \bullet t2)\ `$
$(\{t1t2.\ is\text{-}Lam\ a1\} \cap$
$\{t1.\ Arr\ t1 \wedge Src\ t1 = un\text{-}Lam\ a1\} \times \{t2.\ Arr\ t2 \wedge Src\ t2 = a2\})$
by *blast*
qed
qed
moreover have $finite\ (\{t1.\ Arr\ t1 \wedge Src\ t1 = a1\} \times \{t2.\ Arr\ t2 \wedge Src\ t2 = a2\})$
using *a ind1 ind2 Ide.simps*(4) **by** *blast*
moreover have $is\text{-}Lam\ a1 \Longrightarrow$
$finite\ (\{t1.\ Arr\ t1 \wedge Src\ t1 = un\text{-}Lam\ a1\} \times \{t2.\ Arr\ t2 \wedge Src\ t2 = a2\})$
proof −
assume *a1*: *is-Lam a1*
have *Ide* (*un-Lam a1*)
using *a a1 is-Lam-def* **by** *force*
have *Lam* ` $\{t1.\ Arr\ t1 \wedge Src\ t1 = un\text{-}Lam\ a1\} = \{t.\ Arr\ t \wedge Src\ t = a1\}$
proof
show *Lam* ` $\{t1.\ Arr\ t1 \wedge Src\ t1 = un\text{-}Lam\ a1\} \subseteq \{t.\ Arr\ t \wedge Src\ t = a1\}$
using *a1* **by** *fastforce*
show $\{t.\ Arr\ t \wedge Src\ t = a1\} \subseteq Lam\ `\ \{t1.\ Arr\ t1 \wedge Src\ t1 = un\text{-}Lam\ a1\}$
proof
fix t

```
      assume t: t ∈ {t. Arr t ∧ Src t = a1}
      have is-Lam t
        using a1 t by auto
      hence un-Lam t ∈ {t1. Arr t1 ∧ Src t1 = un-Lam a1}
        using is-Lam-def t by force
      thus t ∈ Lam ‘ {t1. Arr t1 ∧ Src t1 = un-Lam a1}
        by (metis ‹is-Lam t› lambda.collapse(2) rev-image-eqI)
    qed
  qed
  moreover have inj Lam
    using inj-on-def by blast
  ultimately have finite {t1. Arr t1 ∧ Src t1 = un-Lam a1}
    by (metis (mono-tags, lifting) Ide.simps(4) a finite-imageD ind1 injD inj-onI)
  moreover have finite {t2. Arr t2 ∧ Src t2 = a2}
    using Ide.simps(4) a ind2 by blast
  ultimately
  show finite ({t1. Arr t1 ∧ Src t1 = un-Lam a1} × {t2. Arr t2 ∧ Src t2 = a2})
    by blast
qed
ultimately show finite {t. Arr t ∧ Src t = a1 ⋅ a2}
  using a ind1 ind2 by simp
qed
```

3.2.3 Residuation and Substitution

We now develop a series of lemmas that involve the interaction of residuation and substitution.

```
lemma Raise-resid:
shows t ⌣ u ⟹ Raise k n (t \ u) = Raise k n t \ Raise k n u
proof −

  let ?P = λ(t, u). ∀k n. t ⌣ u ⟶ Raise k n (t \ u) = Raise k n t \ Raise k n u
  have ⋀t u.
         ∀t′ u′. ((t′, u′), (t, u)) ∈ subterm-pair-rel ⟶
                 (∀k n. t′ ⌣ u′ ⟶
                        Raise k n (t′ \ u′) = Raise k n t′ \ Raise k n u′) ⟹
         (⋀k n. t ⌣ u ⟹ Raise k n (t \ u) = Raise k n t \ Raise k n u)
    using subterm-lemmas Coinitial-iff-Con Coinitial-Raise-Raise Raise-subst by auto
  thus t ⌣ u ⟹ Raise k n (t \ u) = Raise k n t \ Raise k n u
    using wf-subterm-pair-rel wf-induct [of subterm-pair-rel ?P] by blast
qed

lemma Con-Raise:
shows t ⌣ u ⟹ Raise d n t ⌣ Raise d n u
  by (metis Raise-not-Nil Raise-resid)
```

The following is Huet's Commutation Theorem [7]: "substitution commutes with residuation".

```
lemma resid-Subst:
```

assumes $t \frown t'$ **and** $u \frown u'$
shows *Subst n t u* $\setminus$ *Subst n t' u'* = *Subst n* $(t \setminus t')$ $(u \setminus u')$
proof −
  let *?P* = $\lambda(u, u').\ \forall n\ t\ t'.\ t \frown t' \wedge u \frown u' \longrightarrow$
                Subst n t u $\setminus$ *Subst n t' u'* = *Subst n* $(t \setminus t')$ $(u \setminus u')$
  have $\bigwedge u\ u'.\ \forall w\ w'.\ ((w, w'), (u, u')) \in$ *subterm-pair-rel* $\longrightarrow$
            $(\forall n\ v\ v'.\ v \frown v' \wedge w \frown w' \longrightarrow$
              Subst n v w $\setminus$ *Subst n v' w'* = *Subst n* $(v \setminus v')$ $(w \setminus w')) \Longrightarrow$
      $\forall n\ t\ t'.\ t \frown t' \wedge u \frown u' \longrightarrow$
            Subst n t u $\setminus$ *Subst n t' u'* = *Subst n* $(t \setminus t')$ $(u \setminus u')$
   using *subterm-lemmas Raise-resid Subst-not-Nil Con-Raise Raise-subst substitution-lemma*
    by *auto*
  thus *?thesis*
    using *assms wf-subterm-pair-rel wf-induct* [*of subterm-pair-rel ?P*] **by** *auto*
qed

lemma *Trg-Subst* [*simp*]:
shows $[\![Arr\ t;\ Arr\ u]\!] \Longrightarrow$ *Trg* (*Subst d t u*) = *Subst d* (*Trg t*) (*Trg u*)
  by (*metis Arr-Subst Arr-Trg Arr-not-Nil resid-Arr-self resid-Subst*)

lemma *Src-resid*:
shows $t \frown u \Longrightarrow$ *Src* $(t \setminus u)$ = *Trg u*
proof (*induct u arbitrary*: *t*, *auto simp add*: *Arr-resid*)
  fix *t t1'*
  show $\bigwedge t2'.\ [\![\bigwedge t1.\ t1 \frown t1' \Longrightarrow$ *Src* $(t1 \setminus t1')$ = *Trg t1'*;
          $\bigwedge t2.\ t2 \frown t2' \Longrightarrow$ *Src* $(t2 \setminus t2')$ = *Trg t2'*;
          $t \frown t1' \circ t2']\!]$
            $\Longrightarrow$ *Src* $(t \setminus (t1' \circ t2'))$ = *Trg t1'* $\circ$ *Trg t2'*
    apply (*cases t*; *cases t1'*)
                apply *auto*
    by (*metis Src.simps*(*3*) *lambda.distinct*(*3*) *lambda.sel*(*2*) *resid.simps*(*2*))
qed

lemma *Coinitial-resid-resid*:
assumes $t \frown v$ **and** $u \frown v$
shows *Coinitial* $(t \setminus v)$ $(u \setminus v)$
  using *assms Src-resid Arr-resid Coinitial-iff-Con* **by** *presburger*

lemma *Con-implies-is-Lam-iff-is-Lam*:
assumes $t \frown u$
shows *is-Lam t* $\longleftrightarrow$ *is-Lam u*
  using *assms* **by** *auto*

lemma *Con-implies-Coinitial3*:
assumes $t \setminus v \frown u \setminus v$
shows *Coinitial v u* **and** *Coinitial v t* **and** *Coinitial u t*
  using *assms*
  by (*metis Coinitial-iff-Con resid.simps*(*7*))+

We can now prove Lévy's "Cube Lemma" [8], which is the key axiom for a residuated

transition system.

```
lemma Cube:
shows v \ t ⌢ u \ t ⟹ (v \ t) \ (u \ t) = (v \ u) \ (t \ u)
proof −
  let ?P = λ(t, u, v). v \ t ⌢ u \ t ⟶ (v \ t) \ (u \ t) = (v \ u) \ (t \ u)
  have ⋀t u v.
         ∀ t′ u′ v′.
           ((t′, u′, v′), (t, u, v)) ∈ subterm-triple-rel ⟶ ?P (t′, u′, v′) ⟹
             v \ t ⌢ u \ t ⟶ (v \ t) \ (u \ t) = (v \ u) \ (t \ u)
  proof −
    fix t u v
    assume ind: ∀ t′ u′ v′.
                  ((t′, u′, v′), (t, u, v)) ∈ subterm-triple-rel ⟶ ?P (t′, u′, v′)
    show v \ t ⌢ u \ t ⟶ (v \ t) \ (u \ t) = (v \ u) \ (t \ u)
    proof (intro impI)
      assume con: v \ t ⌢ u \ t
      have Con v t
        using con by auto
      moreover have Con u t
        using con by auto
      ultimately show (v \ t) \ (u \ t) = (v \ u) \ (t \ u)
        using subterm-lemmas ind Coinitial-iff-Con Coinitial-resid-resid resid-Subst
        apply (elim ConE [of v t] ConE [of u t])
                        apply simp-all
              apply metis
             apply metis
            apply (cases un-App1 t; cases un-App1 v, simp-all)
            apply metis
           apply metis
          apply metis
         apply metis
        apply metis
       apply (cases un-App1 u, simp-all)
       apply metis
      by metis
    qed
  qed
  hence ?P (t, u, v)
    using wf-subterm-triple-rel wf-induct [of subterm-triple-rel ?P] by blast
  thus v \ t ⌢ u \ t ⟹ (v \ t) \ (u \ t) = (v \ u) \ (t \ u)
    by simp
qed
```

3.2.4 Residuation Determines an RTS

We are now in a position to verify that the residuation operation that we have defined satisfies the axioms for a residuated transition system, and that various notions which we have defined syntactically above (*e.g.* arrow, source, target) agree with the versions derived abstractly from residuation.

sublocale *partial-magma resid*
  apply *unfold-locales*
  by (*metis Arr.simps*(*1*) *Coinitial-iff-Con*)

lemma *null-char* [*simp*]:
shows *null* = ♯
  using *null-def*
  by (*metis null-is-zero*(*2*) *resid.simps*(*7*))

sublocale *residuation resid*
  using *null-char Arr-resid Coinitial-iff-Con Cube*
  apply (*unfold-locales*, *auto*)
  by *metis*+

notation *resid* (**infix** ‹\› *70*)

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

lemma *arr-char* [*iff*]:
shows *arr t* ⟷ *Arr t*
  using *Coinitial-iff-Con arr-def con-def null-char* **by** *auto*

lemma *ide-char* [*iff*]:
shows *ide t* ⟷ *Ide t*
  by (*metis Ide-iff-Trg-self Ide-implies-Arr arr-char arr-resid-iff-con ide-def*
      resid-Arr-self)

lemma *resid-Arr-Ide*:
shows ⟦*Ide a*; *Coinitial t a*⟧ ⟹ *t* \ *a* = *t*
  using *Ide-iff-Src-self*
  by (*induct t arbitrary*: *a*, *auto*)

lemma *resid-Ide-Arr*:
shows ⟦*Ide a*; *Coinitial a t*⟧ ⟹ *Ide* (*a* \ *t*)
  by (*metis Coinitial-resid-resid ConI Ide-iff-Trg-self cube resid-Arr-Ide resid-Arr-self*)

lemma *resid-Arr-Src* [*simp*]:
assumes *Arr t*
shows *t* \ *Src t* = *t*
  using *assms Ide-Src*
  by (*simp add*: *Ide-implies-Arr resid-Arr-Ide*)

lemma *resid-Src-Arr* [*simp*]:
assumes *Arr t*
shows *Src t* \ *t* = *Trg t*
  using *assms*

by (*metis* (*full-types*) *Con-Arr-Src*(*2*) *Con-implies-Arr1 Src-Src Src-resid cube resid-Arr-Src resid-Arr-self*)

sublocale *rts resid*
proof
show $\bigwedge a\ t.\ [\![ide\ a;\ con\ t\ a]\!] \Longrightarrow t \setminus a = t$
using *ide-char resid-Arr-Ide*
by (*metis Coinitial-iff-Con con-def null-char*)
show $\bigwedge t.\ arr\ t \Longrightarrow ide\ (trg\ t)$
by (*simp add*: *Ide-Trg resid-Arr-self trg-def*)
show $\bigwedge a\ t.\ [\![ide\ a;\ con\ a\ t]\!] \Longrightarrow ide\ (resid\ a\ t)$
using *ide-char null-char resid-Ide-Arr Coinitial-iff-Con con-def* **by** *force*
show $\bigwedge t\ u.\ con\ t\ u \Longrightarrow \exists a.\ ide\ a \wedge con\ a\ t \wedge con\ a\ u$
by (*metis Coinitial-iff-Con Ide-Src Ide-iff-Src-self Ide-implies-Arr con-def ide-char null-char*)
show $\bigwedge t\ u\ v.\ [\![ide\ (resid\ t\ u);\ con\ u\ v]\!] \Longrightarrow con\ (resid\ t\ u)\ (resid\ v\ u)$
by (*metis Coinitial-resid-resid ide-char not-arr-null null-char resid-Ide-Arr con-def con-sym ide-implies-arr*)
qed

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

lemma *sources-char*$_\Lambda$:
shows *sources t* = (*if Arr t then* {*Src t*} *else* {})
proof (*cases Arr t*)
show ¬ *Arr t* ⟹ *?thesis*
using *arr-char arr-iff-has-source* **by** *auto*
assume *t*: *Arr t*
have *1*: {*Src t*} ⊆ *sources t*
using *t Ide-Src* **by** *force*
moreover have *sources t* ⊆ {*Src t*}
by (*metis Coinitial-iff-Con Ide-iff-Src-self ide-char in-sourcesE null-char con-def singleton-iff subsetI*)
ultimately show *?thesis*
using *t* **by** *auto*
qed

lemma *sources-simp* [*simp*]:
assumes *Arr t*
shows *sources t* = {*Src t*}
using *assms sources-char*$_\Lambda$ **by** *auto*

lemma *sources-simps* [*simp*]:
shows *sources* ♯ = {}
and *sources «x»* = {*«x»*}
and *arr t* ⟹ *sources* **λ**[*t*] = {**λ**[*Src t*]}
and [*arr t*; *arr u*] ⟹ *sources* (*t* ⋅ *u*) = {*Src t* ⋅ *Src u*}

and $[\![arr\ t;\ arr\ u]\!] \Longrightarrow sources\ (\boldsymbol{\lambda}[t] \bullet u) = \{\boldsymbol{\lambda}[Src\ t] \circ Src\ u\}$
using $sources\text{-}char_\Lambda$ **by** *auto*

lemma $targets\text{-}char_\Lambda$:
shows $targets\ t = (if\ Arr\ t\ then\ \{Trg\ t\}\ else\ \{\})$
proof (*cases Arr t*)
show $\neg\ Arr\ t \Longrightarrow ?thesis$
by (*meson arr-char arr-iff-has-target*)
assume *t*: $Arr\ t$
have *1*: $\{Trg\ t\} \subseteq targets\ t$
using *t resid-Arr-self trg-def trg-in-targets* **by** *force*
moreover have $targets\ t \subseteq \{Trg\ t\}$
by (*metis 1 Ide-iff-Src-self arr-char ide-char ide-implies-arr in-targetsE insert-subset prfx-implies-con resid-Arr-self sources-resid sources-simp t*)
ultimately show *?thesis*
using *t* **by** *auto*
qed

lemma *targets-simp* [*simp*]:
assumes $Arr\ t$
shows $targets\ t = \{Trg\ t\}$
using *assms* $targets\text{-}char_\Lambda$ **by** *auto*

lemma *targets-simps* [*simp*]:
shows $targets\ \sharp = \{\}$
and $targets\ «x» = \{«x»\}$
and $arr\ t \Longrightarrow targets\ \boldsymbol{\lambda}[t] = \{\boldsymbol{\lambda}[Trg\ t]\}$
and $[\![arr\ t;\ arr\ u]\!] \Longrightarrow targets\ (t \circ u) = \{Trg\ t \circ Trg\ u\}$
and $[\![arr\ t;\ arr\ u]\!] \Longrightarrow targets\ (\boldsymbol{\lambda}[t] \bullet u) = \{subst\ (Trg\ u)\ (Trg\ t)\}$
using $targets\text{-}char_\Lambda$ **by** *auto*

lemma *seq-char*:
shows $seq\ t\ u \longleftrightarrow Arr\ t \wedge Arr\ u \wedge Trg\ t = Src\ u$
using *seq-def arr-char* $sources\text{-}char_\Lambda$ $targets\text{-}char_\Lambda$ **by** *force*

lemma $seqI_\Lambda$ [*intro*, *simp*]:
assumes $Arr\ t$ **and** $Arr\ u$ **and** $Trg\ t = Src\ u$
shows $seq\ t\ u$
using *assms seq-char* **by** *simp*

lemma $seqE_\Lambda$ [*elim*]:
assumes $seq\ t\ u$
and $[\![Arr\ t;\ Arr\ u;\ Trg\ t = Src\ u]\!] \Longrightarrow T$
shows *T*
using *assms seq-char* **by** *blast*

The following classifies the ways that transitions can be sequential. It is useful for later proofs by case analysis.

lemma *seq-cases*:
assumes *seq t u*
shows (*is-Var t* ∧ *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
using *assms seq-char*
by (*cases t*; *cases u*) *auto*

sublocale *confluent-rts resid*
by (*unfold-locales*) *fastforce*

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

lemma *con-char* [*iff*]:
shows *con t u* ⟷ *Con t u*
by *fastforce*

lemma *coinitial-char* [*iff*]:
shows *coinitial t u* ⟷ *Coinitial t u*
by *fastforce*

lemma *sources-Raise*:
assumes *Arr t*
shows *sources* (*Raise d n t*) = {*Raise d n* (*Src t*)}
using *assms*
by (*simp add*: *Coinitial-Raise-Raise Src-Raise*)

lemma *targets-Raise*:
assumes *Arr t*
shows *targets* (*Raise d n t*) = {*Raise d n* (*Trg t*)}
using *assms*
by (*metis Arr-Raise ConI Raise-resid resid-Arr-self targets-char*$_\Lambda$)

lemma *sources-subst* [*simp*]:
assumes *Arr t* **and** *Arr u*
shows *sources* (*subst t u*) = {*subst* (*Src t*) (*Src u*)}
using *assms sources-char*$_\Lambda$ *Arr-Subst arr-char* **by** *simp*

lemma *targets-subst* [*simp*]:
assumes *Arr t* **and** *Arr u*
shows *targets* (*subst t u*) = {*subst* (*Trg t*) (*Trg u*)}
using *assms targets-char*$_\Lambda$ *Arr-Subst arr-char* **by** *simp*

notation *prfx* (**infix** ‹≲› *50*)

notation *cong* (**infix** ‹∼› *50*)

lemma *prfx-char* [*iff*]:
shows $t \lesssim u \longleftrightarrow Ide\ (t \setminus u)$
using *ide-char* **by** *simp*

lemma *prfx-Var-iff*:
shows $u \lesssim «i» \longleftrightarrow u = «i»$
by (*metis Arr.simps*(*2*) *Coinitial-iff-Con Ide.simps*(*1*) *Ide-iff-Src-self Src.simps*(*2*)
ide-char resid-Arr-Ide)

lemma *prfx-Lam-iff*:
shows $u \lesssim Lam\ t \longleftrightarrow is\text{-}Lam\ u \wedge un\text{-}Lam\ u \lesssim t$
using *ide-char Arr-not-Nil Con-implies-is-Lam-iff-is-Lam Ide-implies-Arr is-Lam-def*
by *fastforce*

lemma *prfx-App-iff*:
shows $u \lesssim t1 \circ t2 \longleftrightarrow is\text{-}App\ u \wedge un\text{-}App1\ u \lesssim t1 \wedge un\text{-}App2\ u \lesssim t2$
using *ide-char*
by (*cases u*; *cases t1*) *auto*

lemma *prfx-Beta-iff*:
shows $u \lesssim \boldsymbol{\lambda}[t1] \bullet t2 \longleftrightarrow$
$(is\text{-}App\ u \wedge un\text{-}App1\ u \lesssim \boldsymbol{\lambda}[t1] \wedge un\text{-}App2\ u \frown t2 \wedge$
$(0 \in FV\ (un\text{-}Lam\ (un\text{-}App1\ u) \setminus t1) \longrightarrow un\text{-}App2\ u \lesssim t2)) \vee$
$(is\text{-}Beta\ u \wedge un\text{-}Beta1\ u \lesssim t1 \wedge un\text{-}Beta2\ u \frown t2 \wedge$
$(0 \in FV\ (un\text{-}Beta1\ u \setminus t1) \longrightarrow un\text{-}Beta2\ u \lesssim t2))$
using *ide-char Ide-Subst-iff*
by (*cases u*; *cases un-App1 u*) *auto*

lemma *cong-Ide-are-eq*:
assumes $t \sim u$ **and** *Ide t* **and** *Ide u*
shows $t = u$
using *assms*
by (*metis Coinitial-iff-Con Ide-iff-Src-self con-char prfx-implies-con*)

lemma *eq-Ide-are-cong*:
assumes $t = u$ **and** *Ide t*
shows $t \sim u$
using *assms Ide-implies-Arr resid-Ide-Arr* **by** *blast*

sublocale *weakly-extensional-rts resid*
apply *unfold-locales*
by (*metis Coinitial-iff-Con Ide-iff-Src-self Ide-implies-Arr ide-char ide-def*)

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

lemma *src-char* [*simp*]:
shows *src t* = (*if Arr t then Src t else* ♯)
using *src-def* **by** *force*

lemma *trg-char* [*simp*]:
shows *trg t* = (*if Arr t then Trg t else* ♯)
by (*metis Coinitial-iff-Con resid-Arr-self trg-def*)

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-extensional-ex1* ≡ $\boldsymbol{\lambda}[\boldsymbol{\lambda}[\text{«0»}] \circ \boldsymbol{\lambda}[\text{«0»}]] \bullet (\boldsymbol{\lambda}[\text{«0»}] \bullet \boldsymbol{\lambda}[\text{«0»}])$

abbreviation *non-extensional-ex2*
where *non-extensional-ex2* ≡ $\boldsymbol{\lambda}[\boldsymbol{\lambda}[\text{«0»}] \circ \boldsymbol{\lambda}[\text{«0»}]] \bullet (\boldsymbol{\lambda}[\text{«0»}] \circ \boldsymbol{\lambda}[\text{«0»}])$

lemma *non-extensional*:
shows $\boldsymbol{\lambda}[\text{«1»}] \bullet$ *non-extensional-ex1* $\sim \boldsymbol{\lambda}[\text{«1»}] \bullet$ *non-extensional-ex2*
and $\boldsymbol{\lambda}[\text{«1»}] \bullet$ *non-extensional-ex1* $\neq \boldsymbol{\lambda}[\text{«1»}] \bullet$ *non-extensional-ex2*
by *auto*

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* ≡
$\boldsymbol{\lambda}[\text{«0»} \circ \text{«0»}] \circ \boldsymbol{\lambda}[\text{«0»} \circ \text{«0»}] \circ (\boldsymbol{\lambda}[\text{«0»} \circ \text{«0»}] \bullet \boldsymbol{\lambda}[\text{«0»} \circ \text{«0»}])$

abbreviation *cong-nontrivial-ex2*
where *cong-nontrivial-ex2* ≡
$\boldsymbol{\lambda}[\text{«0»} \circ \text{«0»}] \bullet \boldsymbol{\lambda}[\text{«0»} \circ \text{«0»}] \circ (\boldsymbol{\lambda}[\text{«0»} \circ \text{«0»}] \circ \boldsymbol{\lambda}[\text{«0»} \circ \text{«0»}])$

lemma *cong-nontrivial*:
shows *coinitial cong-nontrivial-ex1 cong-nontrivial-ex2*
and *coterminal cong-nontrivial-ex1 cong-nontrivial-ex2*
and ¬ *cong cong-nontrivial-ex1 cong-nontrivial-ex2*
by *auto*

Every two coinitial transitions have a join, obtained structurally by unioning the sets of marked redexes.

fun *Join* (**infix** ‹⊔› *52*)
where «*x*» ⊔ «*x′*» = (*if x* = *x′ then* «*x*» *else* ♯)
| $\boldsymbol{\lambda}[t] \sqcup \boldsymbol{\lambda}[t'] = \boldsymbol{\lambda}[t \sqcup t']$
| $\boldsymbol{\lambda}[t] \circ u \sqcup \boldsymbol{\lambda}[t'] \bullet u' = \boldsymbol{\lambda}[(t \sqcup t')] \bullet (u \sqcup u')$
| $\boldsymbol{\lambda}[t] \bullet u \sqcup \boldsymbol{\lambda}[t'] \circ u' = \boldsymbol{\lambda}[(t \sqcup t')] \bullet (u \sqcup u')$
| $t \circ u \sqcup t' \circ u' = (t \sqcup t') \circ (u \sqcup u')$
| $\boldsymbol{\lambda}[t] \bullet u \sqcup \boldsymbol{\lambda}[t'] \bullet u' = \boldsymbol{\lambda}[(t \sqcup t')] \bullet (u \sqcup u')$
| - ⊔ - = ♯

lemma *Join-sym*:
shows $t \sqcup u = u \sqcup t$
  using *Join.induct* [*of* $\lambda t\ u.\ t \sqcup u = u \sqcup t$] **by** *auto*

lemma *Src-Join*:
shows *Coinitial t u* $\Longrightarrow$ *Src* $(t \sqcup u)$ = *Src t*
proof (*induct t arbitrary*: *u*)
  show $\bigwedge u.$ *Coinitial* $\sharp$ *u* $\Longrightarrow$ *Src* $(\sharp \sqcup u)$ = *Src* $\sharp$
    by *simp*
  show $\bigwedge x\ u.$ *Coinitial «x» u* $\Longrightarrow$ *Src* $(«x» \sqcup u)$ = *Src «x»*
    by *auto*
  fix *t u*
  assume *ind*: $\bigwedge u.$ *Coinitial t u* $\Longrightarrow$ *Src* $(t \sqcup u)$ = *Src t*
  assume *tu*: *Coinitial* $\boldsymbol{\lambda}[t]$ *u*
  show *Src* $(\boldsymbol{\lambda}[t] \sqcup u)$ = *Src* $\boldsymbol{\lambda}[t]$
    using *tu ind*
    by (*cases u*) *auto*
  next
  fix *t1 t2 u*
  assume *ind1*: $\bigwedge u1.$ *Coinitial t1 u1* $\Longrightarrow$ *Src* $(t1 \sqcup u1)$ = *Src t1*
  assume *ind2*: $\bigwedge u2.$ *Coinitial t2 u2* $\Longrightarrow$ *Src* $(t2 \sqcup u2)$ = *Src t2*
  assume *tu*: *Coinitial* $(t1 \circ t2)$ *u*
  show *Src* $(t1 \circ t2 \sqcup u)$ = *Src* $(t1 \circ t2)$
    using *tu ind1 ind2*
    apply (*cases u*, *simp-all*)
    apply (*cases t1*, *simp-all*)
    by (*metis Arr.simps*(3) *Join.simps*(2) *Src.simps*(3) *lambda.sel*(2))
  next
  fix *t1 t2 u*
  assume *ind1*: $\bigwedge u1.$ *Coinitial t1 u1* $\Longrightarrow$ *Src* $(t1 \sqcup u1)$ = *Src t1*
  assume *ind2*: $\bigwedge u2.$ *Coinitial t2 u2* $\Longrightarrow$ *Src* $(t2 \sqcup u2)$ = *Src t2*
  assume *tu*: *Coinitial* $(\boldsymbol{\lambda}[t1] \bullet t2)$ *u*
  show *Src* $((\boldsymbol{\lambda}[t1] \bullet t2) \sqcup u)$ = *Src* $(\boldsymbol{\lambda}[t1] \bullet t2)$
    using *tu ind1 ind2*
    apply (*cases u*, *simp-all*)
    by (*cases un-App1 u*) *auto*
qed

lemma *resid-Join*:
shows *Coinitial t u* $\Longrightarrow$ $(t \sqcup u) \setminus u = t \setminus u$
proof (*induct t arbitrary*: *u*)
  show $\bigwedge u.$ *Coinitial* $\sharp$ *u* $\Longrightarrow$ $(\sharp \sqcup u) \setminus u = \sharp \setminus u$
    by *auto*
  show $\bigwedge x\ u.$ *Coinitial «x» u* $\Longrightarrow$ $(«x» \sqcup u) \setminus u = «x» \setminus u$
    by *auto*
  fix *t u*
  assume *ind*: $\bigwedge u.$ *Coinitial t u* $\Longrightarrow$ $(t \sqcup u) \setminus u = t \setminus u$
  assume *tu*: *Coinitial* $\boldsymbol{\lambda}[t]$ *u*

show $(\boldsymbol{\lambda}[t] \sqcup u) \setminus u = \boldsymbol{\lambda}[t] \setminus u$
using *tu ind*
by (*cases u*) *auto*
next
fix *t1 t2 u*
assume *ind1*: $\bigwedge u1.\ Coinitial\ t1\ u1 \Longrightarrow (t1 \sqcup u1) \setminus u1 = t1 \setminus u1$
assume *ind2*: $\bigwedge u2.\ Coinitial\ t2\ u2 \Longrightarrow (t2 \sqcup u2) \setminus u2 = t2 \setminus u2$
assume *tu*: *Coinitial* $(t1 \circ t2)\ u$
show $(t1 \circ t2 \sqcup u) \setminus u = (t1 \circ t2) \setminus u$
using *tu ind1 ind2 Coinitial-iff-Con*
apply (*cases u*, *simp-all*)
proof −
fix *u1 u2*
assume *u*: $u = \boldsymbol{\lambda}[u1] \bullet u2$
have *t2u2*: $t2 \frown u2$
using *Arr-not-Nil Arr-resid tu u* **by** *simp*
have *t1u1*: *Coinitial* $(un\text{-}Lam\ t1 \sqcup u1)\ u1$
proof −
have *Arr* $(un\text{-}Lam\ t1 \sqcup u1)$
by (*metis Arr.simps*(3−5) *Coinitial-iff-Con Con-implies-is-Lam-iff-is-Lam Join.simps*(2) *Src.simps*(3−5) *ind1 lambda.collapse*(2) *lambda.disc*(8) *lambda.sel*(3) *tu u*)
thus *?thesis*
using *Src-Join*
by (*metis Arr.simps*(3−5) *Coinitial-iff-Con Con-implies-is-Lam-iff-is-Lam Src.simps*(3−5) *lambda.collapse*(2) *lambda.disc*(8) *lambda.sel*(2−3) *tu u*)
qed
have $un\text{-}Lam\ t1 \frown u1$
using *t1u1*
by (*metis Coinitial-iff-Con Con-implies-is-Lam-iff-is-Lam ConD*(4) *lambda.collapse*(2) *lambda.disc*(8) *resid.simps*(2) *tu u*)
thus $(t1 \circ t2 \sqcup \boldsymbol{\lambda}[u1] \bullet u2) \setminus (\boldsymbol{\lambda}[u1] \bullet u2) = (t1 \circ t2) \setminus (\boldsymbol{\lambda}[u1] \bullet u2)$
using *u tu t1u1 t2u2 ind1 ind2*
apply (*cases t1*, *auto*)
proof −
fix *v*
assume *v*: $t1 = \boldsymbol{\lambda}[v]$
show *subst* $(t2 \setminus u2)\ ((v \sqcup u1) \setminus u1) = subst\ (t2 \setminus u2)\ (v \setminus u1)$
proof −
have *subst* $(t2 \setminus u2)\ ((v \sqcup u1) \setminus u1) = (t1 \circ t2 \sqcup \boldsymbol{\lambda}[u1] \bullet u2) \setminus (\boldsymbol{\lambda}[u1] \bullet u2)$
by (*simp add*: *Coinitial-iff-Con ind2 t2u2 v*)
also have ... $= (t1 \circ t2) \setminus (\boldsymbol{\lambda}[u1] \bullet u2)$
proof −
have $(t1 \circ t2 \sqcup \boldsymbol{\lambda}[u1] \bullet u2) \setminus (\boldsymbol{\lambda}[u1] \bullet u2) =$
$(\boldsymbol{\lambda}[(v \sqcup u1)] \bullet (t2 \sqcup u2)) \setminus (\boldsymbol{\lambda}[u1] \bullet u2)$
using *v* **by** *simp*
also have ... $= subst\ (t2 \setminus u2)\ ((v \sqcup u1) \setminus u1)$
by (*simp add*: *Coinitial-iff-Con ind2 t2u2*)
also have ... $= subst\ (t2 \setminus u2)\ (v \setminus u1)$

```
        proof −
          have (t1 ⊔ λ[u1]) \ λ[u1] = t1 \ λ[u1]
            using u tu ind1 by simp
          thus ?thesis
            using ‹un-Lam t1 \ u1 ≠ ♯› t1u1 v by force
        qed
        also have ... = (t1 ∘ t2) \ (λ[u1] • u2)
          using tu u v by force
        finally show ?thesis by blast
      qed
      also have ... = subst (t2 \ u2) (v \ u1)
        by (simp add: t2u2 v)
      finally show ?thesis by auto
    qed
  qed
  next
  fix t1 t2 u
  assume ind1: ⋀u1. Coinitial t1 u1 ⟹ (t1 ⊔ u1) \ u1 = t1 \ u1
  assume ind2: ⋀u2. Coinitial t2 u2 ⟹ (t2 ⊔ u2) \ u2 = t2 \ u2
  assume tu: Coinitial (λ[t1] • t2) u
  show ((λ[t1] • t2) ⊔ u) \ u = (λ[t1] • t2) \ u
    using tu ind1 ind2 Coinitial-iff-Con
    apply (cases u, simp-all)
  proof −
    fix u1 u2
    assume u: u = u1 ∘ u2
    show (λ[t1] • t2 ⊔ u1 ∘ u2) \ (u1 ∘ u2) = (λ[t1] • t2) \ (u1 ∘ u2)
      using ind1 ind2 tu u
      by (cases u1) auto
  qed
qed

lemma prfx-Join:
shows Coinitial t u ⟹ u ≲ t ⊔ u
proof (induct t arbitrary: u)
  show ⋀u. Coinitial ♯ u ⟹ u ≲ ♯ ⊔ u
    by simp
  show ⋀x u. Coinitial «x» u ⟹ u ≲ «x» ⊔ u
    by auto
  fix t u
  assume ind: ⋀u. Coinitial t u ⟹ u ≲ t ⊔ u
  assume tu: Coinitial λ[t] u
  show u ≲ λ[t] ⊔ u
    using tu ind
    apply (cases u, auto)
    by force
  next
  fix t1 t2 u
```

assume *ind1*: $\bigwedge u1.\ Coinitial\ t1\ u1 \Longrightarrow u1 \lesssim t1 \sqcup u1$
assume *ind2*: $\bigwedge u2.\ Coinitial\ t2\ u2 \Longrightarrow u2 \lesssim t2 \sqcup u2$
assume *tu*: $Coinitial\ (t1 \circ t2)\ u$
show $u \lesssim t1 \circ t2 \sqcup u$
using *tu ind1 ind2 Coinitial-iff-Con*
apply (*cases u*, *simp-all*)
apply (*metis Ide.simps*(*1*))
proof −
fix *u1 u2*
assume *u*: $u = \boldsymbol{\lambda}[u1] \bullet u2$
assume *1*: $Arr\ t1 \wedge Arr\ t2 \wedge Arr\ u1 \wedge Arr\ u2 \wedge Src\ t1 = \boldsymbol{\lambda}[Src\ u1] \wedge Src\ t2 = Src\ u2$
have *2*: $u1 \frown un\text{-}Lam\ t1 \sqcup u1$
by (*metis 1 Coinitial-iff-Con Con-implies-is-Lam-iff-is-Lam Con-Arr-Src*(*2*)
lambda.collapse(*2*) *lambda.disc*(*8*) *resid.simps*(*2*) *resid-Join*)
have *3*: $u2 \frown t2 \sqcup u2$
by (*metis 1 conE ind2 null-char prfx-implies-con*)
show $Ide\ ((\boldsymbol{\lambda}[u1] \bullet u2) \setminus (t1 \circ t2 \sqcup \boldsymbol{\lambda}[u1] \bullet u2))$
using *u tu 1 2 3 ind1 ind2*
apply (*cases t1*, *simp-all*)
by (*metis Arr.simps*(*3*) *Ide.simps*(*3*) *Ide-Subst Join.simps*(*2*) *Src.simps*(*3*) *resid.simps*(*2*))
qed
next
fix *t1 t2 u*
assume *ind1*: $\bigwedge u1.\ Coinitial\ t1\ u1 \Longrightarrow u1 \lesssim t1 \sqcup u1$
assume *ind2*: $\bigwedge u2.\ Coinitial\ t2\ u2 \Longrightarrow u2 \lesssim t2 \sqcup u2$
assume *tu*: $Coinitial\ (\boldsymbol{\lambda}[t1] \bullet t2)\ u$
show $u \lesssim (\boldsymbol{\lambda}[t1] \bullet t2) \sqcup u$
using *tu ind1 ind2 Coinitial-iff-Con*
apply (*cases u*, *simp-all*)
apply (*cases un-App1 u*, *simp-all*)
by (*metis Ide.simps*(*1*) *Ide-Subst*)+
qed

lemma *Ide-resid-Join*:
shows $Coinitial\ t\ u \Longrightarrow Ide\ (u \setminus (t \sqcup u))$
using *ide-char prfx-Join* **by** *blast*

lemma *join-of-Join*:
assumes *Coinitial t u*
shows *join-of t u* $(t \sqcup u)$
proof (*unfold join-of-def composite-of-def*, *intro conjI*)
show $t \lesssim t \sqcup u$
using *assms Join-sym prfx-Join* [*of u t*] **by** *simp*
show $u \lesssim t \sqcup u$
using *assms Ide-resid-Join ide-char* **by** *simp*
show $(t \sqcup u) \setminus t \lesssim u \setminus t$
by (*metis* ‹*prfx u* (*Join t u*)› *arr-char assms cong-subst-right*(*2*) *prfx-implies-con*
prfx-reflexive resid-Join con-sym cube)
show $u \setminus t \lesssim (t \sqcup u) \setminus t$

by (*metis Coinitial-resid-resid* ‹*prfx t* (*Join t u*)› ‹*prfx u* (*Join t u*)› *conE ide-char null-char prfx-implies-con resid-Ide-Arr cube*)
show $(t \sqcup u) \setminus u \lesssim t \setminus u$
using ‹$(t \sqcup u) \setminus t \lesssim u \setminus t$› *cube* **by** *auto*
show $t \setminus u \lesssim (t \sqcup u) \setminus u$
by (*metis* ‹$(t \sqcup u) \setminus t \lesssim u \setminus t$› *assms cube resid-Join*)
qed

sublocale *rts-with-joins resid*
using *join-of-Join*
apply *unfold-locales*
by (*metis Coinitial-iff-Con conE joinable-def null-char*)

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

3.2.5 Simulations from Syntactic Constructors

Here we show that the syntactic constructors *Lam* and *App*, as well as the substitution operation *subst*, determine simulations. In addition, we show that *Beta* determines a transformation from $App \circ (Lam \times Id)$ to *subst*.

abbreviation Lam_{ext}
where $Lam_{ext}\ t \equiv$ *if arr t then* $\boldsymbol{\lambda}[t]$ *else* $\sharp$

lemma *Lam-is-simulation*:
shows *simulation resid resid* Lam_{ext}
using *Arr-resid Coinitial-iff-Con*
by *unfold-locales auto*

interpretation *Lam*: *simulation resid resid* Lam_{ext}
using *Lam-is-simulation* **by** *simp*

interpretation $\Lambda x\Lambda$: *product-of-weakly-extensional-rts resid resid*
..

abbreviation App_{ext}
where $App_{ext}\ t \equiv$ *if* $\Lambda x\Lambda.arr\ t$ *then fst t* $\circ$ *snd t else* $\sharp$

lemma *App-is-binary-simulation*:
shows *binary-simulation resid resid resid* App_{ext}
proof
show $\bigwedge t.\ \neg\ \Lambda x\Lambda.arr\ t \Longrightarrow App_{ext}\ t = null$
by *auto*
show $\bigwedge t\ u.\ \Lambda x\Lambda.con\ t\ u \Longrightarrow con\ (App_{ext}\ t)\ (App_{ext}\ u)$
using $\Lambda x\Lambda.con\text{-}char$ *Coinitial-iff-Con* **by** *auto*
show $\bigwedge t\ u.\ \Lambda x\Lambda.con\ t\ u \Longrightarrow App_{ext}\ (\Lambda x\Lambda.resid\ t\ u) = App_{ext}\ t \setminus App_{ext}\ u$
using $\Lambda x\Lambda.arr\text{-}char$ $\Lambda x\Lambda.resid\text{-}def$
apply *simp*

by (*metis Arr-resid Con-implies-Arr1 Con-implies-Arr2*)
qed

interpretation *App*: *binary-simulation resid resid resid* App_{ext}
using *App-is-binary-simulation* **by** *simp*

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
show $\bigwedge t.\ \neg\ \Lambda x\Lambda.arr\ t \Longrightarrow subst_{ext}\ t = null$
by *auto*
show $\bigwedge t\ u.\ \Lambda x\Lambda.con\ t\ u \Longrightarrow con\ (subst_{ext}\ t)\ (subst_{ext}\ u)$
using $\Lambda x\Lambda.con\text{-}char$ *con-char Subst-not-Nil resid-Subst* $\Lambda x\Lambda.coinitialE$
$\Lambda x\Lambda.con\text{-}imp\text{-}coinitial$
apply *simp*
by *metis*
show $\bigwedge t\ u.\ \Lambda x\Lambda.con\ t\ u \Longrightarrow subst_{ext}\ (\Lambda x\Lambda.resid\ t\ u) = subst_{ext}\ t \setminus subst_{ext}\ u$
using $\Lambda x\Lambda.arr\text{-}char$ $\Lambda x\Lambda.resid\text{-}def$
apply *simp*
by (*metis Arr-resid Con-implies-Arr1 Con-implies-Arr2 resid-Subst*)
qed

interpretation *subst*: *binary-simulation resid resid resid* $subst_{ext}$
using *subst-is-binary-simulation* **by** *simp*

interpretation *Id*: *identity-simulation resid*
..
interpretation *Lam-Id*: *product-simulation resid resid resid resid* Lam_{ext} *Id.map*
..
interpretation *App-o-Lam-Id*: *composite-simulation* $\Lambda x\Lambda.resid$ $\Lambda x\Lambda.resid$ *resid Lam-Id.map* App_{ext}
..

abbreviation $Beta_{ext}$
where $Beta_{ext}\ t \equiv$ *if* $\Lambda x\Lambda.arr\ t$ *then* $\boldsymbol{\lambda}$[*fst t*] • *snd t else* ♯

lemma *Beta-is-transformation*:
shows *transformation* $\Lambda x\Lambda.resid$ *resid App-o-Lam-Id.map* $subst_{ext}$ $Beta_{ext}$
proof
show $\bigwedge a\ a'.\ [\![\Lambda x\Lambda.ide\ a;\ \Lambda x\Lambda.cong\ a\ a']\!] \Longrightarrow Beta_{ext}\ a = Beta_{ext}\ a'$
using $\Lambda x\Lambda.ide\text{-}backward\text{-}stable$ $\Lambda x\Lambda.weak\text{-}extensionality$ **by** *blast*
show $\bigwedge f.\ \neg\ \Lambda x\Lambda.arr\ f \Longrightarrow Beta_{ext}\ f = null$
by *simp*
show $\bigwedge f.\ \Lambda x\Lambda.ide\ f \Longrightarrow src\ (Beta_{ext}\ f) = App\text{-}o\text{-}Lam\text{-}Id.map\ f$
using $\Lambda x\Lambda.src\text{-}char$ $\Lambda x\Lambda.src\text{-}ide$ *Lam-Id.map-def* **by** *force*
show $\bigwedge f.\ \Lambda x\Lambda.ide\ f \Longrightarrow trg\ (Beta_{ext}\ f) = subst_{ext}\ f$

```
    using ΛxΛ.trg-char ΛxΛ.trg-ide by force
  show ⋀a f. a ∈ ΛxΛ.sources f ⟹
            Beta_ext a \ App-o-Lam-Id.map f = Beta_ext (ΛxΛ.resid a f)
  proof −
    fix a f
    assume a ∈ ΛxΛ.sources f
    hence f: ΛxΛ.arr f ∧ a = ΛxΛ.src f
      using ΛxΛ.sources-char_WE by simp
    have Beta_ext (ΛxΛ.src f) \ App-o-Lam-Id.map f = Beta_ext (ΛxΛ.trg f)
      using f ΛxΛ.src-char ΛxΛ.trg-char Arr-Trg Arr-not-Nil Lam-Id.map-def by simp
    thus Beta_ext a \ App-o-Lam-Id.map f = Beta_ext (ΛxΛ.resid a f)
      using f by auto
  qed
  show ⋀a f. a ∈ ΛxΛ.sources f ⟹ App-o-Lam-Id.map f \ Beta_ext a = subst_ext f
     using ΛxΛ.src-char ΛxΛ.trg-char Lam-Id.map-def ΛxΛ.sources-char by auto
  show ⋀a f. a ∈ ΛxΛ.sources f ⟹ join-of (Beta_ext a) (App-o-Lam-Id.map f) (Beta_ext f)
  proof −
    fix a f
    assume a ∈ ΛxΛ.sources f
    hence f: ΛxΛ.arr f ∧ a = ΛxΛ.src f
      using ΛxΛ.sources-char_WE by auto
    show join-of (Beta_ext a) (App-o-Lam-Id.map f) (Beta_ext f)
    proof (intro join-ofI composite-ofI)
      show App-o-Lam-Id.map f ≲ Beta_ext f
        using f Lam-Id.map-def Ide-Subst arr-char prfx-char prfx-reflexive
        by auto
      show Beta_ext f \ Beta_ext a ∼ App-o-Lam-Id.map f \ Beta_ext a
        using f Lam-Id.map-def ΛxΛ.src-char trg-char trg-def ΛxΛ.sources-char
        apply auto
        by (metis Arr-Subst Ide-Trg)
      show 1: Beta_ext f \ App-o-Lam-Id.map f ∼ Beta_ext a \ App-o-Lam-Id.map f
        using f Lam-Id.map-def Ide-Subst ΛxΛ.src-char Ide-Trg Arr-resid Coinitial-iff-Con
              resid-Arr-self
        apply simp
        by metis
      show Beta_ext a ≲ Beta_ext f
        using f 1 Lam-Id.map-def Ide-Subst ΛxΛ.src-char resid-Arr-self ΛxΛ.sources-char
        by auto
    qed
  qed
qed
```

The next two results are used to show that mapping App over lists of transitions preserves paths.

```
lemma App-is-simulation1:
assumes ide a
shows simulation resid resid (λt. if arr t then t ∘ a else ♯)
proof −
  have (λt. if ΛxΛ.arr (t, a) then fst (t, a) ∘ snd (t, a) else ♯) =
```

```
      (λt. if arr t then t ∘ a else ♯)
    using assms ide-implies-arr by force
  thus ?thesis
    using assms App.fixing-ide-gives-simulation-0 [of a] by auto
qed

lemma App-is-simulation2:
assumes ide a
shows simulation resid resid (λt. if arr t then a ∘ t else ♯)
proof −
  have (λt. if ΛxΛ.arr (a, t) then fst (a, t) ∘ snd (a, t) else ♯) =
      (λt. if arr t then a ∘ t else ♯)
    using assms ide-implies-arr by force
  thus ?thesis
    using assms App.fixing-ide-gives-simulation-1 [of a] by auto
qed
```

3.2.6 Reduction and Conversion

Here we define the usual relations of reduction and conversion. Reduction is the least transitive relation that relates a to b if there exists an arrow t having a as its source and b as its target. Conversion is the least transitive relation that relates a to b if there exists an arrow t in either direction between a and b.

```
inductive red
where Arr t ⟹ red (Src t) (Trg t)
  | ⟦red a b; red b c⟧ ⟹ red a c

inductive cnv
where Arr t ⟹ cnv (Src t) (Trg t)
  | Arr t ⟹ cnv (Trg t) (Src t)
  | ⟦cnv a b; cnv b c⟧ ⟹ cnv a c

lemma cnv-refl:
assumes Ide a
shows cnv a a
  using assms
  by (metis Ide-iff-Src-self Ide-implies-Arr cnv.simps)

lemma cnv-sym:
shows cnv a b ⟹ cnv b a
  apply (induct rule: cnv.induct)
  using cnv.intros(1−2)
    apply auto[2]
  using cnv.intros(3) by blast

lemma red-imp-cnv:
shows red a b ⟹ cnv a b
  using cnv.intros(1,3) red.inducts by blast
```

end

We now define a locale that extends the residuation operation defined above to paths, using general results that have already been shown for paths in an RTS. In particular, we are taking advantage of the general proof of the Cube Lemma for residuation on paths.

Our immediate goal is to prove the Church-Rosser theorem, so we first prove a lemma that connects the reduction relation to paths. Later, we will prove many more facts in this locale, thereby developing a general framework for reasoning about reduction paths in the λ-calculus.

locale *reduction-paths* =
  Λ: *lambda-calculus*
begin

  sublocale Λ: *rts* Λ.*resid*
    by (*simp add*: Λ.*is-rts-with-joins rts-with-joins.axioms*(*1*))
  sublocale *paths-in-weakly-extensional-rts* Λ.*resid*
    ..
  sublocale *paths-in-confluent-rts* Λ.*resid*
    using *confluent-rts.axioms*(*1*) Λ.*is-confluent-rts paths-in-rts-def*
        paths-in-confluent-rts.intro
    by *blast*

  notation Λ.*resid* (**infix** $\langle\backslash\rangle$ *70*)
  notation Λ.*con* (**infix** $\langle\frown\rangle$ *50*)
  notation Λ.*prfx* (**infix** $\langle\lesssim\rangle$ *50*)
  notation Λ.*cong* (**infix** $\langle\sim\rangle$ *50*)

  notation *Resid* (**infix** $\langle^*\backslash^*\rangle$ *70*)
  notation *Resid1x* (**infix** $\langle^1\backslash^*\rangle$ *70*)
  notation *Residx1* (**infix** $\langle^*\backslash^1\rangle$ *70*)
  notation *con* (**infix** $\langle^*\frown^*\rangle$ *50*)
  notation *prfx* (**infix** $\langle^*\lesssim^*\rangle$ *50*)
  notation *cong* (**infix** $\langle^*\sim^*\rangle$ *50*)

  lemma *red-iff*:
  shows $\Lambda.red\ a\ b \longleftrightarrow (\exists\, T.\ Arr\ T \wedge Src\ T = a \wedge Trg\ T = b)$
  proof
    show $\Lambda.red\ a\ b \Longrightarrow \exists\, T.\ Arr\ T \wedge Src\ T = a \wedge Trg\ T = b$
    proof (*induct rule*: Λ.*red.induct*)
      show $\bigwedge t.\ \Lambda.Arr\ t \Longrightarrow \exists\, T.\ Arr\ T \wedge Src\ T = \Lambda.Src\ t \wedge Trg\ T = \Lambda.Trg\ t$
        by (*metis Arr.simps*(*2*) *Srcs.simps*(*2*) *Srcs-simp*$_{PWE}$ *Trg.simps*(*2*) Λ.*trg-def*
            Λ.*arr-char* Λ.*resid-Arr-self* Λ.*sources-char*$_\Lambda$ *singleton-insert-inj-eq'*)
      show $\bigwedge a\ b\ c.\ [\![\exists\, T.\ Arr\ T \wedge Src\ T = a \wedge Trg\ T = b;$
                  $\exists\, T.\ Arr\ T \wedge Src\ T = b \wedge Trg\ T = c]\!]$
                    $\Longrightarrow \exists\, T.\ Arr\ T \wedge Src\ T = a \wedge Trg\ T = c$
        by (*metis Arr.simps*(*1*) *Arr-appendI*$_{PWE}$ *Srcs-append Srcs-simp*$_{PWE}$ *Trgs-append*
            Trgs-simp$_{PWE}$ *singleton-insert-inj-eq'*)
    qed
    show $\exists\, T.\ Arr\ T \wedge Src\ T = a \wedge Trg\ T = b \Longrightarrow \Lambda.red\ a\ b$

```
    proof −
      have Arr T ⟹ Λ.red (Src T) (Trg T) for T
      proof (induct T)
        show Arr [] ⟹ Λ.red (Src []) (Trg [])
          by auto
        fix t T
        assume ind: Arr T ⟹ Λ.red (Src T) (Trg T)
        assume Arr: Arr (t # T)
        show Λ.red (Src (t # T)) (Trg (t # T))
        proof (cases T = [])
          show T = [] ⟹ ?thesis
            using Arr arr-char Λ.red.intros(1) by simp
          assume T: T ≠ []
          have Λ.red (Src (t # T)) (Λ.Trg t)
            apply simp
            by (meson Arr Arr.simps(2) Con-Arr-self Con-implies-Arr(1) Con-initial-left
                Λ.arr-char Λ.red.intros(1))
          moreover have Λ.Trg t = Src T
            using Arr
            by (metis Arr.elims(2) Srcs-simp_PWE T Λ.arr-iff-has-target insert-subset
                Λ.targets-char_Λ list.sel(1) list.sel(3) singleton-iff)
          ultimately show ?thesis
            using ind
            by (metis (no-types, opaque-lifting) Arr Con-Arr-self Con-implies-Arr(2)
                Resid-cons(2) T Trg.simps(3) Λ.red.intros(2) neq-Nil-conv)
        qed
      qed
      thus ∃ T. Arr T ∧ Src T = a ∧ Trg T = b ⟹ Λ.red a b
        by blast
    qed
  qed

end
```

3.2.7 The Church-Rosser Theorem

```
  context lambda-calculus
  begin

    interpretation Λx: reduction-paths .

    theorem church-rosser:
    shows cnv a b ⟹ ∃ c. red a c ∧ red b c
    proof (induct rule: cnv.induct)
      show ⋀t. Arr t ⟹ ∃ c. red (Src t) c ∧ red (Trg t) c
        by (metis Ide-Trg Ide-iff-Src-self Ide-iff-Trg-self Ide-implies-Arr red.intros(1))
      thus ⋀t. Arr t ⟹ ∃ c. red (Trg t) c ∧ red (Src t) c
        by auto
      show ⋀a b c. ⟦cnv a b; cnv b c; ∃ x. red a x ∧ red b x; ∃ y. red b y ∧ red c y⟧
```

```
                ⟹ ∃ z. red a z ∧ red c z
  proof −
    fix a b c
    assume ind1: ∃ x. red a x ∧ red b x and ind2: ∃ y. red b y ∧ red c y
    obtain x where x: red a x ∧ red b x
      using ind1 by blast
    obtain y where y: red b y ∧ red c y
      using ind2 by blast
    obtain T1 U1 where 1: Λx.Arr T1 ∧ Λx.Arr U1 ∧ Λx.Src T1 = a ∧ Λx.Src U1 = b ∧
                          Λx.Trgs T1 = Λx.Trgs U1
      using x Λx.red-iff [of a x] Λx.red-iff [of b x] by fastforce
    obtain T2 U2 where 2: Λx.Arr T2 ∧ Λx.Arr U2 ∧ Λx.Src T2 = b ∧ Λx.Src U2 = c ∧
                          Λx.Trgs T2 = Λx.Trgs U2
      using y Λx.red-iff [of b y] Λx.red-iff [of c y] by fastforce
    show ∃ e. red a e ∧ red c e
    proof −
      let ?T = T1 @ (Λx.Resid T2 U1) and ?U = U2 @ (Λx.Resid U1 T2)
      have 3: Λx.Arr ?T ∧ Λx.Arr ?U ∧ Λx.Src ?T = a ∧ Λx.Src ?U = c
        using 1 2
        by (metis Λx.Arr-appendI_PWE Λx.Arr-has-Trg Λx.Con-imp-Arr-Resid Λx.Src-append
            Λx.Src-resid Λx.Srcs-simp_PWE Λx.Trgs.simps(1) Λx.Trgs-simp_PWE Λx.arrI_P
            Λx.arr-append-imp-seq Λx.confluence-ind singleton-insert-inj-eq')
      moreover have Λx.Trgs ?T = Λx.Trgs ?U
        using 1 2 3 Λx.Srcs-simp_PWE Λx.Trgs-Resid-sym Λx.Trgs-append Λx.confluence-ind
        by presburger
      ultimately have ∃ T U. Λx.Arr T ∧ Λx.Arr U ∧ Λx.Src T = a ∧ Λx.Src U = c ∧
                             Λx.Trgs T = Λx.Trgs U
        by blast
      thus ?thesis
        using Λx.red-iff Λx.Arr-has-Trg by fastforce
    qed
  qed
qed

corollary weak-diamond:
assumes red a b and red a b'
obtains c where red b c and red b' c
proof −
  have cnv b b'
    using assms
    by (metis cnv.intros(1,3) cnv-sym red.induct)
  thus ?thesis
    using that church-rosser by blast
qed
```

As a consequence of the Church-Rosser Theorem, the collection of all reduction paths forms a coherent normal sub-RTS of the RTS of reduction paths, and on identities the congruence induced by this normal sub-RTS coincides with convertibility. The quotient of the λ-calculus RTS by this congruence is then obviously discrete: the only transitions

are identities.

interpretation *Red*: *normal-sub-rts* $\Lambda x.\, Resid\ \langle Collect\ \Lambda x.\, Arr\rangle$
proof
show $\bigwedge t.\ t \in Collect\ \Lambda x.\, Arr \Longrightarrow \Lambda x.\, arr\ t$
by *blast*
show $\bigwedge a.\ \Lambda x.\, ide\ a \Longrightarrow a \in Collect\ \Lambda x.\, Arr$
using $\Lambda x.\, Ide\text{-}char\ \Lambda x.\, ide\text{-}char$ **by** *blast*
show $\bigwedge u\ t.\ [\![u \in Collect\ \Lambda x.\, Arr;\ \Lambda x.\, coinitial\ t\ u]\!] \Longrightarrow \Lambda x.\, Resid\ u\ t \in Collect\ \Lambda x.\, Arr$
by (*metis* $\Lambda x.\, Con\text{-}imp\text{-}Arr\text{-}Resid\ \Lambda x.\, Resid.simps(1)\ \Lambda x.\, con\text{-}sym\ \Lambda x.\, confluence_P\ \Lambda x.\, ide\text{-}def$
$\langle\bigwedge a.\ \Lambda x.\, ide\ a \Longrightarrow a \in Collect\ \Lambda x.\, Arr\rangle$ *mem-Collect-eq* $\Lambda x.\, arr\text{-}resid\text{-}iff\text{-}con$)
show $\bigwedge u\ t.\ [\![u \in Collect\ \Lambda x.\, Arr;\ \Lambda x.\, Resid\ t\ u \in Collect\ \Lambda x.\, Arr]\!] \Longrightarrow t \in Collect\ \Lambda x.\, Arr$
by (*metis* $\Lambda x.\, Arr.simps(1)\ \Lambda x.\, Con\text{-}implies\text{-}Arr(1)$ *mem-Collect-eq*)
show $\bigwedge u\ t.\ [\![u \in Collect\ \Lambda x.\, Arr;\ \Lambda x.\, seq\ u\ t]\!] \Longrightarrow \exists v.\ \Lambda x.\, composite\text{-}of\ u\ t\ v$
by (*meson* $\Lambda x.\, obtains\text{-}composite\text{-}of$)
show $\bigwedge u\ t.\ [\![u \in Collect\ \Lambda x.\, Arr;\ \Lambda x.\, seq\ t\ u]\!] \Longrightarrow \exists v.\ \Lambda x.\, composite\text{-}of\ t\ u\ v$
by (*meson* $\Lambda x.\, obtains\text{-}composite\text{-}of$)
qed

interpretation *Red*: *coherent-normal-sub-rts* $\Lambda x.\, Resid\ \langle Collect\ \Lambda x.\, Arr\rangle$
apply *unfold-locales*
by (*metis Red.Cong-closure-props*(4) *Red.Cong-imp-arr*(2) $\Lambda x.\, Con\text{-}imp\text{-}Arr\text{-}Resid$
$\Lambda x.\, arr\text{-}resid\text{-}iff\text{-}con\ \Lambda x.\, con\text{-}char\ \Lambda x.\, sources\text{-}resid$ *mem-Collect-eq*)

lemma *cnv-iff-Cong*:
assumes *ide a* **and** *ide b*
shows *cnv a b* $\longleftrightarrow$ *Red.Cong* [*a*] [*b*]
proof
assume *1*: *Red.Cong* [*a*] [*b*]
obtain *U V*
where *UV*: $\Lambda x.\, Arr\ U \wedge \Lambda x.\, Arr\ V \wedge Red.Cong_0\ (\Lambda x.\, Resid\ [a]\ U)\ (\Lambda x.\, Resid\ [b]\ V)$
using *1 Red.Cong-def* [*of* [*a*] [*b*]] **by** *blast*
have *red a* $(\Lambda x.\, Trg\ U) \wedge$ *red b* $(\Lambda x.\, Trg\ V)$
by (*metis UV* $\Lambda x.\, Arr.simps(1)\ \Lambda x.\, Con\text{-}implies\text{-}Arr(1)\ \Lambda x.\, Resid\text{-}single\text{-}ide(2)\ \Lambda x.\, Src\text{-}resid$
$\Lambda x.\, Trg.simps(2)$ *assms*(1−2) *mem-Collect-eq reduction-paths.red-iff trg-ide*)
moreover have $\Lambda x.\, Trg\ U = \Lambda x.\, Trg\ V$
using *UV*
by (*metis* (*no-types*, *lifting*) *Red.Cong*$_0$*-imp-con* $\Lambda x.\, Arr.simps(1)\ \Lambda x.\, Con\text{-}Arr\text{-}self$
$\Lambda x.\, Con\text{-}implies\text{-}Arr(1)\ \Lambda x.\, Resid\text{-}single\text{-}ide(2)\ \Lambda x.\, Src\text{-}resid\ \Lambda x.\, cube\ \Lambda x.\, ide\text{-}def$
$\Lambda x.\, resid\text{-}arr\text{-}ide$ *assms*(1) *mem-Collect-eq*)
ultimately show *cnv a b*
by (*metis cnv-sym cnv.intros*(3) *red-imp-cnv*)
next
assume *1*: *cnv a b*
obtain *c* **where** *c*: *red a c* $\wedge$ *red b c*
using *1 church-rosser* **by** *blast*
obtain *U* **where** *U*: $\Lambda x.\, Arr\ U \wedge \Lambda x.\, Src\ U = a \wedge \Lambda x.\, Trg\ U = c$
using *c* $\Lambda x.\, red\text{-}iff$ **by** *blast*
obtain *V* **where** *V*: $\Lambda x.\, Arr\ V \wedge \Lambda x.\, Src\ V = b \wedge \Lambda x.\, Trg\ V = c$
using *c* $\Lambda x.\, red\text{-}iff$ **by** *blast*

```
  have Λx.Resid1x a U = c ∧ Λx.Resid1x b V = c
    by (metis U V Λx.Con-single-ide-ind Λx.Ide.simps(2) Λx.Resid1x-as-Resid
        Λx.Resid-Ide-Arr-ind Λx.Resid-single-ide(2) Λx.Srcs-simp_PWE Λx.Trg.simps(2)
        Λx.Trg-resid-sym Λx.ex-un-Src assms(1−2) singletonD trg-ide)
  hence Red.Cong_0 (Λx.Resid [a] U) (Λx.Resid [b] V)
    by (metis Red.Cong_0-reflexive U V Λx.Con-single-ideI(1) Λx.Resid1x-as-Resid
        Λx.Srcs-simp_PWE Λx.arr-resid Λx.con-char assms(1−2) empty-set
        list.set-intros(1) list.simps(15))
  thus Red.Cong [a] [b]
    using U V Red.Cong-def [of [a] [b]] by blast
qed

interpretation Λq: quotient-by-coherent-normal Λx.Resid ‹Collect Λx.Arr›
  ..

lemma quotient-by-cnv-is-discrete:
shows Λq.arr t ⟷ Λq.ide t
  by (metis Red.Cong-class-memb-is-arr Λq.arr-char Λq.ide-char' Λx.arr-char
      mem-Collect-eq subsetI)
```

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 ≡ Ide a ∧ (∀ t. Arr t ∧ Src t = a ⟶ Ide t)

lemma (in reduction-paths) path-from-NF-is-Ide:
assumes Λ.NF a
shows ⟦Arr U; Src U = a⟧ ⟹ Ide U
proof (induct U, simp)
  fix u U
  assume ind: ⟦Arr U; Src U = a⟧ ⟹ Ide U
  assume uU: Arr (u # U) and a: Src (u # U) = a
  have Λ.Ide u
    using assms a Λ.NF-def uU by force
  thus Ide (u # U)
    using a uU ind
    by (metis Arr-consE Con-Arr-self Con-imp-eq-Srcs Con-initial-right Ide.simps(2)
          Ide-consI Srcs.simps(2) Srcs-simp_PWE Λ.Ide-iff-Src-self Λ.Ide-implies-Arr
          Λ.sources-char_Λ Λ.trg-ide Λ.ide-char
          singleton-insert-inj-eq)
qed

lemma NF-reduct-is-trivial:
assumes NF a and red a b
shows a = b
proof −
  interpret Λx: reduction-paths .
  have ⋀U. ⟦Λx.Arr U; a ∈ Λx.Srcs U⟧ ⟹ Λx.Ide U
```

using *assms* $\Lambda x.$*path-from-NF-is-Ide*
by (*simp add*: $\Lambda x.$*Srcs-simp*$_{PWE}$)
thus *?thesis*
using *assms* $\Lambda x.$*red-iff*
by (*metis* $\Lambda x.$*Con-Arr-self* $\Lambda x.$*Resid-Arr-Ide-ind* $\Lambda x.$*Src-resid* $\Lambda x.$*path-from-NF-is-Ide*)
qed

lemma *NF-unique*:
assumes *red t u* **and** *red t u'* **and** *NF u* **and** *NF u'*
shows $u = u'$
using *assms weak-diamond NF-reduct-is-trivial* **by** *metis*

A term is *normalizable* if it is an identity that is reducible to a normal form.

definition *normalizable*
where *normalizable a* $\equiv$ *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* = {Λ.*Src* (*hd t*)}
by (*metis Arr-has-Src Srcs.elims list.sel*(*1*) Λ.*sources-char*$_\Lambda$)

lemma *Trgs-simp*$_{\Lambda P}$:
shows *Arr t* $\Longrightarrow$ *Trgs t* = {Λ.*Trg* (*last t*)}
by (*metis Arr.simps*(*1*) *Arr-has-Trg Trgs.simps*(*2*) *Trgs-append*
append-butlast-last-id not-Cons-self2 Λ.*targets-char*$_\Lambda$)

lemma *sources-single-Src* [*simp*]:
assumes Λ.*Arr t*
shows *sources* [Λ.*Src t*] = *sources* [*t*]
using *assms*
by (*metis* Λ.*Con-Arr-Src*(*1*) Λ.*Ide-Src Ide.simps*(*2*) *Resid.simps*(*3*) *con-char ideE*
ide-char sources-resid Λ.*con-char* Λ.*ide-char list.discI* Λ.*resid-Arr-Src*)

lemma *targets-single-Trg* [*simp*]:
assumes Λ.*Arr t*
shows *targets* [Λ.*Trg t*] = *targets* [*t*]
using *assms*
by (*metis* (*full-types*) *Resid.simps*(*3*) *conI*$_P$ Λ.*Arr-Trg* Λ.*arr-char* Λ.*resid-Arr-Src*
Λ.*resid-Src-Arr* Λ.*arr-resid-iff-con targets-resid-sym*)

lemma *sources-single-Trg* [*simp*]:
assumes $\Lambda.Arr\ t$
shows *sources* $[\Lambda.Trg\ t] = targets\ [t]$
using *assms*
by (*metis* $\Lambda.Ide\text{-}Trg$ *Ide.simps*(*2*) *ideE ide-char sources-resid* $\Lambda.ide\text{-}char$ *targets-single-Trg*)

lemma *targets-single-Src* [*simp*]:
assumes $\Lambda.Arr\ t$
shows *targets* $[\Lambda.Src\ t] = sources\ [t]$
using *assms*
by (*metis* $\Lambda.Arr\text{-}Src$ $\Lambda.Trg\text{-}Src$ *sources-single-Src sources-single-Trg*)

lemma *single-Src-hd-in-sources*:
assumes *Arr T*
shows $[\Lambda.Src\ (hd\ T)] \in sources\ T$
using *assms*
by (*metis Arr.simps*(*1*) *Arr-has-Src Ide.simps*(*2*) *Resid-Arr-Src* $Srcs\text{-}simp_P$ $\Lambda.source\text{-}is\text{-}ide$ $conI_P$ *empty-set ide-char in-sourcesI* $\Lambda.sources\text{-}char_\Lambda$ *list.set-intros*(*1*) *list.simps*(*15*))

lemma *single-Trg-last-in-targets*:
assumes *Arr T*
shows $[\Lambda.Trg\ (last\ T)] \in targets\ T$
using *assms* $targets\text{-}char_P$ *Arr-imp-arr-last* $Trgs\text{-}simp_{\Lambda P}$ $\Lambda.Ide\text{-}Trg$ **by** *fastforce*

lemma *in-sources-iff*:
assumes *Arr T*
shows $A \in sources\ T \longleftrightarrow A\ {}^*\!\sim^*\ [\Lambda.Src\ (hd\ T)]$
using *assms*
by (*meson single-Src-hd-in-sources sources-are-cong sources-cong-closed*)

lemma *in-targets-iff*:
assumes *Arr T*
shows $B \in targets\ T \longleftrightarrow B\ {}^*\!\sim^*\ [\Lambda.Trg\ (last\ T)]$
using *assms*
by (*meson single-Trg-last-in-targets targets-are-cong targets-cong-closed*)

lemma *seq-imp-cong-Trg-last-Src-hd*:
assumes *seq T U*
shows $\Lambda.Trg\ (last\ T) \sim \Lambda.Src\ (hd\ U)$
using *assms Arr-imp-arr-hd Arr-imp-arr-last* $Srcs\text{-}simp_{PWE}$ $Trgs\text{-}simp_{PWE}$ $\Lambda.cong\text{-}reflexive$ *seq-char*
by (*metis* $Srcs\text{-}simp_{\Lambda P}$ $Trgs\text{-}simp_{\Lambda P}$ $\Lambda.Arr\text{-}Trg$ $\Lambda.arr\text{-}char$ *singleton-inject*)

lemma $sources\text{-}char_{\Lambda P}$:
shows $sources\ T = \{A.\ Arr\ T \wedge A\ {}^*\!\sim^*\ [\Lambda.Src\ (hd\ T)]\}$
using *in-sources-iff arr-char* $sources\text{-}char_P$ **by** *auto*

lemma *targets-char*$_{\Lambda P}$:
shows *targets T* = {*B*. *Arr T* ∧ *B* $^*\sim^*$ [Λ.*Trg* (*last T*)]}
using *in-targets-iff arr-char targets-char* **by** *auto*

lemma *Src-hd-eqI*:
assumes *T* $^*\sim^*$ *U*
shows Λ.*Src* (*hd T*) = Λ.*Src* (*hd U*)
using *assms*
by (*metis Con-imp-eq-Srcs Con-implies-Arr*(*1*) *Ide.simps*(*1*) *Srcs-simp*$_{\Lambda P}$ *ide-char singleton-insert-inj-eq*′)

lemma *Trg-last-eqI*:
assumes *T* $^*\sim^*$ *U*
shows Λ.*Trg* (*last T*) = Λ.*Trg* (*last U*)
proof −
have *1*: [Λ.*Trg* (*last T*)] ∈ *targets T* ∧ [Λ.*Trg* (*last U*)] ∈ *targets U*
using *assms*
by (*metis Con-implies-Arr*(*1*) *Ide.simps*(*1*) *ide-char single-Trg-last-in-targets*)
have Λ.*cong* (Λ.*Trg* (*last T*)) (Λ.*Trg* (*last U*))
by (*metis 1 Ide.simps*(*2*) *Resid.simps*(*3*) *assms con-char cong-implies-coterminal coterminal-iff ide-char prfx-implies-con targets-are-cong*)
moreover have Λ.*Ide* (Λ.*Trg* (*last T*)) ∧ Λ.*Ide* (Λ.*Trg* (*last U*))
using *1 Ide.simps*(*2*) *ide-char* **by** *blast*
ultimately show *?thesis*
using Λ.*weak-extensionality* **by** *blast*
qed

lemma *Trg-last-Src-hd-eqI*:
assumes *seq T U*
shows Λ.*Trg* (*last T*) = Λ.*Src* (*hd U*)
using *assms Arr-imp-arr-hd Arr-imp-arr-last* Λ.*Ide-Src* Λ.*weak-extensionality* Λ.*Ide-Trg seq-char seq-imp-cong-Trg-last-Src-hd*
by *force*

lemma *seqI*$_{\Lambda P}$ [*intro*]:
assumes *Arr T* **and** *Arr U* **and** Λ.*Trg* (*last T*) = Λ.*Src* (*hd U*)
shows *seq T U*
by (*metis assms Arr-imp-arr-last Srcs-simp*$_{\Lambda P}$ Λ.*arr-char* Λ.*targets-char*$_{\Lambda}$ *Trgs-simp*$_P$ *seq-char*)

lemma *conI*$_{\Lambda P}$ [*intro*]:
assumes *arr T* **and** *arr U* **and** Λ.*Src* (*hd T*) = Λ.*Src* (*hd U*)
shows *T* $^*\frown^*$ *U*
using *assms*
by (*simp add*: *Srcs-simp*$_{\Lambda P}$ *arr-char con-char confluence-ind*)

3.3.2 Mapping Constructors over Paths

lemma *Arr-map-Lam*:
assumes *Arr T*
shows *Arr* (*map* Λ.*Lam T*)
proof −
  interpret *Lam*: *simulation* Λ.*resid* Λ.*resid* ‹λ*t*. *if* Λ.*arr t then* **λ**[*t*] *else* ♯›
    using Λ.*Lam-is-simulation* **by** *simp*
  interpret *simulation Resid Resid*
            ‹λ*T*. *if Arr T then map* (λ*t*. *if* Λ.*arr t then* **λ**[*t*] *else* ♯) *T else* []›
    using *assms Lam.lifts-to-paths* **by** *blast*
  have *map* (λ*t*. *if* Λ.*Arr t then* **λ**[*t*] *else* ♯) *T* = *map* Λ.*Lam T*
    using *assms set-Arr-subset-arr* **by** *fastforce*
  thus *?thesis*
    using *assms preserves-reflects-arr* [*of T*] *arr-char*
    by (*simp add*: ‹*map* (λ*t*. *if* Λ.*Arr t then* **λ**[*t*] *else* ♯) *T* = *map* Λ.*Lam T*›)
qed

lemma *Arr-map-App1*:
assumes Λ.*Ide b* **and** *Arr T*
shows *Arr* (*map* (λ*t*. *t* ⋅ *b*) *T*)
proof −
  interpret *App1*: *simulation* Λ.*resid* Λ.*resid* ‹λ*t*. *if* Λ.*arr t then t* ⋅ *b else* ♯›
    using *assms* Λ.*App-is-simulation1* [*of b*] **by** *simp*
  interpret *simulation Resid Resid*
            ‹λ*T*. *if Arr T then map* (λ*t*. *if* Λ.*arr t then t* ⋅ *b else* ♯) *T else* []›
    using *assms App1.lifts-to-paths* **by** *blast*
  have *map* (λ*t*. *if* Λ.*arr t then t* ⋅ *b else* ♯) *T* = *map* (λ*t*. *t* ⋅ *b*) *T*
    using *assms set-Arr-subset-arr* **by** *auto*
  thus *?thesis*
    using *assms preserves-reflects-arr arr-char*
    by (*metis* (*mono-tags*, *lifting*))
qed

lemma *Arr-map-App2*:
assumes Λ.*Ide a* **and** *Arr T*
shows *Arr* (*map* (Λ.*App a*) *T*)
proof −
  interpret *App2*: *simulation* Λ.*resid* Λ.*resid* ‹λ*u*. *if* Λ.*arr u then a* ⋅ *u else* ♯›
    using *assms* Λ.*App-is-simulation2* **by** *simp*
  interpret *simulation Resid Resid*
            ‹λ*T*. *if Arr T then map* (λ*u*. *if* Λ.*arr u then a* ⋅ *u else* ♯) *T else* []›
    using *assms App2.lifts-to-paths* **by** *blast*
  have *map* (λ*u*. *if* Λ.*arr u then a* ⋅ *u else* ♯) *T* = *map* (λ*u*. *a* ⋅ *u*) *T*
    using *assms set-Arr-subset-arr* **by** *auto*
  thus *?thesis*
    using *assms preserves-reflects-arr arr-char*
    by (*metis* (*mono-tags*, *lifting*))
qed

interpretation Λ_{Lam}: *source-replete-sub-rts* $\Lambda.resid$ ‹$\lambda t.\ \Lambda.Arr\ t \wedge \Lambda.is\text{-}Lam\ t$›
proof
 show $\bigwedge t.\ \Lambda.Arr\ t \wedge \Lambda.is\text{-}Lam\ t \Longrightarrow \Lambda.arr\ t$
  by *blast*
 show $\bigwedge t.\ \Lambda.Arr\ t \wedge \Lambda.is\text{-}Lam\ t \Longrightarrow \Lambda.sources\ t \subseteq \{t.\ \Lambda.Arr\ t \wedge \Lambda.is\text{-}Lam\ t\}$
  by *auto*
 show $[\![\Lambda.Arr\ t \wedge \Lambda.is\text{-}Lam\ t;\ \Lambda.Arr\ u \wedge \Lambda.is\text{-}Lam\ u;\ \Lambda.con\ t\ u]\!]$
            $\Longrightarrow \Lambda.Arr\ (t \setminus u) \wedge \Lambda.is\text{-}Lam\ (t \setminus u)$
       for *t u*
  apply (*cases t*; *cases u*)
                  apply *simp-all*
  using $\Lambda.Coinitial\text{-}resid\text{-}resid$
  by *presburger*
qed

interpretation *un-Lam*: *simulation* $\Lambda_{Lam}.resid$ $\Lambda.resid$
                    ‹$\lambda t.$ *if* $\Lambda_{Lam}.arr\ t$ *then* $\Lambda.un\text{-}Lam\ t$ *else* $\sharp$›
proof
 let *?un-Lam* = $\lambda t.$ *if* $\Lambda_{Lam}.arr\ t$ *then* $\Lambda.un\text{-}Lam\ t$ *else* $\sharp$
 show $\bigwedge t.\ \neg\ \Lambda_{Lam}.arr\ t \Longrightarrow$ *?un-Lam* $t = \Lambda.null$
  by *auto*
 show $\bigwedge t\ u.\ \Lambda_{Lam}.con\ t\ u \Longrightarrow \Lambda.con$ (*?un-Lam t*) (*?un-Lam u*)
  by (*auto simp add*: $\Lambda_{Lam}.con\text{-}char$)
 show $\bigwedge t\ u.\ \Lambda_{Lam}.con\ t\ u \Longrightarrow$ *?un-Lam* $(\Lambda_{Lam}.resid\ t\ u)$ = *?un-Lam* $t \setminus$ *?un-Lam* u
  using $\Lambda_{Lam}.resid\text{-}closed$ $\Lambda_{Lam}.resid\text{-}def$ **by** (*auto simp add*: $\Lambda_{Lam}.con\text{-}char$)
qed

lemma *Arr-map-un-Lam*:
assumes *Arr T* **and** *set T* $\subseteq$ *Collect* $\Lambda.is\text{-}Lam$
shows *Arr* (*map* $\Lambda.un\text{-}Lam$ *T*)
proof −
 have *map* ($\lambda t.$ *if* $\Lambda_{Lam}.arr\ t$ *then* $\Lambda.un\text{-}Lam\ t$ *else* $\sharp$) *T* = *map* $\Lambda.un\text{-}Lam$ *T*
  using *assms set-Arr-subset-arr* **by** (*auto simp add*: $\Lambda_{Lam}.arr\text{-}char$)
 thus *?thesis*
  using *assms*
  by (*metis* (*no-types*, *lifting*) $\Lambda_{Lam}.path\text{-}reflection$ $\Lambda.arr\text{-}char$ *mem-Collect-eq*
     set-Arr-subset-arr subset-code(*1*) *un-Lam.preserves-paths*)
qed

interpretation Λ_{App}: *source-replete-sub-rts* $\Lambda.resid$ ‹$\lambda t.\ \Lambda.Arr\ t \wedge \Lambda.is\text{-}App\ t$›
proof
 show $\bigwedge t.\ \Lambda.Arr\ t \wedge \Lambda.is\text{-}App\ t \Longrightarrow \Lambda.arr\ t$
  by *blast*
 show $\bigwedge t.\ \Lambda.Arr\ t \wedge \Lambda.is\text{-}App\ t \Longrightarrow \Lambda.sources\ t \subseteq \{t.\ \Lambda.Arr\ t \wedge \Lambda.is\text{-}App\ t\}$
  by *auto*
 show $[\![\Lambda.Arr\ t \wedge \Lambda.is\text{-}App\ t;\ \Lambda.Arr\ u \wedge \Lambda.is\text{-}App\ u;\ \Lambda.con\ t\ u]\!]$
        $\Longrightarrow \Lambda.Arr\ (t \setminus u) \wedge \Lambda.is\text{-}App\ (t \setminus u)$
     for *t u*
  using $\Lambda.Arr\text{-}resid$

```
  by (cases t; cases u) auto
qed

interpretation un-App1: simulation Λ_App.resid Λ.resid
                      ‹λt. if Λ_App.arr t then Λ.un-App1 t else ♯›
proof
  let ?un-App1 = λt. if Λ_App.arr t then Λ.un-App1 t else ♯
  show ⋀t. ¬ Λ_App.arr t ⟹ ?un-App1 t = Λ.null
    by auto
  show ⋀t u. Λ_App.con t u ⟹ Λ.con (?un-App1 t) (?un-App1 u)
    by (auto simp add: Λ_App.con-char)
  show Λ_App.con t u ⟹ ?un-App1 (Λ_App.resid t u) = ?un-App1 t \ ?un-App1 u
        for t u
    using Λ_App.resid-def Λ.Arr-resid
    by (cases t; cases u) (auto simp add: Λ_App.con-char)
qed

interpretation un-App2: simulation Λ_App.resid Λ.resid
                      ‹λt. if Λ_App.arr t then Λ.un-App2 t else ♯›
proof
  let ?un-App2 = λt. if Λ_App.arr t then Λ.un-App2 t else ♯
  show ⋀t. ¬ Λ_App.arr t ⟹ ?un-App2 t = Λ.null
    by auto
  show ⋀t u. Λ_App.con t u ⟹ Λ.con (?un-App2 t) (?un-App2 u)
    by (auto simp add: Λ_App.con-char)
  show Λ_App.con t u ⟹ ?un-App2 (Λ_App.resid t u) = ?un-App2 t \ ?un-App2 u
        for t u
    using Λ_App.resid-def Λ.Arr-resid
    by (cases t; cases u) (auto simp add: Λ_App.con-char)
qed

lemma Arr-map-un-App1:
assumes Arr T and set T ⊆ Collect Λ.is-App
shows Arr (map Λ.un-App1 T)
proof −
  interpret P_App: paths-in-rts Λ_App.resid
    ..
  interpret un-App1: simulation P_App.Resid Resid
                   ‹λT. if P_App.Arr T then
                          map (λt. if Λ_App.arr t then Λ.un-App1 t else ♯) T
                        else []›
    using un-App1.lifts-to-paths by simp
  have 1: map (λt. if Λ_App.arr t then Λ.un-App1 t else ♯) T = map Λ.un-App1 T
    using assms set-Arr-subset-arr by (auto simp add: Λ_App.arr-char)
  have 2: P_App.Arr T
    using assms set-Arr-subset-arr Λ_App.path-reflection [of T] by blast
  hence arr (if P_App.Arr T then map (λt. if Λ_App.arr t then Λ.un-App1 t else ♯) T else [])
    using un-App1.preserves-reflects-arr [of T] by blast
  hence Arr (if P_App.Arr T then map (λt. if Λ_App.arr t then Λ.un-App1 t else ♯) T else [])
```

```
    using arr-char by auto
  hence Arr (if P_App.Arr T then map Λ.un-App1 T else [])
    using 1 by metis
  thus ?thesis
    using 2 by simp
qed

lemma Arr-map-un-App2:
assumes Arr T and set T ⊆ Collect Λ.is-App
shows Arr (map Λ.un-App2 T)
proof −
  interpret P_App: paths-in-rts Λ_App.resid
    ..
  interpret un-App2: simulation P_App.Resid Resid
                     ‹λT. if P_App.Arr T then
                            map (λt. if Λ_App.arr t then Λ.un-App2 t else ♯) T
                          else []›
    using un-App2.lifts-to-paths by simp
  have 1: map (λt. if Λ_App.arr t then Λ.un-App2 t else ♯) T = map Λ.un-App2 T
    using assms set-Arr-subset-arr by (auto simp add: Λ_App.arr-char)
  have 2: P_App.Arr T
    using assms set-Arr-subset-arr Λ_App.path-reflection [of T] by blast
  hence arr (if P_App.Arr T then map (λt. if Λ_App.arr t then Λ.un-App2 t else ♯) T else [])
    using un-App2.preserves-reflects-arr [of T] by blast
  hence Arr (if P_App.Arr T then map (λt. if Λ_App.arr t then Λ.un-App2 t else ♯) T else [])
    using arr-char by blast
  hence Arr (if P_App.Arr T then map Λ.un-App2 T else [])
    using 1 by metis
  thus ?thesis
    using 2 by simp
qed

lemma map-App-map-un-App1:
shows ⟦Arr U; set U ⊆ Collect Λ.is-App; Λ.Ide b; Λ.un-App2 ‘ set U ⊆ {b}⟧ ⟹
        map (λt. Λ.App t b) (map Λ.un-App1 U) = U
  by (induct U) auto

lemma map-App-map-un-App2:
shows ⟦Arr U; set U ⊆ Collect Λ.is-App; Λ.Ide a; Λ.un-App1 ‘ set U ⊆ {a}⟧ ⟹
        map (Λ.App a) (map Λ.un-App2 U) = U
  by (induct U) auto

lemma map-Lam-Resid:
assumes coinitial T U
shows map Λ.Lam (T *\* U) = map Λ.Lam T *\* map Λ.Lam U
proof −
  interpret Lam: simulation Λ.resid Λ.resid ‹λt. if Λ.arr t then λ[t] else ♯›
    using Λ.Lam-is-simulation by simp
  interpret Lamx: simulation Resid Resid
```

```
                ‹λT. if Arr T then
                      map (λt. if Λ.arr t then λ[t] else ♯) T
                 else []›
    using Lam.lifts-to-paths by simp
  have ⋀T. Arr T ⟹ map (λt. if Λ.arr t then λ[t] else ♯) T = map Λ.Lam T
    using set-Arr-subset-arr by auto
  moreover have Arr (T *\* U)
    using assms confluence_P Con-imp-Arr-Resid con-char by force
  moreover have T *⌢* U
    using assms confluence by simp
  moreover have Arr T ∧ Arr U
    using assms arr-char by auto
  ultimately show ?thesis
    using assms Lamx.preserves-resid [of T U] by presburger
qed

lemma map-App1-Resid:
assumes Λ.Ide x and coinitial T U
shows map (Λ.App x) (T *\* U) = map (Λ.App x) T *\* map (Λ.App x) U
proof −
  interpret App: simulation Λ.resid Λ.resid ‹λt. if Λ.arr t then x ⋆ t else ♯›
    using assms Λ.App-is-simulation2 by simp
  interpret Appx: simulation Resid Resid
                ‹λT. if Arr T then map (λt. if Λ.arr t then x ⋆ t else ♯) T else []›
    using App.lifts-to-paths by simp
  have ⋀T. Arr T ⟹ map (λt. if Λ.arr t then x ⋆ t else ♯) T = map (Λ.App x) T
    using set-Arr-subset-arr by auto
  moreover have Arr (T *\* U)
    using assms confluence_P Con-imp-Arr-Resid con-char by force
  moreover have T *⌢* U
    using assms confluence by simp
  moreover have Arr T ∧ Arr U
    using assms arr-char by auto
  ultimately show ?thesis
    using assms Appx.preserves-resid [of T U] by presburger
qed

lemma map-App2-Resid:
assumes Λ.Ide x and coinitial T U
shows map (λt. t ⋆ x) (T *\* U) = map (λt. t ⋆ x) T *\* map (λt. t ⋆ x) U
proof −
  interpret App: simulation Λ.resid Λ.resid ‹λt. if Λ.arr t then t ⋆ x else ♯›
    using assms Λ.App-is-simulation1 by simp
  interpret Appx: simulation Resid Resid
                ‹λT. if Arr T then map (λt. if Λ.arr t then t ⋆ x else ♯) T else []›
    using App.lifts-to-paths by simp
  have ⋀T. Arr T ⟹ map (λt. if Λ.arr t then t ⋆ x else ♯) T = map (λt. t ⋆ x) T
    using set-Arr-subset-arr by auto
  moreover have Arr (T *\* U)
```

```
    using assms confluence_P Con-imp-Arr-Resid con-char by force
  moreover have T *⌢* U
    using assms confluence by simp
  moreover have Arr T ∧ Arr U
    using assms arr-char by auto
  ultimately show ?thesis
    using assms Appx.preserves-resid [of T U] by presburger
qed

lemma cong-map-Lam:
shows T *∼* U ⟹ map Λ.Lam T *∼* map Λ.Lam U
  apply (induct U arbitrary: T)
   apply (simp add: ide-char)
  by (metis map-Lam-Resid cong-implies-coinitial cong-reflexive ideE
      map-is-Nil-conv Con-imp-Arr-Resid arr-char)

lemma cong-map-App1:
shows ⟦Λ.Ide x; T *∼* U⟧ ⟹ map (Λ.App x) T *∼* map (Λ.App x) U
  apply (induct U arbitrary: x T)
   apply (simp add: ide-char)
  apply (intro conjI)
  by (metis Nil-is-map-conv arr-resid-iff-con con-char con-imp-coinitial
            cong-reflexive ideE map-App1-Resid)+

lemma cong-map-App2:
shows ⟦Λ.Ide x; T *∼* U⟧ ⟹ map (λX. X ∘ x) T *∼* map (λX. X ∘ x) U
  apply (induct U arbitrary: x T)
   apply (simp add: ide-char)
  apply (intro conjI)
  by (metis Nil-is-map-conv arr-resid-iff-con con-char cong-implies-coinitial
            cong-reflexive ide-def arr-char ideE map-App2-Resid)+
```

3.3.3 Decomposition of 'App Paths'

The following series of results is aimed at showing that a reduction path, all of whose transitions have *App* as their top-level constructor, can be factored up to congruence into a reduction path in which only the "rator" components are reduced, followed by a reduction path in which only the "rand" components are reduced.

```
lemma orthogonal-App-single-single:
assumes Λ.Arr t and Λ.Arr u
shows [Λ.Src t ∘ u] *\* [t ∘ Λ.Src u] = [Λ.Trg t ∘ u]
and [t ∘ Λ.Src u] *\* [Λ.Src t ∘ u] = [t ∘ Λ.Trg u]
  using assms arr-char Λ.Arr-not-Nil by auto

lemma orthogonal-App-single-Arr:
shows ⟦Arr [t]; Arr U⟧ ⟹
        map (Λ.App (Λ.Src t)) U *\* [t ∘ Λ.Src (hd U)] = map (Λ.App (Λ.Trg t)) U ∧
        [t ∘ Λ.Src (hd U)] *\* map (Λ.App (Λ.Src t)) U = [t ∘ Λ.Trg (last U)]
proof (induct U arbitrary: t)
```

```
show ⋀t. ⟦Arr [t]; Arr []⟧ ⟹
           map (Λ.App (Λ.Src t)) [] *\* [t ⋆ Λ.Src (hd [])] = map (Λ.App (Λ.Trg t)) [] ∧
           [t ⋆ Λ.Src (hd [])] *\* map (Λ.App (Λ.Src t)) [] = [t ⋆ Λ.Trg (last [])]
  by fastforce
fix t u U
assume ind: ⋀t. ⟦Arr [t]; Arr U⟧ ⟹
                 map (Λ.App (Λ.Src t)) U *\* [t ⋆ Λ.Src (hd U)] =
                 map (Λ.App (Λ.Trg t)) U ∧
                 [t ⋆ Λ.Src (hd U)] *\* map (Λ.App (Λ.Src t)) U = [t ⋆ Λ.Trg (last U)]
assume t: Arr [t]
assume uU: Arr (u # U)
show map (Λ.App (Λ.Src t)) (u # U) *\* [t ⋆ Λ.Src (hd (u # U))] =
     map (Λ.App (Λ.Trg t)) (u # U) ∧
     [t ⋆ Λ.Src (hd (u # U))] *\* map (Λ.App (Λ.Src t)) (u # U) =
     [t ⋆ Λ.Trg (last (u # U))]
proof (cases U = [])
  show U = [] ⟹ ?thesis
    using t uU orthogonal-App-single-single by simp
  assume U: U ≠ []
  have 2: coinitial ([Λ.Src t ⋆ u] @ map (Λ.App (Λ.Src t)) U) [t ⋆ Λ.Src u]
  proof (intro coinitialI')
    show 3: arr ([Λ.Src t ⋆ u] @ map (Λ.App (Λ.Src t)) U)
      using t uU
      by (metis Arr-iff-Con-self Arr-map-App2 Con-rec(1) append-Cons append-Nil
          Λ.Con-implies-Arr2 Λ.Ide-Src Λ.con-char list.simps(9) arr-char)
    show src ([Λ.Src t ⋆ u] @ map (Λ.App (Λ.Src t)) U) ∼ src [t ⋆ Λ.Src u]
    proof −
      have seq [Λ.Src t ⋆ u] (map (Λ.App (Λ.Src t)) U)
        using U 3 arr-append-imp-seq by force
      hence sources ([Λ.Src t ⋆ u] @ map (Λ.App (Λ.Src t)) U) = sources [t ⋆ Λ.Src u]
        using sources-append [of [Λ.Src t ⋆ u] map (Λ.App (Λ.Src t)) U]
              sources-single-Src [of Λ.Src t ⋆ u]
              sources-single-Src [of t ⋆ Λ.Src u]
        using arr-char t
        by (simp add: seq-char)
      thus ?thesis
        using 3 coinitial-iff' by blast
    qed
  qed
  show ?thesis
  proof
    show 4: map (Λ.App (Λ.Src t)) (u # U) *\* [t ⋆ Λ.Src (hd (u # U))] =
            map (Λ.App (Λ.Trg t)) (u # U)
    proof −
      have map (Λ.App (Λ.Src t)) (u # U) *\* [t ⋆ Λ.Src (hd (u # U))] =
           ([Λ.Src t ⋆ u] @ map (Λ.App (Λ.Src t)) U) *\* [t ⋆ Λ.Src u]
        by simp
      also have ... = [Λ.Src t ⋆ u] *\* [t ⋆ Λ.Src u] @
                      map (Λ.App (Λ.Src t)) U *\* ([t ⋆ Λ.Src u] *\* [Λ.Src t ⋆ u])
```

```
      by (meson 2 Resid-append(1) con-char confluence not-Cons-self2)
    also have ... = [Λ.Trg t ∘ u] @ map (Λ.App (Λ.Src t)) U *\* [t ∘ Λ.Trg u]
      using t Λ.Arr-not-Nil
      by (metis Arr-imp-arr-hd Λ.arr-char list.sel(1) orthogonal-App-single-single(1)
          orthogonal-App-single-single(2) uU)
    also have ... = [Λ.Trg t ∘ u] @ map (Λ.App (Λ.Trg t)) U
    proof −
      have Λ.Src (hd U) = Λ.Trg u
        using U uU Arr.elims(2) Srcs-simpΛP by force
      thus ?thesis
        using t uU ind Arr.elims(2) by fastforce
    qed
    also have ... = map (Λ.App (Λ.Trg t)) (u # U)
      by auto
    finally show ?thesis by blast
  qed
  show [t ∘ Λ.Src (hd (u # U))] *\* map (Λ.App (Λ.Src t)) (u # U) =
       [t ∘ Λ.Trg (last (u # U))]
  proof −
    have [t ∘ Λ.Src (hd (u # U))] *\* map (Λ.App (Λ.Src t)) (u # U) =
         ([t ∘ Λ.Src (hd (u # U))] *\* [Λ.Src t ∘ u]) *\* map (Λ.App (Λ.Src t)) U
      by (metis U 4 Con-sym Resid-cons(2) list.distinct(1) list.simps(9) map-is-Nil-conv)
    also have ... = [t ∘ Λ.Trg u] *\* map (Λ.App (Λ.Src t)) U
      by (metis Arr-imp-arr-hd lambda-calculus.arr-char list.sel(1)
          orthogonal-App-single-single(2) t uU)
    also have ... = [t ∘ Λ.Trg (last (u # U))]
     by (metis 2 t U uU Con-Arr-self Con-cons(1) Con-implies-Arr(1) Trg-last-Src-hd-eqI
         arr-append-imp-seq coinitialE ind Λ.Src.simps(4) Λ.Trg.simps(3)
         Λ.lambda.inject(3) last.simps list.distinct(1) list.map-sel(1) map-is-Nil-conv)
    finally show ?thesis by blast
  qed
 qed
qed

lemma orthogonal-App-Arr-Arr:
shows ⟦Arr T; Arr U⟧ ⟹
        map (Λ.App (Λ.Src (hd T))) U *\* map (λX. Λ.App X (Λ.Src (hd U))) T =
        map (Λ.App (Λ.Trg (last T))) U ∧
        map (λX. X ∘ Λ.Src (hd U)) T *\* map (Λ.App (Λ.Src (hd T))) U =
        map (λX. X ∘ Λ.Trg (last U)) T
proof (induct T arbitrary: U)
  show ⋀U. ⟦Arr []; Arr U⟧
           ⟹ map (Λ.App (Λ.Src (hd []))) U *\* map (λX. X ∘ Λ.Src (hd U)) [] =
              map (Λ.App (Λ.Trg (last []))) U ∧
              map (λX. X ∘ Λ.Src (hd U)) [] *\* map (Λ.App (Λ.Src (hd []))) U =
              map (λX. X ∘ Λ.Trg (last U)) []
    by simp
  fix t T U
```

assume *ind*: $\bigwedge U.\ [\![Arr\ T;\ Arr\ U]\!]$
$\Longrightarrow map\ (\Lambda.App\ (\Lambda.Src\ (hd\ T)))\ U\ {}^*\backslash^*$
$map\ (\lambda X.\ \Lambda.App\ X\ (\Lambda.Src\ (hd\ U)))\ T =$
$map\ (\Lambda.App\ (\Lambda.Trg\ (last\ T)))\ U \wedge$
$map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ T\ {}^*\backslash^*\ map\ (\Lambda.App\ (\Lambda.Src\ (hd\ T)))\ U =$
$map\ (\lambda X.\ X \circ \Lambda.Trg\ (last\ U))\ T$
assume *tT*: $Arr\ (t\ \#\ T)$
assume *U*: $Arr\ U$
show $map\ (\Lambda.App\ (\Lambda.Src\ (hd\ (t\ \#\ T))))\ U\ {}^*\backslash^*\ map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ (t\ \#\ T) =$
$map\ (\Lambda.App\ (\Lambda.Trg\ (last\ (t\ \#\ T))))\ U \wedge$
$map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ (t\ \#\ T)\ {}^*\backslash^*\ map\ (\Lambda.App\ (\Lambda.Src\ (hd\ (t\ \#\ T))))\ U =$
$map\ (\lambda X.\ X \circ \Lambda.Trg\ (last\ U))\ (t\ \#\ T)$
proof $(cases\ T = [])$
show $T = [] \Longrightarrow ?thesis$
using *tT U*
by (*simp add*: *orthogonal-App-single-Arr*)
assume *T*: $T \neq []$
have *1*: $Arr\ T$
using *T tT Arr-imp-Arr-tl* **by** *fastforce*
have *2*: $\Lambda.Src\ (hd\ T) = \Lambda.Trg\ t$
using *tT T Arr.elims*(*2*) *Srcs-simp*$_{\Lambda P}$ **by** *force*
show *?thesis*
proof
show *3*: $map\ (\Lambda.App\ (\Lambda.Src\ (hd\ (t\ \#\ T))))\ U\ {}^*\backslash^*$
$map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ (t\ \#\ T) =$
$map\ (\Lambda.App\ (\Lambda.Trg\ (last\ (t\ \#\ T))))\ U$
proof −
have $map\ (\Lambda.App\ (\Lambda.Src\ (hd\ (t\ \#\ T))))\ U\ {}^*\backslash^*\ map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ (t\ \#\ T)$
=
$map\ (\Lambda.App\ (\Lambda.Src\ t))\ U\ {}^*\backslash^*$
$([\Lambda.App\ t\ (\Lambda.Src\ (hd\ U))]\ @\ map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ T)$
using *tT U* **by** *simp*
also have ... = $(map\ (\Lambda.App\ (\Lambda.Src\ t))\ U\ {}^*\backslash^*\ [t \circ \Lambda.Src\ (hd\ U)])\ {}^*\backslash^*$
$map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ T$
using *tT U Resid-append*(*2*)
by (*metis Con-appendI*(*2*) *Resid.simps*(*1*) *T map-is-Nil-conv not-Cons-self2*)
also have ... = $map\ (\Lambda.App\ (\Lambda.Trg\ t))\ U\ {}^*\backslash^*\ map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ T$
using *tT U orthogonal-App-single-Arr Arr-imp-arr-hd* **by** *fastforce*
also have ... = $map\ (\Lambda.App\ (\Lambda.Trg\ (last\ (t\ \#\ T))))\ U$
using *tT U 1 2 ind* **by** *auto*
finally show *?thesis* **by** *blast*
qed
show $map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ (t\ \#\ T)\ {}^*\backslash^*$
$map\ (\Lambda.App\ (\Lambda.Src\ (hd\ (t\ \#\ T))))\ U =$
$map\ (\lambda X.\ X \circ \Lambda.Trg\ (last\ U))\ (t\ \#\ T)$
proof −
have $map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ (t\ \#\ T)\ {}^*\backslash^*$
$map\ (\Lambda.App\ (\Lambda.Src\ (hd\ (t\ \#\ T))))\ U =$
$([t \circ \Lambda.Src\ (hd\ U)]\ @\ map\ (\lambda X.\ X \circ \Lambda.Src\ (hd\ U))\ T)\ {}^*\backslash^*$

```
          map (Λ.App (Λ.Src t)) U
        using tT U by simp
      also have ... = ([t ∘ Λ.Src (hd U)] *\* map (Λ.App (Λ.Src t)) U) @
                      (map (λX. X ∘ Λ.Src (hd U)) T *\*
                         (map (Λ.App (Λ.Src t)) U *\* [t ∘ Λ.Src (hd U)]))
        using tT U 3 Con-sym
              Resid-append(1)
                [of [t ∘ Λ.Src (hd U)] map (λX. X ∘ Λ.Src (hd U)) T
                 map (Λ.App (Λ.Src t)) U]
        by fastforce
      also have ... = [t ∘ Λ.Trg (last U)] @
                        map (λX. X ∘ Λ.Src (hd U)) T *\* map (Λ.App (Λ.Trg t)) U
        using tT U Arr-imp-arr-hd orthogonal-App-single-Arr by fastforce
      also have ... = [t ∘ Λ.Trg (last U)] @ map (λX. X ∘ Λ.Trg (last U)) T
        using tT U 1 2 ind by presburger
      also have ... = map (λX. X ∘ Λ.Trg (last U)) (t # T)
        by simp
      finally show ?thesis by blast
    qed
  qed
 qed
qed

lemma orthogonal-App-cong:
assumes Arr T and Arr U
shows map (λX. X ∘ Λ.Src (hd U)) T @ map (Λ.App (Λ.Trg (last T))) U *∼*
      map (Λ.App (Λ.Src (hd T))) U @ map (λX. X ∘ Λ.Trg (last U)) T

proof
  have 1: Arr (map (λX. X ∘ Λ.Src (hd U)) T)
    using assms Arr-imp-arr-hd Arr-map-App1 Λ.Ide-Src by force
  have 2: Arr (map (Λ.App (Λ.Trg (last T))) U)
    using assms Arr-imp-arr-last Arr-map-App2 Λ.Ide-Trg by force
  have 3: Arr (map (Λ.App (Λ.Src (hd T))) U)
    using assms Arr-imp-arr-hd Arr-map-App2 Λ.Ide-Src by force
  have 4: Arr (map (λX. X ∘ Λ.Trg (last U)) T)
    using assms Arr-imp-arr-last Arr-map-App1 Λ.Ide-Trg by force
  have 5: Arr (map (λX. X ∘ Λ.Src (hd U)) T @ map (Λ.App (Λ.Trg (last T))) U)
    using assms
    by (metis (no-types, lifting) 1 2 Arr.simps(2) Arr-has-Src Arr-imp-arr-last
        Srcs.simps(1) Srcs-Resid-Arr-single Trgs-simpP arr-append arr-char last-map
        orthogonal-App-single-Arr seq-char)
  have 6: Arr (map (Λ.App (Λ.Src (hd T))) U @ map (λX. X ∘ Λ.Trg (last U)) T)
    using assms
    by (metis (no-types, lifting) 3 4 Arr.simps(2) Arr-has-Src Arr-imp-arr-hd
        Srcs.simps(1) Srcs.simps(2) Srcs-Resid Srcs-simpP arr-append arr-char hd-map
        orthogonal-App-single-Arr seq-char)
  have 7: Con (map (λX. X ∘ Λ.Src (hd U)) T @ map ((∘) (Λ.Trg (last T))) U)
              (map ((∘) (Λ.Src (hd T))) U @ map (λX. X ∘ Λ.Trg (last U)) T)
```

```
    using assms orthogonal-App-Arr-Arr [of T U]
    by (metis 1 2 5 6 Con-imp-eq-Srcs Resid.simps(1) Srcs-append confluence-ind)
  have 8: Con (map ((∘) (Λ.Src (hd T))) U @ map (λX. X ∘ Λ.Trg (last U)) T)
           (map (λX. X ∘ Λ.Src (hd U)) T @ map ((∘) (Λ.Trg (last T))) U)
    using 7 Con-sym by simp
  show map (λX. X ∘ Λ.Src (hd U)) T @ map ((∘) (Λ.Trg (last T))) U *≲*
       map ((∘) (Λ.Src (hd T))) U @ map (λX. X ∘ Λ.Trg (last U)) T
  proof −
    have (map (λX. X ∘ Λ.Src (hd U)) T @ map ((∘) (Λ.Trg (last T))) U) *\*
          (map ((∘) (Λ.Src (hd T))) U @ map (λX. X ∘ Λ.Trg (last U)) T) =
         map (λX. X ∘ Λ.Trg (last U)) T *\* map (λX. X ∘ Λ.Trg (last U)) T @
          (map ((∘) (Λ.Trg (last T))) U *\* map ((∘) (Λ.Trg (last T))) U) *\*
            (map (λX. X ∘ Λ.Trg (last U)) T *\* map (λX. X ∘ Λ.Trg (last U)) T)
      using assms 7 orthogonal-App-Arr-Arr
            Resid-append2
              [of map (λX. X ∘ Λ.Src (hd U)) T map (Λ.App (Λ.Trg (last T))) U
                  map (Λ.App (Λ.Src (hd T))) U map (λX. X ∘ Λ.Trg (last U)) T]
      by fastforce
    moreover have Ide ...
      using assms 1 2 3 4 5 6 7 Resid-Arr-self
      by (metis Arr-append-iff_P Con-Arr-self Con-imp-Arr-Resid Ide-appendI_P
          Resid-Ide-Arr-ind append-Nil2 calculation)
    ultimately show ?thesis
      using ide-char by presburger
  qed
  show map ((∘) (Λ.Src (hd T))) U @ map (λX. X ∘ Λ.Trg (last U)) T *≲*
       map (λX. X ∘ Λ.Src (hd U)) T @ map ((∘) (Λ.Trg (last T))) U
  proof −
    have map ((∘) (Λ.Src (hd T))) U *\* map (λX. X ∘ Λ.Src (hd U)) T =
         map ((∘) (Λ.Trg (last T))) U
      by (simp add: assms orthogonal-App-Arr-Arr)
    have (map ((∘) (Λ.Src (hd T))) U @ map (λX. X ∘ Λ.Trg (last U)) T) *\*
          (map (λX. X ∘ Λ.Src (hd U)) T @ map ((∘) (Λ.Trg (last T))) U) =
         (map ((∘) (Λ.Trg (last T))) U) *\* map ((∘) (Λ.Trg (last T))) U @
           (map (λX. X ∘ Λ.Trg (last U)) T *\* map (λX. X ∘ Λ.Trg (last U)) T) *\*
             (map ((∘) (Λ.Trg (last T))) U *\* map ((∘) (Λ.Trg (last T))) U)
      using assms 8 orthogonal-App-Arr-Arr [of T U]
            Resid-append2
              [of map (Λ.App (Λ.Src (hd T))) U map (λX. X ∘ Λ.Trg (last U)) T
                  map (λX. X ∘ Λ.Src (hd U)) T map (Λ.App (Λ.Trg (last T))) U]
      by fastforce
    moreover have Ide ...
      using assms 1 2 3 4 5 6 8 Resid-Arr-self Arr-append-iff_P Con-sym
      by (metis Con-Arr-self Con-imp-Arr-Resid Ide-appendI_P Resid-Ide-Arr-ind
          append-Nil2 calculation)
    ultimately show ?thesis
      using ide-char by presburger
  qed
qed
```

We arrive at the final objective of this section: factorization, up to congruence, of a path whose transitions all have *App* as the top-level constructor, into the composite of a path that reduces only the "rators" and a path that reduces only the "rands".

```
lemma map-App-decomp:
shows ⟦Arr U; set U ⊆ Collect Λ.is-App⟧ ⟹
        map (λX. X ∘ Λ.Src (Λ.un-App2 (hd U))) (map Λ.un-App1 U) @
          map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U) *∼*
        U
proof (induct U)
  show Arr [] ⟹ map (λX. X ∘ Λ.Src (Λ.un-App2 (hd []))) (map Λ.un-App1 []) @
                  map (Λ.App (Λ.Trg (Λ.un-App1 (last [])))) (map Λ.un-App2 []) *∼*
                []
    by simp
  fix u U
  assume ind: ⟦Arr U; set U ⊆ Collect Λ.is-App⟧ ⟹
                map (λX. Λ.App X (Λ.Src (Λ.un-App2 (hd U)))) (map Λ.un-App1 U) @
                  map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U) *∼*
                U
  assume uU: Arr (u # U)
  assume set: set (u # U) ⊆ Collect Λ.is-App
  have u: Λ.Arr u ∧ Λ.is-App u
    using set set-Arr-subset-arr uU by fastforce
  show map (λX. X ∘ Λ.Src (Λ.un-App2 (hd (u # U)))) (map Λ.un-App1 (u # U)) @
         map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) (map Λ.un-App2 (u # U)) *∼*
       u # U
  proof (cases U = [])
    assume U: U = []
    show ?thesis
      using u U Λ.Con-sym Λ.Ide-iff-Src-self Λ.resid-Arr-self Λ.resid-Src-Arr
            Λ.resid-Arr-Src Λ.Src-resid Λ.Arr-resid ide-char Λ.Arr-not-Nil
      by (cases u, simp-all)
    next
    assume U: U ≠ []
    have 1: Arr (map Λ.un-App1 U)
      using U set Arr-map-un-App1 uU
      by (metis Arr-imp-Arr-tl list.distinct(1) list.map-disc-iff list.map-sel(2) list.sel(3))
    have 2: Arr [Λ.un-App2 u]
      using U uU set
      by (metis Arr.simps(2) Arr-imp-arr-hd Arr-map-un-App2 hd-map list.discI list.sel(1))
    have 3: Λ.Arr (Λ.un-App1 u) ∧ Λ.Arr (Λ.un-App2 u)
      using uU set
      by (metis Arr-imp-arr-hd Arr-map-un-App1 Arr-map-un-App2 Λ.arr-char
          list.distinct(1) list.map-sel(1) list.sel(1))
    have 4: map (λX. X ∘ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
              [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u] *∼*
            [Λ.Src (hd (map Λ.un-App1 U)) ∘ Λ.un-App2 u] @
              map (λX. X ∘ Λ.Trg (last [Λ.un-App2 u])) (map Λ.un-App1 U)
    proof −
      have map (λX. X ∘ Λ.Src (hd [Λ.un-App2 u])) (map Λ.un-App1 U) =
```

```
      map (λX. X ∘ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U)
    using U uU set by simp
  moreover have map (Λ.App (Λ.Trg (last (map Λ.un-App1 U)))) [Λ.un-App2 u] =
               [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u]
    by (simp add: U last-map)
  moreover have map (Λ.App (Λ.Src (hd (map Λ.un-App1 U)))) [Λ.un-App2 u] =
               [Λ.Src (hd (map Λ.un-App1 U)) ∘ Λ.un-App2 u]
    by simp
  moreover have map (λX. X ∘ Λ.Trg (last [Λ.un-App2 u])) (map Λ.un-App1 U) =
               map (λX. X ∘ Λ.Trg (last [Λ.un-App2 u])) (map Λ.un-App1 U)
    using U uU set by blast
  ultimately show ?thesis
    using U uU set last-map hd-map 1 2 3
        orthogonal-App-cong [of map Λ.un-App1 U [Λ.un-App2 u]]
    by presburger
qed
have 5: Λ.Arr (Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u))
  by (simp add: 3)
have 6: Arr (map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U))
  by (metis 1 Arr-imp-arr-last Arr-map-App2 Arr-map-un-App2 Con-implies-Arr(2)
      Ide.simps(1) Resid-Arr-self Resid-cons(2) U insert-subset
      Λ.Ide-Trg Λ.arr-char last-map list.simps(15) set uU)
have 7: Λ.Arr (Λ.Trg (Λ.un-App1 (last U)))
  by (metis 4 Arr.simps(2) Arr-append-iff_P Con-implies-Arr(2) Ide.simps(1)
      U ide-char Λ.Arr.simps(4) Λ.arr-char list.map-disc-iff not-Cons-self2)
have 8: Λ.Src (hd (map Λ.un-App1 U)) = Λ.Trg (Λ.un-App1 u)
proof −
  have Λ.Src (hd U) = Λ.Trg u
    using u uU U by fastforce
  thus ?thesis
    using u uU U set
    apply (cases u; cases hd U)
                  apply (simp-all add: list.map-sel(1))
    using list.set-sel(1)
    by fastforce
qed
have 9: Λ.Src (Λ.un-App2 (hd U)) = Λ.Trg (Λ.un-App2 u)
proof −
  have Λ.Src (hd U) = Λ.Trg u
    using u uU U by fastforce
  thus ?thesis
    using u uU U set
    apply (cases u; cases hd U)
                  apply simp-all
    by (metis lambda-calculus.lambda.disc(15) list.set-sel(1) mem-Collect-eq
      subset-code(1))
qed
have map (λX. X ∘ Λ.Src (Λ.un-App2 (hd (u # U)))) (map Λ.un-App1 (u # U)) @
      map ((∘) (Λ.Trg (Λ.un-App1 (last (u # U))))) (map Λ.un-App2 (u # U)) =
```

```
    [Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)] @
      (map (λX. X ∘ Λ.Src (Λ.un-App2 u))
          (map Λ.un-App1 U) @ [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u]) @
        map ((∘) (Λ.Trg (Λ.un-App1 (last U)))) (map Λ.un-App2 U)
  using uU U by simp
also have 12: cong ... ([Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)] @
                    ([Λ.Src (hd (map Λ.un-App1 U)) ∘ Λ.un-App2 u] @
                      map (λX. X ∘ Λ.Trg (last [Λ.un-App2 u])) (map Λ.un-App1 U)) @
                    map ((∘) (Λ.Trg (Λ.un-App1 (last U)))) (map Λ.un-App2 U))
proof (intro cong-append [of [Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)]]
          cong-append [where U = map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X)
                                     (map Λ.un-App2 U)])
  show [Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)] *∼* [Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)]
    using 5 arr-char cong-reflexive Arr.simps(2) Λ.arr-char by presburger
  show map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U) *∼*
       map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U)
    using 6 cong-reflexive by auto
  show map (λX. X ∘ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
         [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u] *∼*
       [Λ.Src (hd (map Λ.un-App1 U)) ∘ Λ.un-App2 u] @
         map (λX. X ∘ Λ.Trg (last [Λ.un-App2 u])) (map Λ.un-App1 U)
    using 4 by simp
  show 10: seq [Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)]
               ((map (λX. X ∘ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
                 [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u]) @
                   map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U))
  proof
    show Arr [Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)]
      using 5 Arr.simps(2) by blast
    show Arr ((map (λX. X ∘ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
                [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u]) @
              map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U))
    proof (intro Arr-appendI_PWE)
      show Arr (map (λX. X ∘ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U))
        using 1 3 Arr-map-App1 lambda-calculus.Ide-Src by blast
      show Arr [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u]
        by (simp add: 3 7)
      show Trg (map (λX. X ∘ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U)) =
           Src [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u]
        by (metis 4 Arr-appendE_PWE Con-implies-Arr(2) Ide.simps(1) U ide-char
            list.map-disc-iff not-Cons-self2)
      show Arr (map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U))
        using 6 by simp
      show Trg (map (λX. X ∘ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
                  [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u]) =
           Src (map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U))
        using U uU set 1 3 6 7 9 Srcs-simp_PWE Arr-imp-arr-hd Arr-imp-arr-last
        apply auto
        by (metis Nil-is-map-conv hd-map Λ.Src.simps(4) Λ.Src-Trg Λ.Trg-Trg
```

```
        last-map list.map-comp)
    qed
    show Λ.Trg (last [Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)]) =
         Λ.Src (hd ((map (λX. X ∘ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
                     [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u]) @
                    map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U)))
      using 8 9
      by (simp add: 3 U hd-map)
  qed
  show seq (map (λX. X ∘ Λ.Src (Λ.un-App2 u)) (map Λ.un-App1 U) @
            [Λ.Trg (Λ.un-App1 (last U)) ∘ Λ.un-App2 u])
           (map (λX. Λ.Trg (Λ.un-App1 (last U)) ∘ X) (map Λ.un-App2 U))
    by (metis Nil-is-map-conv U 10 append-is-Nil-conv arr-append-imp-seq seqE)
qed
also have 11: [Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)] @
              ([Λ.Src (hd (map Λ.un-App1 U)) ∘ Λ.un-App2 u] @
                map (λX. X ∘ Λ.Trg (last [Λ.un-App2 u])) (map Λ.un-App1 U)) @
               map ((∘) (Λ.Trg (Λ.un-App1 (last U)))) (map Λ.un-App2 U) =
             ([Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)] @
              [Λ.Src (hd (map Λ.un-App1 U)) ∘ Λ.un-App2 u]) @
              map (λX. X ∘ Λ.Trg (last [Λ.un-App2 u])) (map Λ.un-App1 U) @
               map ((∘) (Λ.Trg (Λ.un-App1 (last U)))) (map Λ.un-App2 U)
  by simp
also have cong ... ([u] @ U)
proof (intro cong-append)
  show seq ([Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)] @
            [Λ.Src (hd (map Λ.un-App1 U)) ∘ Λ.un-App2 u])
           (map (λX. X ∘ Λ.Trg (last [Λ.un-App2 u])) (map Λ.un-App1 U) @
             map ((∘) (Λ.Trg (Λ.un-App1 (last U)))) (map Λ.un-App2 U))
  by (metis 5 11 12 U Arr.simps(1−2) Con-implies-Arr(2) Ide.simps(1) Nil-is-map-conv
      append-is-Nil-conv arr-append-imp-seq arr-char ide-char Λ.arr-char)
  show [Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)] @
       [Λ.Src (hd (map Λ.un-App1 U)) ∘ Λ.un-App2 u] *∼*
      [u]
  proof −
    have [Λ.un-App1 u ∘ Λ.Src (Λ.un-App2 u)] @
         [Λ.Trg (Λ.un-App1 u) ∘ Λ.un-App2 u] *∼*
        [u]
      using u uU U Λ.Arr-Trg Λ.Arr-not-Nil Λ.resid-Arr-self
      apply (cases u)
        apply auto
      by force+
    thus ?thesis using 8 by simp
  qed
  show map (λX. X ∘ Λ.Trg (last [Λ.un-App2 u])) (map Λ.un-App1 U) @
       map ((∘) (Λ.Trg (Λ.un-App1 (last U)))) (map Λ.un-App2 U) *∼*
      U
    using ind set 9
    apply simp
```

```
      using U uU by blast
    qed
    also have [u] @ U = u # U
      by simp
    finally show ?thesis by blast
  qed
qed
```

3.3.4 Miscellaneous

```
lemma Resid-parallel:
assumes cong t t' and coinitial t u
shows u *\* t = u *\* t'
proof −
  have u *\* t = (u *\* t) *\* (t' *\* t)
    using assms
    by (metis con-target conI_P con-sym resid-arr-ide)
  also have ... = (u *\* t') *\* (t *\* t')
    using cube by auto
  also have ... = u *\* t'
    using assms
    by (metis con-target conI_P con-sym resid-arr-ide)
  finally show ?thesis by blast
qed

lemma set-Ide-subset-single-hd:
shows Ide T ⟹ set T ⊆ {hd T}
  apply (induct T, auto)
  using Λ.coinitial-ide-are-cong
by (metis Arr-imp-arr-hd Ide-consE Ide-imp-Ide-hd Ide-implies-Arr Srcs-simp_PWE Srcs-simp_ΛP
    Λ.trg-ide equals0D Λ.Ide-iff-Src-self Λ.arr-char Λ.ide-char set-empty singletonD
    subset-code(1))
```

A single parallel reduction with *Beta* as the top-level operator factors, up to congruence, either as a path in which the top-level redex is contracted first, or as a path in which the top-level redex is contracted last.

```
lemma Beta-decomp:
assumes Λ.Arr t and Λ.Arr u
shows [λ[Λ.Src t] • Λ.Src u] @ [Λ.subst u t] *∼* [λ[t] • u]
and [λ[t] ∘ u] @ [λ[Λ.Trg t] • Λ.Trg u] *∼* [λ[t] • u]
  using assms Λ.Arr-not-Nil Λ.Subst-not-Nil ide-char Λ.Ide-Subst Λ.Ide-Trg
      Λ.Arr-Subst Λ.resid-Arr-self
  by auto
```

If a reduction path follows an initial reduction whose top-level constructor is *Lam*, then all the terms in the path have *Lam* as their top-level constructor.

```
lemma seq-Lam-Arr-implies:
shows ⟦seq [t] U; Λ.is-Lam t⟧ ⟹ set U ⊆ Collect Λ.is-Lam
proof (induct U arbitrary: t)
```

```
    show ⋀t. ⟦seq [t] []; Λ.is-Lam t⟧ ⟹ set [] ⊆ Collect Λ.is-Lam
      by simp
    fix u U t
    assume ind: ⋀t. ⟦seq [t] U; Λ.is-Lam t⟧ ⟹ set U ⊆ Collect Λ.is-Lam
    assume uU: seq [t] (u # U)
    assume t: Λ.is-Lam t
    show set (u # U) ⊆ Collect Λ.is-Lam
    proof −
      have Λ.is-Lam u
        by (metis Trg-last-Src-hd-eqI Λ.Src.simps(1−2,4−5) Λ.Trg.simps(2) Λ.is-App-def
          Λ.is-Beta-def Λ.is-Lam-def Λ.is-Var-def Λ.lambda.disc(9) Λ.lambda.exhaust-disc
          last-ConsL list.sel(1) t uU)
      moreover have set U ⊆ Collect Λ.is-Lam
      proof (cases U = [])
        show U = [] ⟹ ?thesis
          by simp
        assume U: U ≠ []
        have seq [u] U
          by (metis U append-Cons arr-append-imp-seq not-Cons-self2 self-append-conv2
              seqE uU)
        thus ?thesis
          using ind calculation by simp
      qed
      ultimately show ?thesis by auto
    qed
  qed

  lemma seq-map-un-Lam:
  assumes seq [λ[t]] U
  shows seq [t] (map Λ.un-Lam U)
  proof −
    have Arr (λ[t] # U)
      using assms
      by (simp add: seq-char)
    hence Arr (map Λ.un-Lam (λ[t] # U)) ∧ Arr U
      using seq-Lam-Arr-implies
      by (metis Arr-map-un-Lam ‹seq [λ[t]] U› Λ.lambda.discI(2) mem-Collect-eq
          seq-char set-ConsD subset-code(1))
    hence Arr (Λ.un-Lam λ[t] # map Λ.un-Lam U) ∧ Arr U
      by simp
    thus ?thesis
      using seq-char
      by (metis (no-types, lifting) Arr.simps(1) Con-imp-eq-Srcs Con-implies-Arr(2)
          Con-initial-right Resid-rec(1) Resid-rec(3) Srcs-Resid Λ.lambda.sel(2)
          map-is-Nil-conv confluence-ind)
  qed

end
```

3.4 Developments

A *development* is a reduction path from a term in which at each step exactly one redex is contracted, and the only redexes that are contracted are those that are residuals of redexes present in the original term. That is, no redexes are contracted that were newly created as a result of the previous reductions. The main theorem about developments is the Finite Developments Theorem, which states that all developments are finite. A proof of this theorem was published by Hindley [6], who attributes the result to Schroer [9]. Other proofs were published subsequently. Here we follow the paper by de Vrijer [5], which may in some sense be considered the definitive work because de Vrijer's proof gives an exact bound on the number of steps in a development. Since de Vrijer used a classical, named-variable representation of 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 *elementary-reduction* $\sharp \longleftrightarrow$ *False*
| *elementary-reduction* (*«-»*) $\longleftrightarrow$ *False*
| *elementary-reduction* $\boldsymbol{\lambda}[t] \longleftrightarrow$ *elementary-reduction t*
| *elementary-reduction* $(t \circ u) \longleftrightarrow$
(*elementary-reduction t* $\wedge$ *Ide u*) $\vee$ (*Ide t* $\wedge$ *elementary-reduction u*)
| *elementary-reduction* $(\boldsymbol{\lambda}[t] \bullet u) \longleftrightarrow$ *Ide t* $\wedge$ *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 $t = \boldsymbol{\lambda}[\mathit{«1»}] \bullet (\boldsymbol{\lambda}[\mathit{«0»}] \circ \mathit{«0»})$ and $u = \boldsymbol{\lambda}[\mathit{«1»}] \bullet (\boldsymbol{\lambda}[\mathit{«0»}] \bullet \mathit{«0»})$. 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 *elementary-reduction t* $\Longrightarrow$ *arr t*
using *Ide-implies-Arr arr-char*
by (*induct t*) *auto*

lemma *elementary-reduction-not-ide*:
shows *elementary-reduction t* $\Longrightarrow \neg$ *ide t*
using *ide-char*
by (*induct t*) *auto*

lemma *elementary-reduction-Raise-iff*:
shows $\bigwedge$*d n*. *elementary-reduction* (*Raise d n t*) $\longleftrightarrow$ *elementary-reduction t*
using *Ide-Raise*
by (*induct t*) *auto*

lemma *elementary-reduction-Lam-iff*:
shows *is-Lam t* $\Longrightarrow$ *elementary-reduction t* $\longleftrightarrow$ *elementary-reduction* (*un-Lam t*)
by (*metis elementary-reduction.simps*(*3*) *lambda.collapse*(*2*))

lemma *elementary-reduction-App-iff*:
shows *is-App t* $\Longrightarrow$ *elementary-reduction t* $\longleftrightarrow$
(*elementary-reduction* (*un-App1 t*) $\wedge$ *ide* (*un-App2 t*)) $\vee$
(*ide* (*un-App1 t*) $\wedge$ *elementary-reduction* (*un-App2 t*))
using *ide-char*
by (*metis elementary-reduction.simps*(*4*) *lambda.collapse*(*3*))

lemma *elementary-reduction-Beta-iff*:
shows *is-Beta t* $\Longrightarrow$ *elementary-reduction t* $\longleftrightarrow$ *ide* (*un-Beta1 t*) $\wedge$ *ide* (*un-Beta2 t*)
using *ide-char*
by (*metis elementary-reduction.simps*(*5*) *lambda.collapse*(*4*))

lemma *cong-elementary-reductions-are-equal*:
shows $[\![$*elementary-reduction t*; *elementary-reduction u*; *t* $\sim$ *u*$]\!] \Longrightarrow$ *t* = *u*
proof (*induct t arbitrary*: *u*)
show $\bigwedge$*u*. $[\![$*elementary-reduction* $\sharp$; *elementary-reduction u*; $\sharp \sim u]\!] \Longrightarrow \sharp = u$
by *simp*
show $\bigwedge$*x u*. $[\![$*elementary-reduction «x»*; *elementary-reduction u*; *«x»* $\sim$ *u*$]\!] \Longrightarrow$ *«x»* = *u*
by *simp*
show $\bigwedge$*t u*. $[\![\bigwedge u.\ [\![$*elementary-reduction t*; *elementary-reduction u*; *t* $\sim$ *u*$]\!] \Longrightarrow$ *t* = *u*;
elementary-reduction $\boldsymbol{\lambda}$[*t*]; *elementary-reduction u*; $\boldsymbol{\lambda}$[*t*] $\sim$ *u*$]\!]$
$\Longrightarrow \boldsymbol{\lambda}$[*t*] = *u*
by (*metis elementary-reduction-Lam-iff lambda.collapse*(*2*) *lambda.inject*(*2*) *prfx-Lam-iff*)
show $\bigwedge$*t1 t2*. $[\![\bigwedge u.\ [\![$*elementary-reduction t1*; *elementary-reduction u*; *t1* $\sim$ *u*$]\!] \Longrightarrow$ *t1* = *u*;
$\bigwedge u.\ [\![$*elementary-reduction t2*; *elementary-reduction u*; *t2* $\sim$ *u*$]\!] \Longrightarrow$ *t2* = *u*;
elementary-reduction (*t1* $\circ$ *t2*); *elementary-reduction u*; *t1* $\circ$ *t2* $\sim$ *u*$]\!]$
$\Longrightarrow$ *t1* $\circ$ *t2* = *u*
for *u*
using *prfx-App-iff*
apply (*cases u*)
apply *auto*[*3*]
apply (*metis elementary-reduction-App-iff ide-backward-stable lambda.sel*(*3*−*4*)
weak-extensionality)
by *auto*
show $\bigwedge$*t1 t2*. $[\![\bigwedge u.\ [\![$*elementary-reduction t1*; *elementary-reduction u*; *t1* $\sim$ *u*$]\!] \Longrightarrow$ *t1* = *u*;
$\bigwedge u.\ [\![$*elementary-reduction t2*; *elementary-reduction u*; *t2* $\sim$ *u*$]\!] \Longrightarrow$ *t2* = *u*;
elementary-reduction ($\boldsymbol{\lambda}$[*t1*] $\bullet$ *t2*); *elementary-reduction u*; $\boldsymbol{\lambda}$[*t1*] $\bullet$ *t2* $\sim$ *u*$]\!]$
$\Longrightarrow \boldsymbol{\lambda}$[*t1*] $\bullet$ *t2* = *u*
for *u*
using *prfx-App-iff*
apply (*cases u*, *simp-all*)
by (*metis* (*full-types*) *Coinitial-iff-Con Ide-iff-Src-self Ide.simps*(*1*))
qed

An *elementary reduction path* is a path in which each step is an elementary reduction. It will be convenient to regard the empty list as an elementary reduction path, even

though it is not actually a path according to our previous definition of that notion.

definition (**in** *reduction-paths*) *elementary-reduction-path*
where *elementary-reduction-path T* $\longleftrightarrow$
$(T = [] \vee Arr\ T \wedge set\ T \subseteq Collect\ \Lambda.elementary\text{-}reduction)$

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** ‹⊑› *50*)
where *«i»* $\sqsubseteq$ *«i'»* $\longleftrightarrow i = i'$
| $\boldsymbol{\lambda}[t] \sqsubseteq \boldsymbol{\lambda}[t'] \longleftrightarrow t \sqsubseteq t'$
| $t \circ u \sqsubseteq t' \circ u' \longleftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u'$
| $\boldsymbol{\lambda}[t] \circ u \sqsubseteq \boldsymbol{\lambda}[t'] \bullet u' \longleftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u'$
| $\boldsymbol{\lambda}[t] \bullet u \sqsubseteq \boldsymbol{\lambda}[t'] \bullet u' \longleftrightarrow t \sqsubseteq t' \wedge u \sqsubseteq u'$
| - ⊑ - $\longleftrightarrow$ *False*

lemma *subs-implies-prfx*:
shows $t \sqsubseteq u \Longrightarrow t \lesssim u$
apply (*induct t arbitrary*: *u*)
apply *auto*[*1*]
using *subs.elims*(*2*)
apply *fastforce*
proof −
show $\bigwedge t.\ [\![\bigwedge u.\ t \sqsubseteq u \Longrightarrow t \lesssim u;\ \boldsymbol{\lambda}[t] \sqsubseteq u]\!] \Longrightarrow \boldsymbol{\lambda}[t] \lesssim u$ **for** *u*
by (*cases u*, *auto*) *fastforce*
show $\bigwedge t2.\ [\![\bigwedge u1.\ t1 \sqsubseteq u1 \Longrightarrow t1 \lesssim u1;$
$\bigwedge u2.\ t2 \sqsubseteq u2 \Longrightarrow t2 \lesssim u2;$
$t1 \circ t2 \sqsubseteq u]\!]$
$\Longrightarrow t1 \circ t2 \lesssim u$ **for** *t1 u*
apply (*cases t1*; *cases u*)
apply *simp-all*
apply *fastforce+*
apply (*metis Ide-Subst con-char lambda.sel*(*2*) *subs.simps*(*2*) *prfx-Lam-iff prfx-char*

prfx-implies-con)
by *fastforce+*
show $\bigwedge t1\ t2.\ [\![\bigwedge u1.\ t1 \sqsubseteq u1 \Longrightarrow t1 \lesssim u1;$
$\bigwedge u2.\ t2 \sqsubseteq u2 \Longrightarrow t2 \lesssim u2;$
$\boldsymbol{\lambda}[t1] \bullet t2 \sqsubseteq u]\!]$
$\Longrightarrow \boldsymbol{\lambda}[t1] \bullet t2 \lesssim u$ **for** u
using *Ide-Subst*
apply (*cases u*, *simp-all*)
by (*metis Ide.simps*(*1*))
qed

The following is an example showing that two terms can be related by $\lesssim$ without being related by $\sqsubseteq$.

lemma *subs-example*:
shows $\boldsymbol{\lambda}[\text{«}1\text{»}] \bullet (\boldsymbol{\lambda}[\text{«}0\text{»}] \bullet \text{«}0\text{»}) \lesssim \boldsymbol{\lambda}[\text{«}1\text{»}] \bullet (\boldsymbol{\lambda}[\text{«}0\text{»}] \circ \text{«}0\text{»}) = \mathit{True}$
and $\boldsymbol{\lambda}[\text{«}1\text{»}] \bullet (\boldsymbol{\lambda}[\text{«}0\text{»}] \bullet \text{«}0\text{»}) \sqsubseteq \boldsymbol{\lambda}[\text{«}1\text{»}] \bullet (\boldsymbol{\lambda}[\text{«}0\text{»}] \circ \text{«}0\text{»}) = \mathit{False}$
by *auto*

lemma *subs-Ide*:
shows $[\![\mathit{ide}\ u;\ \mathit{Src}\ t = \mathit{Src}\ u]\!] \Longrightarrow u \sqsubseteq t$
using *Ide-Src Ide-implies-Arr Ide-iff-Src-self*
by (*induct t arbitrary*: *u*, *simp-all*) *force+*

lemma *subs-App*:
shows $u \sqsubseteq t1 \circ t2 \longleftrightarrow \mathit{is\text{-}App}\ u \wedge \mathit{un\text{-}App1}\ u \sqsubseteq t1 \wedge \mathit{un\text{-}App2}\ u \sqsubseteq t2$
by (*metis lambda.collapse*(*3*) *prfx-App-iff subs.simps*(*3*) *subs-implies-prfx*)

end

context *reduction-paths*
begin

We now formally define a *development of* t to be an elementary reduction path U that is 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 *development t* [] $\longleftrightarrow \Lambda.\mathit{Arr}\ t$
| *development t* (u # U) $\longleftrightarrow$
$\Lambda.\mathit{elementary\text{-}reduction}\ u \wedge u \sqsubseteq t \wedge \mathit{development}\ (t \setminus u)\ U$

lemma *development-imp-Arr*:
assumes *development t U*
shows $\Lambda.\mathit{Arr}\ t$
using *assms*
by (*metis* Λ.*Con-implies-Arr2* Λ.*Ide.simps*(*1*) Λ.*ide-char* Λ.*subs-implies-prfx*
development.elims(*2*))

lemma *development-Ide*:
shows $\Lambda.Ide\ t \Longrightarrow development\ t\ U \longleftrightarrow U = []$
using $\Lambda.Ide\text{-}implies\text{-}Arr$
apply (*induct U arbitrary*: *t*)
apply *auto*
by (*meson* $\Lambda.elementary\text{-}reduction\text{-}not\text{-}ide$ $\Lambda.ide\text{-}backward\text{-}stable$ $\Lambda.ide\text{-}char$ $\Lambda.subs\text{-}implies\text{-}prfx$)

lemma *development-implies*:
shows $development\ t\ U \Longrightarrow elementary\text{-}reduction\text{-}path\ U \wedge (U \neq [] \longrightarrow U\ ^*\lesssim^*\ [t])$
apply (*induct U arbitrary*: *t*)
using *elementary-reduction-path-def*
apply *simp*
proof −
fix *t u U*
assume *ind*: $\bigwedge t.\ development\ t\ U \Longrightarrow$
$elementary\text{-}reduction\text{-}path\ U \wedge (U \neq [] \longrightarrow U\ ^*\lesssim^*\ [t])$
show $development\ t\ (u\ \#\ U) \Longrightarrow$
$elementary\text{-}reduction\text{-}path\ (u\ \#\ U) \wedge (u\ \#\ U \neq [] \longrightarrow u\ \#\ U\ ^*\lesssim^*\ [t])$
proof (*cases* $U = []$)
assume *uU*: $development\ t\ (u\ \#\ U)$
show $U = [] \Longrightarrow$ *?thesis*
using *uU* $\Lambda.subs\text{-}implies\text{-}prfx$ *ide-char* $\Lambda.elementary\text{-}reduction\text{-}is\text{-}arr$
elementary-reduction-path-def prfx-implies-con
by *force*
assume *U*: $U \neq []$
have $\Lambda.elementary\text{-}reduction\ u \wedge u \sqsubseteq t \wedge development\ (t \setminus u)\ U$
using *U uU development.elims*(*1*) **by** *blast*
hence *1*: $\Lambda.elementary\text{-}reduction\ u \wedge elementary\text{-}reduction\text{-}path\ U \wedge u \sqsubseteq t \wedge$
$(U \neq [] \longrightarrow U\ ^*\lesssim^*\ [t \setminus u])$
using *U uU ind* **by** *auto*
show *?thesis*
proof (*unfold elementary-reduction-path-def*, *intro conjI*)
show $u\ \#\ U = [] \vee Arr\ (u\ \#\ U) \wedge set\ (u\ \#\ U) \subseteq Collect\ \Lambda.elementary\text{-}reduction$
using *U 1*
by (*metis Con-implies-Arr*(*1*) *Con-rec*(*2*) *con-char prfx-implies-con*
elementary-reduction-path-def insert-subset list.simps(*15*) *mem-Collect-eq*
$\Lambda.prfx\text{-}implies\text{-}con$ $\Lambda.subs\text{-}implies\text{-}prfx$)
show $u\ \#\ U \neq [] \longrightarrow u\ \#\ U\ ^*\lesssim^*\ [t]$
proof −
have $u\ \#\ U\ ^*\lesssim^*\ [t] \longleftrightarrow ide\ ([u \setminus t]\ @\ U\ ^*\backslash^*\ [t \setminus u])$
using *1 U Con-rec*(*2*) *Resid-rec*(*2*) *con-char prfx-implies-con*
$\Lambda.prfx\text{-}implies\text{-}con$ $\Lambda.subs\text{-}implies\text{-}prfx$
by *simp*
also have ... $\longleftrightarrow$ *True*
using *U 1 ide-char* $Ide\text{-}append\text{-}iff_{PWE}$ [*of* $[u \setminus t]$ $U\ ^*\backslash^*\ [t \setminus u]$]
by (*metis Ide.simps*(*2*) $Ide\text{-}appendI_{PWE}$ *Src-resid Trg.simps*(*2*)
$\Lambda.apex\text{-}sym$ *con-char* $\Lambda.subs\text{-}implies\text{-}prfx$ *prfx-implies-con*)
finally show *?thesis* **by** *blast*

qed
qed
qed
qed

The converse of the previous result does not hold, because there could be a stage i at which $u_i \lesssim t_i$, but t_i deletes the redex contracted in u_i, so there is nothing forcing that redex to have been originally marked in t. So U being a development of t is a stronger property than U just being an elementary reduction path such that $U\ {}^*\!\lesssim^*\ [t]$.

lemma *development-append*:
shows $[\![$*development t U*; *development* $(t\ {}^1\backslash^*\ U)\ V]\!] \Longrightarrow$ *development t* $(U\ @\ V)$
using *development-imp-Arr null-char*
apply (*induct U arbitrary*: *t V*)
apply *auto*
by (*metis Resid1x.simps*(*2*−*3*) *append-Nil neq-Nil-conv*)

lemma *development-map-Lam*:
shows *development t T* $\Longrightarrow$ *development* $\boldsymbol{\lambda}[t]$ (*map* Λ.*Lam T*)
using Λ.*Arr-not-Nil development-imp-Arr*
by (*induct T arbitrary*: *t*) *auto*

lemma *development-map-App-1*:
shows $[\![$*development t T*; Λ.*Arr u*$]\!] \Longrightarrow$ *development* $(t \circ u)$ (*map* $(\lambda x.\ x \circ \Lambda.Src\ u)\ T$)
apply (*induct T arbitrary*: *t*)
apply (*simp add*: Λ.*Ide-implies-Arr*)
proof −
fix *t T t*′
assume *ind*: $\bigwedge t.\ [\![$*development t T*; Λ.*Arr u*$]\!]$
$\Longrightarrow$ *development* $(t \circ u)$ (*map* $(\lambda x.\ x \circ \Lambda.Src\ u)\ T$)
assume *t*′*T*: *development t* $(t' \#\ T)$
assume *u*: Λ.*Arr u*
show *development* $(t \circ u)$ (*map* $(\lambda x.\ x \circ \Lambda.Src\ u)$ $(t' \#\ T)$)
using *u t*′*T ind*
apply *simp*
using Λ.*Arr-not-Nil* Λ.*Ide-Src development-imp-Arr* Λ.*subs-Ide* **by** *force*
qed

lemma *development-map-App-2*:
shows $[\![\Lambda.Arr\ t$; *development u U*$]\!] \Longrightarrow$ *development* $(t \circ u)$ (*map* $(\lambda x.\ \Lambda.App\ (\Lambda.Src\ t)\ x)$ *U*)
apply (*induct U arbitrary*: *u*)
apply (*simp add*: Λ.*Ide-implies-Arr*)
proof −
fix *u U u*′
assume *ind*: $\bigwedge u.\ [\![\Lambda.Arr\ t$; *development u U*$]\!]$
$\Longrightarrow$ *development* $(t \circ u)$ (*map* $(\Lambda.App\ (\Lambda.Src\ t))\ U$)
assume *u*′*U*: *development u* $(u' \#\ U)$
assume *t*: Λ.*Arr t*
show *development* $(t \circ u)$ (*map* $(\Lambda.App\ (\Lambda.Src\ t))$ $(u' \#\ U)$)

using $t\ u'U\ ind$
apply *simp*
by (*metis* Λ.*Coinitial-iff-Con* Λ.*Ide-Src* Λ.*Ide-iff-Src-self* Λ.*Ide-implies-Arr* *development-imp-Arr* Λ.*ide-char* Λ.*resid-Arr-Ide* Λ.*subs-Ide*)
qed

3.4.1 Finiteness of Developments

A term t has the finite developments property if there exists a finite value that bounds the length of all developments of t. The goal of this section is to prove the Finite Developments Theorem: every term has the finite developments property.

definition *FD*
where $FD\ t \equiv \exists n.\ \forall U.\ development\ t\ U \longrightarrow length\ U \leq n$

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* ⇒ *lambda* ⇒ *nat*
where *mtp x* ♯ = *0*
| *mtp x* «*z*» = (*if z* = *x then 1 else 0*)
| *mtp x* **λ**[*t*] = *mtp* (*Suc x*) *t*
| *mtp x* (*t* ∘ *u*) = *mtp x t* + *mtp x u*
| *mtp x* (**λ**[*t*] • *u*) = *mtp* (*Suc x*) *t* + *mtp x u* ∗ *max 1* (*mtp 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* ⟷ *x* ∈ *FV t*
proof (*induct t arbitrary*: *x*)
show ⋀*x*. *0* < *mtp x* ♯ ⟷ *x* ∈ *FV* ♯

```
    by simp
  show ⋀x z. 0 < mtp x «z» ⟷ x ∈ FV «z»
    by auto
  show Lam: ⋀t x. (⋀x. 0 < mtp x t ⟷ x ∈ FV t)
                  ⟹ 0 < mtp x λ[t] ⟷ x ∈ FV λ[t]
  proof −
    fix t and x :: nat
    assume ind: ⋀x. 0 < mtp x t ⟷ x ∈ FV t
    show 0 < mtp x λ[t] ⟷ x ∈ FV λ[t]
      using ind
      apply auto
      apply (metis Diff-iff One-nat-def diff-Suc-1 empty-iff imageI insert-iff
                   nat.distinct(1))
      by (metis Suc-pred neq0-conv)
  qed
  show ⋀t u x.
        ⟦⋀x. 0 < mtp x t ⟷ x ∈ FV t;
         ⋀x. 0 < mtp x u ⟷ x ∈ FV u⟧
          ⟹ 0 < mtp x (t ∘ u) ⟷ x ∈ FV (t ∘ u)
    by simp
  show ⋀t u x.
        ⟦⋀x. 0 < mtp x t ⟷ x ∈ FV t;
         ⋀x. 0 < mtp x u ⟷ x ∈ FV u⟧
          ⟹ 0 < mtp x (λ[t] • u) ⟷ x ∈ FV (λ[t] • u)
  proof −
    fix t u and x :: nat
    assume ind1: ⋀x. 0 < mtp x t ⟷ x ∈ FV t
    assume ind2: ⋀x. 0 < mtp x u ⟷ x ∈ FV u
    show 0 < mtp x (λ[t] • u) ⟷ x ∈ FV (λ[t] • u)
      using ind1 ind2
      apply simp
      by force
  qed
qed
```

We now establish a fact about commutation of multiplicity and Raise that will be needed subsequently.

```
lemma mtpE-eq-Raise:
shows x < d ⟹ mtp x (Raise d k t) = mtp x t
  by (induct t arbitrary: x k d) auto

lemma mtp-Raise-ind:
shows ⟦l ≤ d; size t ≤ s⟧ ⟹ mtp (x + d + k) (Raise l k t) = mtp (x + d) t
proof (induct s arbitrary: d x k l t)
  show ⋀d x k l. ⟦l ≤ d; size t ≤ 0⟧ ⟹ mtp (x + d + k) (Raise l k t) = mtp (x + d) t
       for t
    by (cases t) auto
  show ⋀s d x k l.
        ⟦⋀d x k l t. ⟦l ≤ d; size t ≤ s⟧ ⟹ mtp (x + d + k) (Raise l k t) = mtp (x + d) t;
```

```
        l ≤ d; size t ≤ Suc s⟧
        ⟹ mtp (x + d + k) (Raise l k t) = mtp (x + d) t
  for t
proof (cases t)
  show ⋀d x k l s. t = ♯ ⟹ mtp (x + d + k) (Raise l k t) = mtp (x + d) t
    by simp
  show ⋀z d x k l s. ⟦l ≤ d; t = «z»⟧
                     ⟹ mtp (x + d + k) (Raise l k t) = mtp (x + d) t
    by simp
  show ⋀u d x k l s. ⟦l ≤ d; size t ≤ Suc s; t = λ[u];
                      (⋀d x k l u. ⟦l ≤ d; size u ≤ s⟧
                                   ⟹ mtp (x + d + k) (Raise l k u) = mtp (x + d) u)⟧
                     ⟹ mtp (x + d + k) (Raise l k t) = mtp (x + d) t
  proof −
    fix u d x s and k l :: nat
    assume l: l ≤ d and s: size t ≤ Suc s and t: t = λ[u]
    assume ind: ⋀d x k l u. ⟦l ≤ d; size u ≤ s⟧
                            ⟹ mtp (x + d + k) (Raise l k u) = mtp (x + d) u
    show mtp (x + d + k) (Raise l k t) = mtp (x + d) t
    proof −
      have mtp (x + d + k) (Raise l k t) = mtp (Suc (x + d + k)) (Raise (Suc l) k u)
        using t by simp
      also have ... = mtp (x + Suc d) u
      proof −
        have size u ≤ s
          using t s by force
        thus ?thesis
          using l s ind [of Suc l Suc d] by simp
      qed
      also have ... = mtp (x + d) t
        using t by auto
      finally show ?thesis by blast
    qed
  qed
  show ⋀t1 t2 d x k l s.
       ⟦⋀d x k l t1. ⟦l ≤ d; size t1 ≤ s⟧
                     ⟹ mtp (x + d + k) (Raise l k t1) = mtp (x + d) t1;
        ⋀d x k l t2. ⟦l ≤ d; size t2 ≤ s⟧
                     ⟹ mtp (x + d + k) (Raise l k t2) = mtp (x + d) t2;
        l ≤ d; size t ≤ Suc s; t = t1 ∘ t2⟧
          ⟹ mtp (x + d + k) (Raise l k t) = mtp (x + d) t
  proof −
    fix t1 t2 s
    assume s: size t ≤ Suc s and t: t = t1 ∘ t2
    have size t1 ≤ s ∧ size t2 ≤ s
      using s t by auto
    thus ⋀d x k l.
         ⟦⋀d x k l t1. ⟦l ≤ d; size t1 ≤ s⟧
                       ⟹ mtp (x + d + k) (Raise l k t1) = mtp (x + d) t1;
```

```
            ⋀d x k l t2. ⟦l ≤ d; size t2 ≤ s⟧
                         ⟹ mtp (x + d + k) (Raise l k t2) = mtp (x + d) t2;
            l ≤ d; size t ≤ Suc s; t = t1 ∘ t2⟧
              ⟹ mtp (x + d + k) (Raise l k t) = mtp (x + d) t
        by simp
    qed
    show ⋀t1 t2 d x k l s.
          ⟦⋀d x k l t1. ⟦l ≤ d; size t1 ≤ s⟧
                         ⟹ mtp (x + d + k) (Raise l k t1) = mtp (x + d) t1;
           ⋀d x k l t2. ⟦l ≤ d; size t2 ≤ s⟧
                         ⟹ mtp (x + d + k) (Raise l k t2) = mtp (x + d) t2;
           l ≤ d; size t ≤ Suc s; t = λ[t1] • t2⟧
             ⟹ mtp (x + d + k) (Raise l k t) = mtp (x + d) t
    proof −
      fix t1 t2 d x s and k l :: nat
      assume l: l ≤ d and s: size t ≤ Suc s and t: t = λ[t1] • t2
      assume ind: ⋀d x k l N. ⟦l ≤ d; size N ≤ s⟧
                              ⟹ mtp (x + d + k) (Raise l k N) = mtp (x + d) N
      show mtp (x + d + k) (Raise l k t) = mtp (x + d) t
      proof −
        have 1: size t1 ≤ s ∧ size t2 ≤ s
          using s t by auto
        have mtp (x + d + k) (Raise l k t) =
             mtp (Suc (x + d + k)) (Raise (Suc l) k t1) +
               mtp (x + d + k) (Raise l k t2) ∗ max 1 (mtp 0 (Raise (Suc l) k t1))
          using t l by simp
        also have ... = mtp (Suc (x + d + k)) (Raise (Suc l) k t1) +
                        mtp (x + d) t2 ∗ max 1 (mtp 0 (Raise (Suc l) k t1))
          using l 1 ind by auto
        also have ... = mtp (x + Suc d) t1 + mtp (x + d) t2 ∗ max 1 (mtp 0 t1)
        proof −
          have mtp (x + Suc d + k) (Raise (Suc l) k t1) = mtp (x + Suc d) t1
            using l 1 ind [of Suc l Suc d t1] by simp
          moreover have mtp 0 (Raise (Suc l) k t1) = mtp 0 t1

            using l 1 ind [of Suc l Suc d t1 k] mtpE-eq-Raise by simp
          ultimately show ?thesis
            by simp
        qed
        also have ... = mtp (x + d) t
          using t by auto
        finally show ?thesis by blast
      qed
    qed
  qed
qed

lemma mtp-Raise:
assumes l ≤ d
```

shows *mtp* (*x* + *d* + *k*) (*Raise l k t*) = *mtp* (*x* + *d*) *t*
using *assms mtp-Raise-ind* **by** *blast*

lemma *mtp-Raise′*:
shows *mtp l* (*Raise l* (*Suc k*) *t*) = *0*
by (*induct t arbitrary*: *k l*) *auto*

lemma *mtp-raise*:
shows *mtp* (*x* + *Suc d*) (*raise d t*) = *mtp* (*Suc x*) *t*
by (*metis Suc-eq-plus1 add.assoc le-add2 le-add-same-cancel2 mtp-Raise plus-1-eq-Suc*)

lemma *mtp-Subst-cancel*:
shows *mtp k* (*Subst* (*Suc d* + *k*) *u t*) = *mtp k t*
proof (*induct t arbitrary*: *k d*)
show ⋀*k d*. *mtp k* (*Subst* (*Suc d* + *k*) *u* ♯) = *mtp k* ♯
by *simp*
show ⋀*k z d*. *mtp k* (*Subst* (*Suc d* + *k*) *u* «*z*») = *mtp k* «*z*»
using *mtp-Raise′*
apply *auto*
by (*metis add-Suc-right add-Suc-shift order-refl raise-plus*)
show ⋀*t k d*. (⋀*k d*. *mtp k* (*Subst* (*Suc d* + *k*) *u t*) = *mtp k t*)
⟹ *mtp k* (*Subst* (*Suc d* + *k*) *u* **λ**[*t*]) = *mtp k* **λ**[*t*]
by (*metis Subst.simps*(*3*) *add-Suc-right mtp.simps*(*3*))
show ⋀*t1 t2 k d*.
⟦⋀*k d*. *mtp k* (*Subst* (*Suc d* + *k*) *u t1*) = *mtp k t1*;
⋀*k d*. *mtp k* (*Subst* (*Suc d* + *k*) *u t2*) = *mtp k t2*⟧
⟹ *mtp k* (*Subst* (*Suc d* + *k*) *u* (*t1* ∘ *t2*)) = *mtp k* (*t1* ∘ *t2*)
by *auto*
show ⋀*t1 t2 k d*.
⟦⋀*k d*. *mtp k* (*Subst* (*Suc d* + *k*) *u t1*) = *mtp k t1*;
⋀*k d*. *mtp k* (*Subst* (*Suc d* + *k*) *u t2*) = *mtp k t2*⟧
⟹ *mtp k* (*Subst* (*Suc d* + *k*) *u* (**λ**[*t1*] • *t2*)) = *mtp k* (**λ**[*t1*] • *t2*)
using *mtp-Raise′*
apply *auto*
by (*metis Nat.add-0-right add-Suc-right*)
qed

lemma *mtp*$_0$*-Subst-cancel*:
shows *mtp 0* (*Subst* (*Suc d*) *u t*) = *mtp 0 t*
using *mtp-Subst-cancel* [*of 0*] **by** *simp*

We can now (!) prove the desired generalization of de Vrijer's formula for the commutation of multiplicity and substitution. This is the main lemma whose form is difficult to find. To get this right, the proper relationships have to exist between the various depth parameters to *Subst* and the arguments to *mtp*.

lemma *mtp-Subst′*:
shows *mtp* (*x* + *Suc d*) (*Subst d u t*) = *mtp* (*x* + *Suc* (*Suc d*)) *t* + *mtp* (*Suc x*) *u* ∗ *mtp d t*
proof (*induct t arbitrary*: *d x u*)
show ⋀*d x u*. *mtp* (*x* + *Suc d*) (*Subst d u* ♯) =

```
    mtp (x + Suc (Suc d)) ♯ + mtp (Suc x) u * mtp d ♯
  by simp
show ⋀z d x u. mtp (x + Suc d) (Subst d u «z») =
                mtp (x + Suc (Suc d)) «z» + mtp (Suc x) u * mtp d «z»
  using mtp-raise by auto
show ⋀t d x u.
       (⋀d x u. mtp (x + Suc d) (Subst d u t) =
                  mtp (x + Suc (Suc d)) t + mtp (Suc x) u * mtp d t)
                 ⟹ mtp (x + Suc d) (Subst d u λ[t]) =
                    mtp (x + Suc (Suc d)) λ[t] + mtp (Suc x) u * mtp d λ[t]
proof −
  fix t u d x
  assume ind: ⋀d x N. mtp (x + Suc d) (Subst d N t) =
                      mtp (x + Suc (Suc d)) t + mtp (Suc x) N * mtp d t
  have mtp (x + Suc d) (Subst d u λ[t]) =
       mtp (Suc x + Suc (Suc d)) t +
       mtp (x + Suc (Suc d)) (raise (Suc d) u) * mtp (Suc d) t
    using ind mtp-raise add-Suc-shift
    by (metis Subst.simps(3) add-Suc-right mtp.simps(3))
  also have ... = mtp (x + Suc (Suc d)) λ[t] + mtp (Suc x) u * mtp d λ[t]
    using Raise-Suc
    by (metis add-Suc-right add-Suc-shift mtp.simps(3) mtp-raise)
  finally show mtp (x + Suc d) (Subst d u λ[t]) =
               mtp (x + Suc (Suc d)) λ[t] + mtp (Suc x) u * mtp d λ[t]
    by blast
qed
show ⋀t1 t2 u d x.
       ⟦⋀d x u. mtp (x + Suc d) (Subst d u t1) =
                  mtp (x + Suc (Suc d)) t1 + mtp (Suc x) u * mtp d t1;
        ⋀d x u. mtp (x + Suc d) (Subst d u t2) =
                  mtp (x + Suc (Suc d)) t2 + mtp (Suc x) u * mtp d t2⟧
         ⟹ mtp (x + Suc d) (Subst d u (t1 ∘ t2)) =
            mtp (x + Suc (Suc d)) (t1 ∘ t2) + mtp (Suc x) u * mtp d (t1 ∘ t2)
  by (simp add: add-mult-distrib2)
show ⋀t1 t2 u d x.
       ⟦⋀d x N. mtp (x + Suc d) (Subst d N t1) =
                  mtp (x + Suc (Suc d)) t1 + mtp (Suc x) N * mtp d t1;
        ⋀d x N. mtp (x + Suc d) (Subst d N t2) =
                  mtp (x + Suc (Suc d)) t2 + mtp (Suc x) N * mtp d t2⟧
         ⟹ mtp (x + Suc d) (Subst d u (λ[t1] • t2)) =
            mtp (x + Suc (Suc d)) (λ[t1] • t2) + mtp (Suc x) u * mtp d (λ[t1] • t2)
proof −
  fix t1 t2 u d x
  assume ind1: ⋀d x N. mtp (x + Suc d) (Subst d N t1) =
                       mtp (x + Suc (Suc d)) t1 + mtp (Suc x) N * mtp d t1
  assume ind2: ⋀d x N. mtp (x + Suc d) (Subst d N t2) =
                       mtp (x + Suc (Suc d)) t2 + mtp (Suc x) N * mtp d t2
  show mtp (x + Suc d) (Subst d u (λ[t1] • t2)) =
       mtp (x + Suc (Suc d)) (λ[t1] • t2) + mtp (Suc x) u * mtp d (λ[t1] • t2)
```

```
    proof −
      let ?A = mtp (Suc x + Suc (Suc d)) t1
      let ?B = mtp (Suc x + Suc d) t2
      let ?M1 = mtp (Suc d) t1
      let ?M2 = mtp d t2
      let ?M1₀ = mtp 0 (Subst (Suc d) u t1)
      let ?M1₀' = mtp 0 t1
      let ?N = mtp (Suc x) u
      have mtp (x + Suc d) (Subst d u (λ[t1] • t2)) =
            mtp (x + Suc d) (λ[Subst (Suc d) u t1] • Subst d u t2)
        by simp
      also have ... = mtp (x + Suc (Suc d)) (Subst (Suc d) u t1) +
                      mtp (x + Suc d) (Subst d u t2) *
                        max 1 (mtp 0 (Subst (Suc d) u t1))
        by simp
      also have ... = (?A + ?N * ?M1) + (?B + ?N * ?M2) * max 1 ?M1₀
        using ind1 ind2 add-Suc-shift by presburger
      also have ... = ?A + ?N * ?M1 + ?B * max 1 ?M1₀ + ?N * ?M2 * max 1 ?M1₀
        by algebra
      also have ... = ?A + ?B * max 1 ?M1₀' + ?N * ?M1 + ?N * ?M2 * max 1 ?M1₀'
      proof −
        have ?M1₀ = ?M1₀'

          using mtp₀-Subst-cancel by blast
        thus ?thesis by auto
      qed
      also have ... = ?A + ?B * max 1 ?M1₀' + ?N * (?M1 + ?M2 * max 1 ?M1₀')
        by algebra
     also have ... =  mtp (Suc x + Suc d) (λ[t1] • t2) + mtp (Suc x) u * mtp d (λ[t1] • t2)
        by simp
      finally show ?thesis by simp
    qed
  qed
qed
```

The following lemma provides expansions that apply when the parameter to *mtp* is *0*, as opposed to the previous lemma, which only applies for parameters greater than *0*.

```
lemma mtp-Subst:
shows mtp k (Subst k u t) = mtp (Suc k) t + mtp k (raise k u) * mtp k t
proof (induct t arbitrary: u k)
  show ⋀u k. mtp k (Subst k u ♯) = mtp (Suc k) ♯ + mtp k (raise k u) * mtp k ♯
    by simp
  show ⋀x u k. mtp k (Subst k u «x») =
                 mtp (Suc k) «x» + mtp k (raise k u) * mtp k «x»
    by auto
  show ⋀t u k. (⋀u k. mtp k (Subst k u t) = mtp (Suc k) t + mtp k (raise k u) * mtp k t)
                       ⟹ mtp k (Subst k u λ[t]) =
                          mtp (Suc k) λ[t] + mtp k (Raise 0 k u) * mtp k λ[t]
    using mtp-Raise [of 0]
```

apply *auto*
by (*metis add.left-neutral*)
show $\bigwedge$*t1 t2 u k.*
$[\![\bigwedge u\ k.\ mtp\ k\ (Subst\ k\ u\ t1) = mtp\ (Suc\ k)\ t1 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t1;$
$\bigwedge u\ k.\ mtp\ k\ (Subst\ k\ u\ t2) = mtp\ (Suc\ k)\ t2 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t2]\!]$
$\Longrightarrow mtp\ k\ (Subst\ k\ u\ (t1 \circ t2)) =$
$mtp\ (Suc\ k)\ (t1 \circ t2) + mtp\ k\ (raise\ k\ u) * mtp\ k\ (t1 \circ t2)$
by (*auto simp add*: *distrib-left*)
show $\bigwedge$*t1 t2 u k.*
$[\![\bigwedge u\ k.\ mtp\ k\ (Subst\ k\ u\ t1) = mtp\ (Suc\ k)\ t1 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t1;$
$\bigwedge u\ k.\ mtp\ k\ (Subst\ k\ u\ t2) = mtp\ (Suc\ k)\ t2 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t2]\!]$
$\Longrightarrow mtp\ k\ (Subst\ k\ u\ (\boldsymbol{\lambda}[t1] \bullet t2)) =$
$mtp\ (Suc\ k)\ (\boldsymbol{\lambda}[t1] \bullet t2) + mtp\ k\ (raise\ k\ u) * mtp\ k\ (\boldsymbol{\lambda}[t1] \bullet t2)$
proof −
fix *t1 t2 u k*
assume *ind1*: $\bigwedge u\ k.\ mtp\ k\ (Subst\ k\ u\ t1) =$
$mtp\ (Suc\ k)\ t1 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t1$
assume *ind2*: $\bigwedge u\ k.\ mtp\ k\ (Subst\ k\ u\ t2) =$
$mtp\ (Suc\ k)\ t2 + mtp\ k\ (raise\ k\ u) * mtp\ k\ t2$
show $mtp\ k\ (Subst\ k\ u\ (\boldsymbol{\lambda}[t1] \bullet t2)) =$
$mtp\ (Suc\ k)\ (\boldsymbol{\lambda}[t1] \bullet t2) + mtp\ k\ (raise\ k\ u) * mtp\ k\ (\boldsymbol{\lambda}[t1] \bullet t2)$
proof −
have $mtp\ (Suc\ k)\ (Raise\ 0\ (Suc\ k)\ u) * mtp\ (Suc\ k)\ t1 +$
$(mtp\ (Suc\ k)\ t2 + mtp\ k\ (Raise\ 0\ k\ u) * mtp\ k\ t2) * max\ (Suc\ 0)\ (mtp\ 0\ t1) =$
$mtp\ (Suc\ k)\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1) +$
$mtp\ k\ (Raise\ 0\ k\ u) * (mtp\ (Suc\ k)\ t1 + mtp\ k\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1))$
proof −
have $mtp\ (Suc\ k)\ (Raise\ 0\ (Suc\ k)\ u) * mtp\ (Suc\ k)\ t1 +$
$(mtp\ (Suc\ k)\ t2 + mtp\ k\ (Raise\ 0\ k\ u) * mtp\ k\ t2) * max\ (Suc\ 0)\ (mtp\ 0\ t1) =$
$mtp\ (Suc\ k)\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1) +$
$mtp\ (Suc\ k)\ (Raise\ 0\ (Suc\ k)\ u) * mtp\ (Suc\ k)\ t1 +$
$mtp\ k\ (Raise\ 0\ k\ u) * mtp\ k\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1)$
by *algebra*
also have ... $= mtp\ (Suc\ k)\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1) +$
$mtp\ (Suc\ k)\ (Raise\ 0\ (Suc\ k)\ u) * mtp\ (Suc\ k)\ t1 +$
$mtp\ 0\ u * mtp\ k\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1)$
using *mtp-Raise* [*of 0 0 0 k u*] **by** *auto*
also have ... $= mtp\ (Suc\ k)\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1) +$
$mtp\ k\ (Raise\ 0\ k\ u) *$
$(mtp\ (Suc\ k)\ t1 + mtp\ k\ t2 * max\ (Suc\ 0)\ (mtp\ 0\ t1))$
by (*metis* (*no-types*, *lifting*) *ab-semigroup-add-class.add-ac*(*1*)
ab-semigroup-mult-class.mult-ac(*1*) *add-mult-distrib2 le-add1 mtp-Raise*
plus-nat.add-0)
finally show *?thesis* **by** *blast*
qed
thus *?thesis*
using *ind1 ind2* mtp_0*-Subst-cancel* **by** *auto*
qed
qed

qed

lemma *mtp0-subst-le*:
shows *mtp 0 (subst u t) ≤ mtp 1 t + mtp 0 u * max 1 (mtp 0 t)*
proof (*cases t*)
  show *t = ♯ ⟹ mtp 0 (subst u t) ≤ mtp 1 t + mtp 0 u * max 1 (mtp 0 t)*
    by *auto*
  show *⋀z. t = «z» ⟹ mtp 0 (subst u t) ≤ mtp 1 t + mtp 0 u * max 1 (mtp 0 t)*
    using *Raise-0* **by** *force*
  show *⋀P. t = λ[P] ⟹ mtp 0 (subst u t) ≤ mtp 1 t + mtp 0 u * max 1 (mtp 0 t)*
    using *mtp-Subst* [*of 0 u t*] *Raise-0* **by** *force*
  show *⋀t1 t2. t = t1 ∘ t2 ⟹ mtp 0 (subst u t) ≤ mtp 1 t + mtp 0 u * max 1 (mtp 0 t)*
    using *mtp-Subst Raise-0 add-mult-distrib2 nat-mult-max-right* **by** *auto*
  show *⋀t1 t2. t = λ[t1] • t2 ⟹ mtp 0 (subst u t) ≤ mtp 1 t + mtp 0 u * max 1 (mtp 0 t)*
    using *mtp-Subst Raise-0*
    by (*metis Nat.add-0-right dual-order.eq-iff max-def mult.commute mult-zero-left*
      not-less-eq-eq plus-1-eq-Suc trans-le-add1)
qed

lemma *elementary-reduction-nonincreases-mtp*:
shows ⟦*elementary-reduction u*; *u ⊑ t*⟧ *⟹ mtp x (resid t u) ≤ mtp x t*
proof (*induct t arbitrary*: *u x*)
  show *⋀u x.* ⟦*elementary-reduction u*; *u ⊑ ♯*⟧ *⟹ mtp x (resid ♯ u) ≤ mtp x ♯*
    by *simp*
  show *⋀x u i.* ⟦*elementary-reduction u*; *u ⊑ «i»*⟧
                  ⟹ mtp x (resid «i» u) ≤ mtp x «i»
    by (*meson Ide.simps*(*2*) *elementary-reduction-not-ide ide-backward-stable ide-char*
      subs-implies-prfx)
  fix *u*
  show *⋀t x.* ⟦*⋀u x.* ⟦*elementary-reduction u*; *u ⊑ t*⟧ *⟹ mtp x (resid t u) ≤ mtp x t*;
            elementary-reduction u; *u ⊑ λ[t]*⟧
              ⟹ mtp x (λ[t] \ u) ≤ mtp x λ[t]
    by (*cases u*) *auto*
  show *⋀t1 t2 x.*
        ⟦*⋀u x.* ⟦*elementary-reduction u*; *u ⊑ t1*⟧ *⟹ mtp x (resid t1 u) ≤ mtp x t1*;
         ⋀u x. ⟦*elementary-reduction u*; *u ⊑ t2*⟧ *⟹ mtp x (resid t2 u) ≤ mtp x t2*;
         elementary-reduction u; *u ⊑ t1 ∘ t2*⟧
           ⟹ mtp x (resid (t1 ∘ t2) u) ≤ mtp x (t1 ∘ t2)
    apply (*cases u*)
      apply *auto*
     apply (*metis Coinitial-iff-Con add-mono-thms-linordered-semiring*(*3*) *resid-Arr-Ide*)
    by (*metis Coinitial-iff-Con add-mono-thms-linordered-semiring*(*2*) *resid-Arr-Ide*)

  show *⋀t1 t2 x.*
        ⟦*⋀u1 x.* ⟦*elementary-reduction u1*; *u1 ⊑ t1*⟧ *⟹ mtp x (resid t1 u1) ≤ mtp x t1*;
         ⋀u2 x. ⟦*elementary-reduction u2*; *u2 ⊑ t2*⟧ *⟹ mtp x (resid t2 u2) ≤ mtp x t2*;
         elementary-reduction u; *u ⊑ λ[t1] • t2*⟧
           ⟹ mtp x ((λ[t1] • t2) \ u) ≤ mtp x (λ[t1] • t2)
  proof −

fix *t1 t2 x*
assume *ind1*: $\bigwedge u1\ x.\ [\![elementary\text{-}reduction\ u1;\ u1 \sqsubseteq t1]\!] \Longrightarrow mtp\ x\ (t1 \setminus u1) \leq mtp\ x\ t1$
assume *ind2*: $\bigwedge u2\ x.\ [\![elementary\text{-}reduction\ u2;\ u2 \sqsubseteq t2]\!] \Longrightarrow mtp\ x\ (t2 \setminus u2) \leq mtp\ x\ t2$
assume *u*: *elementary-reduction u*
assume *subs*: $u \sqsubseteq \boldsymbol{\lambda}[t1] \bullet t2$
have *1*: *is-App u* $\vee$ *is-Beta u*
  using *subs* **by** (*metis prfx-Beta-iff subs-implies-prfx*)
have *is-App u* $\Longrightarrow$ *mtp x* $((\boldsymbol{\lambda}[t1] \bullet t2) \setminus u) \leq mtp\ x\ (\boldsymbol{\lambda}[t1] \bullet t2)$
proof −
  assume *2*: *is-App u*
  obtain *u1 u2* **where** *u1u2*: $u = \boldsymbol{\lambda}[u1] \circ u2$
    using *2 u*
 by (*metis ConD*(*3*) *Con-implies-is-Lam-iff-is-Lam Con-sym con-def is-App-def is-Lam-def*
          lambda.disc(*8*) *null-char prfx-implies-con subs subs-implies-prfx*)
  have *mtp x* $((\boldsymbol{\lambda}[t1] \bullet t2) \setminus u) = mtp\ x\ (\boldsymbol{\lambda}[t1 \setminus u1] \bullet (t2 \setminus u2))$
    using *u1u2 subs*
    by (*metis Con-sym Ide.simps*(*1*) *ide-char resid.simps*(*6*) *subs-implies-prfx*)
  also have ... = *mtp* (*Suc x*) (*resid t1 u1*) +
                  mtp x (*resid t2 u2*) ∗ *max 1* (*mtp 0* (*resid t1 u1*))
    by *simp*
  also have ... ≤ *mtp* (*Suc x*) *t1* + *mtp x* (*resid t2 u2*) ∗ *max 1* (*mtp 0* (*resid t1 u1*))
    using *u1u2 ind1* [*of u1 Suc x*] *con-sym ide-char resid-arr-ide prfx-implies-con*
        subs subs-implies-prfx u
    by *force*
  also have ... ≤ *mtp* (*Suc x*) *t1* + *mtp x t2* ∗ *max 1* (*mtp 0* (*resid t1 u1*))
    using *u1u2 ind2* [*of u2 x*]
    by (*metis* (*no-types*, *lifting*) *Con-implies-Coinitial-ind add-left-mono*
       dual-order.eq-iff elementary-reduction.simps(*4*) *lambda.disc*(*11*)
       mult-le-cancel2 prfx-App-iff resid.simps(*31*) *resid-Arr-Ide subs subs.simps*(*4*)
       subs-implies-prfx u)
  also have ... ≤ *mtp* (*Suc x*) *t1* + *mtp x t2* ∗ *max 1* (*mtp 0 t1*)
    using *ind1* [*of u1 0*]
    by (*metis Con-implies-Coinitial-ind Ide.simps*(*3*) *elementary-reduction.simps*(*3*)
       elementary-reduction.simps(*4*) *lambda.disc*(*11*) *max.mono mult-le-mono*
       nat-add-left-cancel-le nat-le-linear prfx-App-iff resid.simps(*31*) *resid-Arr-Ide*
       subs subs.simps(*4*) *subs-implies-prfx u u1u2*)
  also have ... = *mtp x* $(\boldsymbol{\lambda}[t1] \bullet t2)$
    by *auto*
  finally show *mtp x* $((\boldsymbol{\lambda}[t1] \bullet t2) \setminus u) \leq mtp\ x\ (\boldsymbol{\lambda}[t1] \bullet t2)$ **by** *blast*
qed
moreover have *is-Beta u* $\Longrightarrow$ *mtp x* $((\boldsymbol{\lambda}[t1] \bullet t2) \setminus u) \leq mtp\ x\ (\boldsymbol{\lambda}[t1] \bullet t2)$
proof −
  assume *2*: *is-Beta u*
  obtain *u1 u2* **where** *u1u2*: $u = \boldsymbol{\lambda}[u1] \bullet u2$
    using *2 u is-Beta-def* **by** *auto*
  have *mtp x* $((\boldsymbol{\lambda}[t1] \bullet t2) \setminus u) = mtp\ x\ (subst\ (t2 \setminus u2)\ (t1 \setminus u1))$
    using *u1u2 subs*

```
      by (metis con-def con-sym null-char prfx-implies-con resid.simps(4) subs-implies-prfx)
    also have ... ≤ mtp (Suc x) (resid t1 u1) +
                    mtp x (resid t2 u2) * max 1 (mtp 0 (resid t1 u1))
      apply (cases x = 0)
      using mtp0-subst-le Raise-0 mtp-Subst' [of x − 1 0 resid t2 u2 resid t1 u1]
      by auto
    also have ... ≤ mtp (Suc x) t1 + mtp x t2 * max 1 (mtp 0 t1)
      using ind1 ind2
      apply simp
    by (metis Coinitial-iff-Con Ide.simps(1) dual-order.eq-iff elementary-reduction.simps(5)
        ide-char resid.simps(4) resid-Arr-Ide subs subs-implies-prfx u u1u2)
    also have ... = mtp x (λ[t1] • t2)
      by simp
    finally show mtp x ((λ[t1] • t2) \ u) ≤ mtp x (λ[t1] • t2) by blast
  qed
  ultimately show mtp x ((λ[t1] • t2) \ u) ≤ mtp x (λ[t1] • t2)
    using 1 by blast
 qed
qed
```

Next we define the "height" of a term. This counts the number of steps in a development of maximal length of the given term.

```
fun hgt
where hgt ♯ = 0
    | hgt «-» = 0
    | hgt λ[t] = hgt t
    | hgt (t ∘ u) = hgt t + hgt u
    | hgt (λ[t] • u) = Suc (hgt t + hgt u * max 1 (mtp 0 t))

lemma hgt-resid-ide:
shows ⟦ide u; u ⊑ t⟧ ⟹ hgt (resid t u) ≤ hgt t
  by (metis con-sym eq-imp-le resid-arr-ide prfx-implies-con subs-implies-prfx)

lemma hgt-Raise:
shows hgt (Raise l k t) = hgt t
  using mtpE-eq-Raise
  by (induct t arbitrary: l k) auto

lemma hgt-Subst:
shows Arr u ⟹ hgt (Subst k u t) = hgt t + hgt u * mtp k t
proof (induct t arbitrary: u k)
  show ⋀u k. Arr u ⟹ hgt (Subst k u ♯) = hgt ♯ + hgt u * mtp k ♯
    by simp
  show ⋀x u k. Arr u ⟹ hgt (Subst k u «x») = hgt «x» + hgt u * mtp k «x»
    using hgt-Raise by auto
  show ⋀t u k. ⟦⋀u k. Arr u ⟹ hgt (Subst k u t) = hgt t + hgt u * mtp k t; Arr u⟧
                ⟹ hgt (Subst k u λ[t]) = hgt λ[t] + hgt u * mtp k λ[t]
    by auto
  show ⋀t1 t2 u k.
```

```
      ⟦⋀u k. Arr u ⟹ hgt (Subst k u t1) = hgt t1 + hgt u * mtp k t1;
       ⋀u k. Arr u ⟹ hgt (Subst k u t2) = hgt t2 + hgt u * mtp k t2; Arr u⟧
          ⟹ hgt (Subst k u (t1 ∘ t2)) = hgt (t1 ∘ t2) + hgt u * mtp k (t1 ∘ t2)
  by (simp add: distrib-left)
show ⋀t1 t2 u k.
      ⟦⋀u k. Arr u ⟹ hgt (Subst k u t1) = hgt t1 + hgt u * mtp k t1;
       ⋀u k. Arr u ⟹ hgt (Subst k u t2) = hgt t2 + hgt u * mtp k t2; Arr u⟧
        ⟹ hgt (Subst k u (λ[t1] • t2)) = hgt (λ[t1] • t2) + hgt u * mtp k (λ[t1] • t2)
proof −
  fix t1 t2 u k
  assume ind1: ⋀u k. Arr u ⟹ hgt (Subst k u t1) = hgt t1 + hgt u * mtp k t1
  assume ind2: ⋀u k. Arr u ⟹ hgt (Subst k u t2) = hgt t2 + hgt u * mtp k t2
  assume u: Arr u
  show hgt (Subst k u (λ[t1] • t2)) = hgt (λ[t1] • t2) + hgt u * mtp k (λ[t1] • t2)
  proof −
    have hgt (Subst k u (λ[t1] • t2)) =
         Suc (hgt (Subst (Suc k) u t1) +
           hgt (Subst k u t2) * max 1 (mtp 0 (Subst (Suc k) u t1)))
      by simp
    also have ... = Suc ((hgt t1 + hgt u * mtp (Suc k) t1) +
                         (hgt t2 + hgt u * mtp k t2) * max 1 (mtp 0 (Subst (Suc k) u t1)))
      using u ind1 [of u Suc k] ind2 [of u k] by simp
    also have ... = Suc (hgt t1 + hgt t2 * max 1 (mtp 0 (Subst (Suc k) u t1)) +
                         hgt u * mtp (Suc k) t1) +
                         hgt u * mtp k t2 * max 1 (mtp 0 (Subst (Suc k) u t1))
      using comm-semiring-class.distrib by force
    also have ... = Suc (hgt t1 + hgt t2 * max 1 (mtp 0 (Subst (Suc k) u t1)) +
                         hgt u * (mtp (Suc k) t1 +
                                  mtp k t2 * max 1 (mtp 0 (Subst (Suc k) u t1))))
      by (simp add: distrib-left)
    also have ... = Suc (hgt t1 + hgt t2 * max 1 (mtp 0 t1) +
                         hgt u * (mtp (Suc k) t1 +
                                  mtp k t2 * max 1 (mtp 0 t1)))
    proof −
      have mtp 0 (Subst (Suc k) u t1) = mtp 0 t1
        using mtp0-Subst-cancel by auto
      thus ?thesis by simp
    qed
    also have ... = hgt (λ[t1] • t2) + hgt u * mtp k (λ[t1] • t2)
      by simp
    finally show ?thesis by blast
  qed
qed
qed

lemma elementary-reduction-decreases-hgt:
shows ⟦elementary-reduction u; u ⊑ t⟧ ⟹ hgt (t \ u) < hgt t
proof (induct t arbitrary: u)
  show ⋀u. ⟦elementary-reduction u; u ⊑ ♯⟧ ⟹ hgt (♯ \ u) < hgt ♯
```

by *simp*
show $\bigwedge u\ x.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq «x»]\!] \Longrightarrow hgt\ («x» \setminus u) < hgt\ «x»$
using *Ide.simps(2) elementary-reduction-not-ide ide-backward-stable ide-char subs-implies-prfx*
by *blast*
show $\bigwedge t\ u.\ [\![\bigwedge u.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq t]\!] \Longrightarrow hgt\ (t \setminus u) < hgt\ t;$
$elementary\text{-}reduction\ u;\ u \sqsubseteq \boldsymbol{\lambda}[t]]\!]$
$\Longrightarrow hgt\ (\boldsymbol{\lambda}[t] \setminus u) < hgt\ \boldsymbol{\lambda}[t]$
proof −
fix *t u*
assume *ind*: $\bigwedge u.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq t]\!] \Longrightarrow hgt\ (t \setminus u) < hgt\ t$
assume *u*: *elementary-reduction u*
assume *subs*: $u \sqsubseteq \boldsymbol{\lambda}[t]$
show $hgt\ (\boldsymbol{\lambda}[t] \setminus u) < hgt\ \boldsymbol{\lambda}[t]$
using *u subs ind*
apply (*cases u*)
apply *simp-all*
by *fastforce*
qed
show $\bigwedge t1\ t2\ u.$
$[\![\bigwedge u.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq t1]\!] \Longrightarrow hgt\ (t1 \setminus u) < hgt\ t1;$
$\bigwedge u.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq t2]\!] \Longrightarrow hgt\ (t2 \setminus u) < hgt\ t2;$
$elementary\text{-}reduction\ u;\ u \sqsubseteq t1 \circ t2]\!]$
$\Longrightarrow hgt\ ((t1 \circ t2) \setminus u) < hgt\ (t1 \circ t2)$
proof −
fix *t1 t2 u*
assume *ind1*: $\bigwedge u.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq t1]\!] \Longrightarrow hgt\ (t1 \setminus u) < hgt\ t1$
assume *ind2*: $\bigwedge u.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq t2]\!] \Longrightarrow hgt\ (t2 \setminus u) < hgt\ t2$
assume *u*: *elementary-reduction u*
assume *subs*: $u \sqsubseteq t1 \circ t2$
show $hgt\ ((t1 \circ t2) \setminus u) < hgt\ (t1 \circ t2)$
using *u subs ind1 ind2*
apply (*cases u*)
apply *simp-all*
by (*metis add-le-less-mono add-less-le-mono hgt-resid-ide ide-char not-less0 zero-less-iff-neq-zero*)
qed
show $\bigwedge t1\ t2\ u.$
$[\![\bigwedge u.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq t1]\!] \Longrightarrow hgt\ (t1 \setminus u) < hgt\ t1;$
$\bigwedge u.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq t2]\!] \Longrightarrow hgt\ (t2 \setminus u) < hgt\ t2;$
$elementary\text{-}reduction\ u;\ u \sqsubseteq \boldsymbol{\lambda}[t1] \bullet t2]\!]$
$\Longrightarrow hgt\ ((\boldsymbol{\lambda}[t1] \bullet t2) \setminus u) < hgt\ (\boldsymbol{\lambda}[t1] \bullet t2)$
proof −
fix *t1 t2 u*
assume *ind1*: $\bigwedge u.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq t1]\!] \Longrightarrow hgt\ (t1 \setminus u) < hgt\ t1$
assume *ind2*: $\bigwedge u.\ [\![elementary\text{-}reduction\ u;\ u \sqsubseteq t2]\!] \Longrightarrow hgt\ (t2 \setminus u) < hgt\ t2$
assume *u*: *elementary-reduction u*
assume *subs*: $u \sqsubseteq \boldsymbol{\lambda}[t1] \bullet t2$
have *is-App u* ∨ *is-Beta u*

```
  using subs by (metis prfx-Beta-iff subs-implies-prfx)
moreover have is-App u ⟹ hgt ((λ[t1] • t2) \ u) < hgt (λ[t1] • t2)
proof −
  fix u1 u2
  assume 0: is-App u
  obtain u1 u1′ u2 where 1: u = u1 ⋅ u2 ∧ u1 = λ[u1′]
    using u 0
 by (metis ConD(3) Con-implies-is-Lam-iff-is-Lam Con-sym con-def is-App-def is-Lam-def
          null-char prfx-implies-con subs subs-implies-prfx)
  have hgt ((λ[t1] • t2) \ u) = hgt ((λ[t1] • t2) \ (u1 ⋅ u2))
    using 1 by simp
  also have ... = hgt (λ[t1 \ u1′] • t2 \ u2)
    by (metis 1 Con-sym Ide.simps(1) ide-char resid.simps(6) subs subs-implies-prfx)
 also have ... = Suc (hgt (t1 \ u1′) + hgt (t2 \ u2) ∗ max (Suc 0) (mtp 0 (t1 \ u1′)))
    by auto
  also have ... < hgt (λ[t1] • t2)
  proof −
    have elementary-reduction (un-App1 u) ∧ ide (un-App2 u) ∨
         ide (un-App1 u) ∧ elementary-reduction (un-App2 u)
      using u 1 elementary-reduction-App-iff [of u] by simp
    moreover have elementary-reduction (un-App1 u) ∧ ide (un-App2 u) ⟹ ?thesis
    proof −
      assume 2: elementary-reduction (un-App1 u) ∧ ide (un-App2 u)
      have elementary-reduction u1′ ∧ ide (un-App2 u)
        using 1 2 u elementary-reduction-Lam-iff by force
      moreover have mtp 0 (t1 \ u1′) ≤ mtp 0 t1
        using 1 calculation elementary-reduction-nonincreases-mtp subs
              subs.simps(4)
        by blast
      moreover have mtp 0 (t2 \ u2) ≤ mtp 0 t2
        using 1 hgt-resid-ide [of u2 t2]
        by (metis calculation(1) con-sym eq-refl resid-arr-ide lambda.sel(4)
            prfx-implies-con subs subs.simps(4) subs-implies-prfx)
      ultimately show ?thesis
        using 1 2 ind1 [of u1′] hgt-resid-ide
        apply simp
        by (metis 1 Suc-le-mono ‹mtp 0 (t1 \ u1′) ≤ mtp 0 t1› add-less-le-mono
            le-add1 le-add-same-cancel1 max.mono mult-le-mono subs subs.simps(4))
    qed
    moreover have ide (un-App1 u) ∧ elementary-reduction (un-App2 u) ⟹ ?thesis
    proof −
      assume 2: ide (un-App1 u) ∧ elementary-reduction (un-App2 u)
      have ide (un-App1 u) ∧ elementary-reduction u2
        using 1 2 u elementary-reduction-Lam-iff by force
      moreover have mtp 0 (t1 \ u1′) ≤ mtp 0 t1
        using 1 hgt-resid-ide [of u1′ t1]
        by (metis Ide.simps(3) calculation con-sym eq-refl ide-char resid-arr-ide
            lambda.sel(3) prfx-implies-con subs subs.simps(4) subs-implies-prfx)
      moreover have mtp 0 (t2 \ u2) ≤ mtp 0 t2
```

```
      using 1 elementary-reduction-nonincreases-mtp subs calculation(1) subs.simps(4)
      by blast
    ultimately show ?thesis
      using 1 2 ind2 [of u2]
      apply simp
      by (metis Coinitial-iff-Con Ide-iff-Src-self Nat.add-0-right add-le-less-mono
          ide-char Ide.simps(1) subs.simps(4) le-add1 max-nat.neutr-eq-iff
          mult-less-cancel2 nat.distinct(1) neq0-conv resid-Arr-Src subs
          subs-implies-prfx)
  qed
  ultimately show ?thesis by blast
qed
also have ... = Suc (hgt t1 + hgt t2 * max 1 (mtp 0 t1))
  by simp
also have ... = hgt (λ[t1] • t2)
  by simp
finally show hgt ((λ[t1] • t2) \ u) < hgt (λ[t1] • t2)
  by blast
qed
moreover have is-Beta u ⟹ hgt ((λ[t1] • t2) \ u) < hgt (λ[t1] • t2)
proof −
  fix u1 u2
  assume 0: is-Beta u
  obtain u1 u2 where 1: u = λ[u1] • u2
    using u 0 by (metis lambda.collapse(4))
  have hgt ((λ[t1] • t2) \ u) = hgt ((λ[t1] • t2) \ (λ[u1] • u2))
    using 1 by simp
  also have ... = hgt (subst (resid t2 u2) (resid t1 u1))
    by (metis 1 con-def con-sym null-char prfx-implies-con resid.simps(4)
        subs subs-implies-prfx)
  also have ... = hgt (resid t1 u1) + hgt (resid t2 u2) * mtp 0 (resid t1 u1)
  proof −
    have Arr (resid t2 u2)
      by (metis 1 Coinitial-resid-resid Con-sym Ide.simps(1) ide-char resid.simps(4)
          subs subs-implies-prfx)
    thus ?thesis
      using hgt-Subst [of resid t2 u2 0 resid t1 u1] by simp
  qed
  also have ... < hgt (λ[t1] • t2)
  proof −
    have ide u1 ∧ ide u2
      using u 1 elementary-reduction-Beta-iff [of u] by auto
   thus ?thesis
     using 1 hgt-resid-ide
     by (metis add-le-mono con-sym hgt.simps(5) resid-arr-ide less-Suc-eq-le
         max.cobounded2 nat-mult-max-right prfx-implies-con subs subs.simps(5)
         subs-implies-prfx)
  qed
  finally show hgt ((λ[t1] • t2) \ u) < hgt (λ[t1] • t2)
```

```
      by blast
    qed
    ultimately show hgt ((λ[t1] • t2) \ u) < hgt (λ[t1] • t2) by blast
  qed
qed

end

context reduction-paths
begin

  lemma length-devel-le-hgt:
  shows development t U ⟹ length U ≤ Λ.hgt t
    using Λ.elementary-reduction-decreases-hgt
    by (induct U arbitrary: t, auto, fastforce)
```

We finally arrive at the main result of this section: the Finite Developments Theorem.

```
  theorem finite-developments:
  shows FD t
    using length-devel-le-hgt [of t] FD-def by auto
```

3.4.2 Complete Developments

A *complete development* is a development in which there are no residuals of originally marked redexes left to contract.

```
  definition complete-development
  where complete-development t U ≡ development t U ∧ (Λ.Ide t ∨ [t] *≲* U)

  lemma complete-development-Ide-iff:
  shows complete-development t U ⟹ Λ.Ide t ⟷ U = []
    using complete-development-def development-Ide Ide.simps(1) ide-char
    by (induct t) auto

  lemma complete-development-cons:
  assumes complete-development t (u # U)
  shows complete-development (t \ u) U
    using assms complete-development-def
    by (metis Ide.simps(1) Ide.simps(2) Resid-rec(1) Resid-rec(3)
        complete-development-Ide-iff ide-char development.simps(2)
        Λ.ide-char list.simps(3))

  lemma complete-development-cong:
  shows ⟦complete-development t U; ¬ Λ.Ide t⟧ ⟹ [t] *∼* U
    using complete-development-def development-implies
    by (induct U) auto

  lemma complete-developments-cong:
  assumes ¬ Λ.Ide t and complete-development t U and complete-development t V
  shows U *∼* V
```

using *assms complete-development-cong* [*of t*] *cong-symmetric cong-transitive*
by *blast*

lemma *Trgs-complete-development*:
shows ⟦*complete-development t U*; ¬ Λ.*Ide t*⟧ ⟹ *Trgs U* = {Λ.*Trg t*}
using *complete-development-cong Ide.simps*(*1*) *Srcs-Resid Trgs.simps*(*2*)
Trgs-Resid-sym ide-char complete-development-def development-imp-Arr Λ.*targets-char*$_\Lambda$
apply *simp*
by (*metis Srcs-Resid Trgs.simps*(*2*) *con-char ide-def*)

Now that we know all developments are finite, it is easy to construct a complete development by an iterative process that at each stage contracts one of the remaining marked redexes at each stage. It is also possible to construct a complete development by structural induction without using the finite developments property, but it is more work to prove the correctness.

fun (**in** *lambda-calculus*) *bottom-up-redex*
where *bottom-up-redex* ♯ = ♯
| *bottom-up-redex «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*)
using *Ide-Src*
by (*induct t*) *auto*

lemma (**in** *lambda-calculus*) *subs-bottom-up-redex*:
shows *Arr t* ⟹ *bottom-up-redex t* ⊑ *t*
apply (*induct t*)
apply *auto*[*3*]
apply (*metis Arr.simps*(*4*) *Ide.simps*(*4*) *Ide-Src Ide-iff-Src-self Ide-implies-Arr*
bottom-up-redex.simps(*4*) *ide-char lambda.disc*(*14*) *lambda.sel*(*3*) *lambda.sel*(*4*)
subs-App subs-Ide)
by (*metis Arr.simps*(*5*) *Ide-Src Ide-iff-Src-self Ide-implies-Arr bottom-up-redex.simps*(*5*)
ide-char subs.simps(*4*) *subs.simps*(*5*) *subs-Ide*)

function (*sequential*) *bottom-up-development*
where *bottom-up-development t* =
(*if* ¬ Λ.*Arr t* ∨ Λ.*Ide t then* []
else Λ.*bottom-up-redex t* # (*bottom-up-development* (*t* \ Λ.*bottom-up-redex t*)))
by *pat-completeness auto*

termination *bottom-up-development*
using Λ.*elementary-reduction-decreases-hgt* Λ.*elementary-reduction-bottom-up-redex*

```
      Λ.subs-bottom-up-redex
  by (relation measure Λ.hgt) auto

lemma complete-development-bottom-up-development-ind:
shows ⟦Λ.Arr t; length (bottom-up-development t) ≤ n⟧
        ⟹ complete-development t (bottom-up-development t)
proof (induct n arbitrary: t)
  show ⋀t. ⟦Λ.Arr t; length (bottom-up-development t) ≤ 0⟧
             ⟹ complete-development t (bottom-up-development t)
    using complete-development-def development-Ide by auto
  show ⋀n t. ⟦⋀t. ⟦Λ.Arr t; length (bottom-up-development t) ≤ n⟧
                    ⟹ complete-development t (bottom-up-development t);
              Λ.Arr t; length (bottom-up-development t) ≤ Suc n⟧
                ⟹ complete-development t (bottom-up-development t)
  proof −
    fix n t
    assume t: Λ.Arr t
    assume n: length (bottom-up-development t) ≤ Suc n
    assume ind: ⋀t. ⟦Λ.Arr t; length (bottom-up-development t) ≤ n⟧
                    ⟹ complete-development t (bottom-up-development t)
    show complete-development t (bottom-up-development t)
    proof (cases bottom-up-development t)
      show bottom-up-development t = [] ⟹ ?thesis
        using ind t by force
      fix u U
      assume uU: bottom-up-development t = u # U
      have 1: Λ.elementary-reduction u ∧ u ⊑ t
        using t uU
        by (metis bottom-up-development.simps Λ.elementary-reduction-bottom-up-redex
            list.inject list.simps(3) Λ.subs-bottom-up-redex)
      moreover have complete-development (Λ.resid t u) U
        using 1 ind
       by (metis Suc-le-length-iff Λ.arr-char Λ.arr-resid-iff-con bottom-up-development.simps
                 list.discI list.inject n not-less-eq-eq Λ.prfx-implies-con
                 Λ.con-sym Λ.subs-implies-prfx uU)
      ultimately show ?thesis
        by (metis Con-sym Ide.simps(2) Resid-rec(1) Resid-rec(3)
            complete-development-Ide-iff complete-development-def ide-char
            development.simps(2) development-implies Λ.ide-char list.simps(3) uU)
    qed
  qed
qed

lemma complete-development-bottom-up-development:
assumes Λ.Arr t
shows complete-development t (bottom-up-development t)
  using assms complete-development-bottom-up-development-ind by blast

end
```

3.5 Reduction Strategies

context *lambda-calculus*
begin

A *reduction strategy* is a function taking an identity term to an arrow having that identity as its source.

definition *reduction-strategy*
where *reduction-strategy f* $\longleftrightarrow$ $(\forall\, t.\ Ide\ t \longrightarrow Coinitial\ (f\ t)\ t)$

The following defines the iterated application of a reduction strategy to an identity term.

fun *reduce*
where *reduce f a 0* $=$ *a*
| *reduce f a* $(Suc\ n) = reduce\ f\ (Trg\ (f\ a))\ n$

lemma *red-reduce*:
assumes *reduction-strategy f*
shows *Ide a* $\Longrightarrow$ *red a* $(reduce\ f\ a\ n)$
apply $(induct\ n\ arbitrary{:}\ a,\ auto)$
apply $(metis\ Ide\text{-}iff\text{-}Src\text{-}self\ Ide\text{-}iff\text{-}Trg\text{-}self\ Ide\text{-}implies\text{-}Arr\ red.simps)$
by $(metis\ Ide\text{-}Trg\ Ide\text{-}iff\text{-}Src\text{-}self\ assms\ red.intros(1)\ red.intros(2)\ reduction\text{-}strategy\text{-}def)$

A reduction strategy is *normalizing* if iterated application of it to a normalizable term eventually yields a normal form.

definition *normalizing-strategy*
where *normalizing-strategy f* $\longleftrightarrow$ $(\forall\, a.\ normalizable\ a \longrightarrow (\exists\, n.\ NF\ (reduce\ f\ a\ n)))$

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-strategy f a 0* $=$ $[]$
| *apply-strategy f a* $(Suc\ n) = f\ a\ \#\ apply\text{-}strategy\ f\ (\Lambda.Trg\ (f\ a))\ n$

lemma *apply-strategy-gives-path-ind*:
assumes Λ*.reduction-strategy f*
shows $[\![\Lambda.Ide\ a;\ n > 0]\!] \Longrightarrow Arr\ (apply\text{-}strategy\ f\ a\ n)\ \wedge$
$length\ (apply\text{-}strategy\ f\ a\ n) = n\ \wedge$
$Src\ (apply\text{-}strategy\ f\ a\ n) = a\ \wedge$
$Trg\ (apply\text{-}strategy\ f\ a\ n) = \Lambda.reduce\ f\ a\ n$
proof $(induct\ n\ arbitrary{:}\ a,\ simp)$
fix *n a*
assume *ind*: $\bigwedge a.\ [\![\Lambda.Ide\ a;\ 0 < n]\!] \Longrightarrow Arr\ (apply\text{-}strategy\ f\ a\ n)\ \wedge$
$length\ (apply\text{-}strategy\ f\ a\ n) = n\ \wedge$

```
                                    Src (apply-strategy f a n) = a ∧
                                    Trg (apply-strategy f a n) = Λ.reduce f a n
  assume a: Λ.Ide a
  show Arr (apply-strategy f a (Suc n)) ∧
       length (apply-strategy f a (Suc n)) = Suc n ∧
       Src (apply-strategy f a (Suc n)) = a ∧
       Trg (apply-strategy f a (Suc n)) = Λ.reduce f a (Suc n)
  proof (intro conjI)
    have 1: Λ.Arr (f a) ∧ Λ.Src (f a) = a
      using assms a Λ.reduction-strategy-def
      by (metis Λ.Ide-iff-Src-self)
    show Arr (apply-strategy f a (Suc n))
      using 1 Arr.elims(3) ind Λ.targets-charΛ Λ.Ide-Trg by fastforce
    show Src (apply-strategy f a (Suc n)) = a
      by (simp add: 1)
    show length (apply-strategy f a (Suc n)) = Suc n
      by (metis 1 Λ.Ide-Trg One-nat-def Suc-eq-plus1 ind list.size(3) list.size(4)
          neq0-conv apply-strategy.simps(1) apply-strategy.simps(2))
    show Trg (apply-strategy f a (Suc n)) = Λ.reduce f a (Suc n)
    proof (cases apply-strategy f (Λ.Trg (f a)) n = [])
      show apply-strategy f (Λ.Trg (f a)) n = [] ⟹ ?thesis
        using a 1 ind [of Λ.Trg (f a)] Λ.Ide-Trg Λ.targets-charΛ by force
      assume 2: apply-strategy f (Λ.Trg (f a)) n ≠ []
      have Trg (apply-strategy f a (Suc n)) = Trg (apply-strategy f (Λ.Trg (f a)) n)
        using a 1 ind [of Λ.Trg (f a)]
        by (simp add: 2)
      also have ... = Λ.reduce f a (Suc n)
        using 1 2 Λ.Ide-Trg ind [of Λ.Trg (f a)] by fastforce
      finally show ?thesis by blast
    qed
  qed
qed

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
  using assms apply-strategy-gives-path-ind by auto

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)
  using assms
  apply (induct n)
   apply simp-all
  by (metis Arr.simps(1) Trg-simp apply-strategy-gives-path-ind Λ.Ide-Trg
      Λ.reduce.simps(1) Λ.reduction-strategy-def Λ.trg-char neq0-conv
```

apply-strategy.simps(*1*))

end

3.5.1 Parallel Reduction

context *lambda-calculus*
begin

Parallel reduction is the strategy that contracts all available redexes at each step.

fun *parallel-strategy*
where *parallel-strategy «i»* = *«i»*
| *parallel-strategy* **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 (*unfold reduction-strategy-def*, *intro allI impI*)
fix *t*
show *Ide t* ⟹ *Coinitial* (*parallel-strategy t*) *t*
using *Ide-implies-Arr*
apply (*induct t*, *auto*)
by *force*+
qed

lemma *parallel-strategy-Src-eq*:
shows *Arr t* ⟹ *parallel-strategy* (*Src t*) = *parallel-strategy t*
by (*induct t*) *auto*

lemma *subs-parallel-strategy-Src*:
shows *Arr t* ⟹ *t* ⊑ *parallel-strategy* (*Src t*)
by (*induct t*) *auto*

end

context *reduction-paths*
begin

Parallel reduction is a universal strategy in the sense that every reduction path is $^*\lesssim^*$-below the path generated by the parallel reduction strategy.

lemma *parallel-strategy-is-universal*:
shows ⟦*n* > *0*; *n* ≤ *length U*; *Arr U*⟧
⟹ *take n U* $^*\lesssim^*$ *apply-strategy* Λ.*parallel-strategy* (*Src U*) *n*
proof (*induct n arbitrary*: *U*, *simp*)
fix *n a* **and** *U* :: Λ.*lambda list*
assume *n*: *Suc n* ≤ *length U*

assume *U*: *Arr U*
assume *ind*: $\bigwedge U.\ [\![0 < n;\ n \le length\ U;\ Arr\ U]\!]$
$\qquad \Longrightarrow take\ n\ U\ {}^*\!\lesssim^*\ apply\text{-}strategy\ \Lambda.parallel\text{-}strategy\ (Src\ U)\ n$
have *1*: *take* (*Suc n*) *U* = *hd U* # *take n* (*tl U*)
by (*metis U Arr.simps*(*1*) *take-Suc*)
have *2*: *hd U* $\sqsubseteq$ Λ.*parallel-strategy* (*Src U*)
by (*metis Arr-imp-arr-hd Con-single-ideI*(*2*) *Resid-Arr-Src Src-resid Srcs-simp*$_{\Lambda P}$
Trg.simps(*2*) *U* Λ.*source-is-ide* Λ.*trg-ide empty-set* Λ.*arr-char* Λ.*sources-char*$_\Lambda$
Λ.*subs-parallel-strategy-Src list.set-intros*(*1*) *list.simps*(*15*))
show *take* (*Suc n*) *U* ${}^*\!\lesssim^*$ *apply-strategy* Λ.*parallel-strategy* (*Src U*) (*Suc n*)
proof (*cases apply-strategy* Λ.*parallel-strategy* (*Src U*) (*Suc n*))
show *apply-strategy* Λ.*parallel-strategy* (*Src U*) (*Suc n*) = [] $\Longrightarrow$
take (*Suc n*) *U* ${}^*\!\lesssim^*$ *apply-strategy* Λ.*parallel-strategy* (*Src U*) (*Suc n*)
by *simp*
fix *v V*
assume *3*: *apply-strategy* Λ.*parallel-strategy* (*Src U*) (*Suc n*) = *v* # *V*
show *take* (*Suc n*) *U* ${}^*\!\lesssim^*$ *apply-strategy* Λ.*parallel-strategy* (*Src U*) (*Suc n*)
proof (*cases V* = [])
show *V* = [] $\Longrightarrow$ *?thesis*
using *1 2 3 ind ide-char*
by (*metis Suc-inject Ide.simps*(*2*) *Resid.simps*(*3*) *list.discI list.inject*
Λ.*prfx-implies-con apply-strategy.elims* Λ.*subs-implies-prfx take0*)
assume *V*: *V* ≠ []
have *4*: *Arr* (*v* # *V*)
using *3 apply-strategy-gives-path*(*1*)
by (*metis Arr-imp-arr-hd Srcs-simp*$_{PWE}$ *Srcs-simp*$_{\Lambda P}$ *U* Λ.*Ide-Src* Λ.*arr-iff-has-target*
Λ.*parallel-strategy-is-reduction-strategy* Λ.*targets-char*$_\Lambda$ *singleton-insert-inj-eq*$'$
zero-less-Suc)
have *5*: *Arr* (*hd U* # *take n* (*tl U*))
by (*metis 1 U Arr-append-iff*$_P$ *id-take-nth-drop list.discI not-less take-all-iff*)
have *6*: *Srcs* (*hd U* # *take n* (*tl U*)) = *Srcs* (*v* # *V*)
by (*metis 2 3* Λ.*Coinitial-iff-Con* Λ.*Ide.simps*(*1*) *Srcs.simps*(*2*) *Srcs.simps*(*3*)
Λ.*ide-char list.exhaust-sel list.inject apply-strategy.simps*(*2*) Λ.*sources-char*$_\Lambda$
Λ.*subs-implies-prfx*)
have *take* (*Suc n*) *U* ${}^*\backslash^*$ *apply-strategy* Λ.*parallel-strategy* (*Src U*) (*Suc n*) =
[*hd U* \ *v*] ${}^*\backslash^*$ *V* @ (*take n* (*tl U*) ${}^*\backslash^*$ [*v* \ *hd U*]) ${}^*\backslash^*$ (*V* ${}^*\backslash^*$ [*hd U* \ *v*])
using *U V 1 3 4 5 6*
by (*metis Resid.simps*(*1*) *Resid-cons*(*1*) *Resid-rec*(*3*−*4*) *confluence-ind*)
moreover have *Ide ...*
proof
have *7*: *v* = Λ.*parallel-strategy* (*Src U*) ∧
V = *apply-strategy* Λ.*parallel-strategy* (*Src U* \ *v*) *n*
using *3* Λ.*subs-implies-prfx* Λ.*subs-parallel-strategy-Src*
apply *simp*
by (*metis* (*full-types*) Λ.*Coinitial-iff-Con* Λ.*Ide.simps*(*1*) Λ.*Trg.simps*(*5*)
Λ.*parallel-strategy.simps*(*9*) Λ.*resid-Src-Arr*)
show *8*: *Ide* ([*hd U* \ *v*] ${}^*\backslash^*$ *V*)
by (*metis 2 4 5 6 7 V Con-initial-left Ide.simps*(*2*)
confluence-ind Con-rec(*3*) *Resid-Ide-Arr-ind* Λ.*subs-implies-prfx*)

```
show 9: Ide ((take n (tl U) *\* [v \ hd U]) *\* (V *\* [hd U \ v]))
proof −
  have 10: Λ.Ide (hd U \ v)
    using 2 7 Λ.ide-char Λ.subs-implies-prfx by presburger
  have 11: V = apply-strategy Λ.parallel-strategy (Λ.Trg v) n
    using 3 by auto
  have (take n (tl U) *\* [v \ hd U]) *\* (V *\* [hd U \ v]) =
       (take n (tl U) *\* [v \ hd U]) *\*
         apply-strategy Λ.parallel-strategy (Λ.Trg v) n
    by (metis 8 10 11 Ide.simps(1) Resid-single-ide(2) Λ.prfx-char)
  moreover have Ide ...
  proof −
    have Ide (take n (take n (tl U) *\* [v \ hd U]) *\*
                apply-strategy Λ.parallel-strategy (Λ.Trg v) n)
    proof −
      have 0 < n
      proof −
        have length V = n
          using apply-strategy-gives-path
          by (metis 10 11 V Λ.Coinitial-iff-Con Λ.Ide-Trg Λ.Arr-not-Nil
              Λ.Ide-implies-Arr Λ.parallel-strategy-is-reduction-strategy neq0-conv
              apply-strategy.simps(1))
        thus ?thesis
          using V by blast
      qed
      moreover have n ≤ length (take n (tl U) *\* [v \ hd U])
      proof −
        have length (take n (tl U)) = n
          using n by force
        thus ?thesis
          using n U length-Resid [of take n (tl U) [v \ hd U]]
          by (metis 4 5 6 Arr.simps(1) Con-cons(2) Con-rec(2)
              confluence-ind dual-order.eq-iff)
      qed
      moreover have Λ.Trg v = Src (take n (tl U) *\* [v \ hd U])
      proof −
        have Src (take n (tl U) *\* [v \ hd U]) = Trg [v \ hd U]
          by (metis Src-resid calculation(1−2) linorder-not-less list.size(3))
        also have ... = Λ.Trg v
          by (metis 10 Trg.simps(2) Λ.Arr-not-Nil Λ.apex-sym Λ.trg-ide
              Λ.Ide-iff-Src-self Λ.Ide-implies-Arr Λ.Src-resid Λ.prfx-char)
        finally show ?thesis by simp
      qed
      ultimately show ?thesis
        using ind [of Resid (take n (tl U)) [Λ.resid v (hd U)]] ide-char
        by (metis Con-imp-Arr-Resid le-zero-eq less-not-refl list.size(3))
    qed
    moreover have take n (take n (tl U) *\* [v \ hd U]) =
                  take n (tl U) *\* [v \ hd U]
```

```
          proof −
            have Arr (take n (tl U) *\* [v \ hd U])
              by (metis Con-imp-Arr-Resid Con-implies-Arr(1) Ide.simps(1) calculation
                  take-Nil)
            thus ?thesis
              by (metis 1 Arr.simps(1) length-Resid dual-order.eq-iff length-Cons
                    length-take min.absorb2 n old.nat.inject take-all)
          qed
          ultimately show ?thesis by simp
        qed
        ultimately show ?thesis by auto
      qed
      show Trg ([hd U \ v] *\* V) =
           Src ((take n (tl U) *\* [v \ hd U]) *\* (V *\* [hd U \ v]))
        by (metis 9 Ide.simps(1) Src-resid Trg-resid-sym)
    qed
    ultimately show ?thesis
      using ide-char by presburger
  qed
 qed

end

context lambda-calculus
begin
```

Parallel reduction is a normalizing strategy.

```
 lemma parallel-strategy-is-normalizing:
 shows normalizing-strategy parallel-strategy
 proof −
  interpret Λx: reduction-paths .

  have ⋀a. normalizable a ⟹ ∃ n. NF (reduce parallel-strategy a n)
  proof −
    fix a
    assume 1: normalizable a
    obtain U b where U: Λx.Arr U ∧ Λx.Src U = a ∧ Λx.Trg U = b ∧ NF b
      using 1 normalizable-def Λx.red-iff by blast
    have 2: ⋀n. ⟦0 < n; n ≤ length U⟧
                  ⟹ Λx.Ide (Λx.Resid (take n U) (Λx.apply-strategy parallel-strategy a n))
      using U Λx.parallel-strategy-is-universal Λx.ide-char by blast
    let ?PR = Λx.apply-strategy parallel-strategy a (length U)
    have Λx.Trg ?PR = b
    proof −
      have 3: Λx.Ide (Λx.Resid U ?PR)
        using U 2 [of length U] by force
      have Λx.Trg (Λx.Resid ?PR U) = b
      by (metis 3 NF-reduct-is-trivial U Λx.Con-imp-Arr-Resid Λx.Con-sym Λx.Ide.simps(1)
```

```
        Λx.Src-resid reduction-paths.red-iff)
    thus ?thesis
      by (metis 3 Λx.Con-Arr-self Λx.Ide-implies-Arr Λx.Resid-Arr-Ide-ind
          Λx.Src-resid Λx.Trg-resid-sym)
  qed
  hence reduce parallel-strategy a (length U) = b
    using 1 U
    by (metis Λx.Arr.simps(1) length-greater-0-conv normalizable-def
        Λx.apply-strategy-gives-path(4) parallel-strategy-is-reduction-strategy)
  thus ∃n. NF (reduce parallel-strategy a n)
    using U by blast
qed
thus ?thesis
  using normalizing-strategy-def by blast
qed
```

An alternative characterization of a normal form is a term on which the parallel reduction strategy yields an identity.

```
abbreviation has-redex
where has-redex t ≡ Arr t ∧ ¬ Ide (parallel-strategy t)

lemma NF-iff-has-no-redex:
shows Arr t ⟹ NF t ⟷ ¬ has-redex t
proof (induct t)
  show Arr ♯ ⟹ NF ♯ ⟷ ¬ has-redex ♯
    using NF-def by simp
  show ⋀x. Arr «x» ⟹ NF «x» ⟷ ¬ has-redex «x»
    using NF-def by force
  show ⋀t. ⟦Arr t ⟹ NF t ⟷ ¬ has-redex t; Arr λ[t]⟧ ⟹ NF λ[t] ⟷ ¬ has-redex λ[t]
  proof -
    fix t
    assume ind: Arr t ⟹ NF t ⟷ ¬ has-redex t
    assume t: Arr λ[t]
    show NF λ[t] ⟷ ¬ has-redex λ[t]
    proof
      show NF λ[t] ⟹ ¬ has-redex λ[t]
        using t ind
        by (metis NF-def Arr.simps(3) Ide.simps(3) Src.simps(3) parallel-strategy.simps(2))
      show ¬ has-redex λ[t] ⟹ NF λ[t]
        using t ind
        by (metis NF-def ide-backward-stable ide-char parallel-strategy-Src-eq
            subs-implies-prfx subs-parallel-strategy-Src)
    qed
  qed
  show ⋀t1 t2. ⟦Arr t1 ⟹ NF t1 ⟷ ¬ has-redex t1;
                Arr t2 ⟹ NF t2 ⟷ ¬ has-redex t2;
                Arr (λ[t1] • t2)⟧
                  ⟹ NF (λ[t1] • t2) ⟷ ¬ has-redex (λ[t1] • t2)
    using NF-def Ide.simps(5) parallel-strategy.simps(8) by presburger
```

```
  show ⋀t1 t2. ⟦Arr t1 ⟹ NF t1 ⟷ ¬ has-redex t1;
                Arr t2 ⟹ NF t2 ⟷ ¬ has-redex t2;
                Arr (t1 ∘ t2)⟧
                  ⟹ NF (t1 ∘ t2) ⟷ ¬ has-redex (t1 ∘ t2)
  proof −
    fix t1 t2
    assume ind1: Arr t1 ⟹ NF t1 ⟷ ¬ has-redex t1
    assume ind2: Arr t2 ⟹ NF t2 ⟷ ¬ has-redex t2
    assume t: Arr (t1 ∘ t2)
    show NF (t1 ∘ t2) ⟷ ¬ has-redex (t1 ∘ t2)
      using t ind1 ind2 NF-def
      apply (intro iffI)
       apply (metis Ide-iff-Src-self parallel-strategy-is-reduction-strategy
          reduction-strategy-def)
      apply (cases t1)
          apply simp-all
       apply (metis Ide-iff-Src-self ide-char parallel-strategy.simps(1,5)
          parallel-strategy-is-reduction-strategy reduction-strategy-def resid-Arr-Src
          subs-implies-prfx subs-parallel-strategy-Src)
      by (metis Ide-iff-Src-self ide-char ind1 Arr.simps(4) parallel-strategy.simps(6)
          parallel-strategy-is-reduction-strategy reduction-strategy-def resid-Arr-Src
          subs-implies-prfx subs-parallel-strategy-Src)
  qed
qed

lemma (in lambda-calculus) not-NF-elim:
assumes ¬ NF t and Ide t
obtains u where coinitial t u ∧ ¬ Ide u
  using assms NF-def by auto

lemma (in lambda-calculus) NF-Lam-iff:
shows NF λ[t] ⟷ NF t
  using NF-def
  by (metis Ide-implies-Arr NF-iff-has-no-redex Ide.simps(3) parallel-strategy.simps(2))

lemma (in lambda-calculus) NF-App-iff:
shows NF (t1 ∘ t2) ⟷ ¬ is-Lam t1 ∧ NF t1 ∧ NF t2
proof −
  have ¬ NF (t1 ∘ t2) ⟹ is-Lam t1 ∨ ¬ NF t1 ∨ ¬ NF t2
    apply (cases is-Lam t1)
     apply simp-all
    apply (cases t1)
       apply simp-all
    using NF-def Ide.simps(1) apply presburger
      apply (metis Ide-implies-Arr NF-def NF-iff-has-no-redex Ide.simps(4)
        parallel-strategy.simps(5))
    apply (metis Ide-implies-Arr NF-def NF-iff-has-no-redex Ide.simps(4)
        parallel-strategy.simps(6))
    using NF-def Ide.simps(5) by presburger
```

```
  moreover have is-Lam t1 ∨ ¬ NF t1 ∨ ¬ NF t2 ⟹ ¬ NF (t1 ∘ t2)
  proof −
    have is-Lam t1 ⟹ ¬NF (t1 ∘ t2)
      by (metis Ide-implies-Arr NF-def NF-iff-has-no-redex Ide.simps(5) lambda.collapse(2)
          parallel-strategy.simps(3,8))
    moreover have ¬ NF t1 ⟹ ¬NF (t1 ∘ t2)
      using NF-def Ide-iff-Src-self Ide-implies-Arr
      apply auto
      by (metis (full-types) Arr.simps(4) Ide.simps(4) Src.simps(4))
    moreover have ¬ NF t2 ⟹ ¬NF (t1 ∘ t2)
      using NF-def Ide-iff-Src-self Ide-implies-Arr
      apply auto
      by (metis (full-types) Arr.simps(4) Ide.simps(4) Src.simps(4))
    ultimately show is-Lam t1 ∨ ¬ NF t1 ∨ ¬ NF t2 ⟹ ¬ NF (t1 ∘ t2)
      by auto
  qed
  ultimately show ?thesis by blast
qed
```

3.5.2 Head Reduction

Head reduction is the strategy that only contracts a redex at the "head" position, which is found at the end of the "left spine" of applications, and does nothing if there is no such redex.

The following function applies to an arbitrary arrow t, and it marks the redex at the head position, if any, otherwise it yields *Src t*.

```
fun head-strategy
where head-strategy «i» = «i»
    | head-strategy λ[t] = λ[head-strategy t]
    | head-strategy (λ[t] ∘ u) = λ[Src t] • Src u
    | head-strategy (t ∘ u) = head-strategy t ∘ Src u
    | head-strategy (λ[t] • u) = λ[Src t] • Src u
    | head-strategy ♯ = ♯

lemma Arr-head-strategy:
shows Arr t ⟹ Arr (head-strategy t)
  apply (induct t)
     apply auto
proof −
  fix t u
  assume ind: Arr (head-strategy t)
  assume t: Arr t and u: Arr u
  show Arr (head-strategy (t ∘ u))
    using t u ind
    by (cases t) auto
qed

lemma Src-head-strategy:
```

shows *Arr t* $\Longrightarrow$ *Src* (*head-strategy t*) = *Src t*
apply (*induct t*)
apply *auto*
proof −
fix *t u*
assume *ind*: *Src* (*head-strategy t*) = *Src t*
assume *t*: *Arr t* **and** *u*: *Arr u*
have *Src* (*head-strategy* (*t* ∘ *u*)) = *Src* (*head-strategy t* ∘ *Src u*)
using *t ind*
by (*cases t*) *auto*
also have ... = *Src t* ∘ *Src u*
using *t u ind* **by** *auto*
finally show *Src* (*head-strategy* (*t* ∘ *u*)) = *Src t* ∘ *Src u* **by** *simp*
qed

lemma *Con-head-strategy*:
shows *Arr t* $\Longrightarrow$ *Con t* (*head-strategy t*)
apply (*induct t*)
apply *auto*
apply (*simp add*: *Arr-head-strategy Src-head-strategy*)
using *Arr-Subst Arr-not-Nil* **by** *auto*

lemma *head-strategy-Src*:
shows *Arr t* $\Longrightarrow$ *head-strategy* (*Src t*) = *head-strategy t*
apply (*induct t*)
apply *auto*
using *Arr.elims*(*2*) **by** *fastforce*

lemma *head-strategy-is-elementary*:
shows ⟦*Arr t*; ¬ *Ide* (*head-strategy t*)⟧ $\Longrightarrow$ *elementary-reduction* (*head-strategy t*)
using *Ide-Src*
apply (*induct t*)
apply *auto*
proof −
fix *t1 t2*
assume *t1*: *Arr t1* **and** *t2*: *Arr t2*
assume *t*: ¬ *Ide* (*head-strategy* (*t1* ∘ *t2*))
assume *1*: ¬ *Ide* (*head-strategy t1*) $\Longrightarrow$ *elementary-reduction* (*head-strategy t1*)
assume *2*: ¬ *Ide* (*head-strategy t2*) $\Longrightarrow$ *elementary-reduction* (*head-strategy t2*)
show *elementary-reduction* (*head-strategy* (*t1* ∘ *t2*))
using *t t1 t2 1 2 Ide-Src Ide-implies-Arr*
by (*cases t1*) *auto*
qed

lemma *head-strategy-is-reduction-strategy*:
shows *reduction-strategy head-strategy*
proof (*unfold reduction-strategy-def*, *intro allI impI*)
fix *t*
show *Ide t* $\Longrightarrow$ *Coinitial* (*head-strategy t*) *t*

```
  proof (induct t)
    show Ide ♯ ⟹ Coinitial (head-strategy ♯) ♯
      by simp
    show ⋀x. Ide «x» ⟹ Coinitial (head-strategy «x») «x»
      by simp
    show ⋀t. ⟦Ide t ⟹ Coinitial (head-strategy t) t; Ide λ[t]⟧
                ⟹ Coinitial (head-strategy λ[t]) λ[t]
      by simp
    fix t1 t2
      assume ind1: Ide t1 ⟹ Coinitial (head-strategy t1) t1
    assume ind2: Ide t2 ⟹ Coinitial (head-strategy t2) t2
    assume t: Ide (t1 ⋅ t2)
    show Coinitial (head-strategy (t1 ⋅ t2)) (t1 ⋅ t2)
      using t ind1 Ide-implies-Arr Ide-iff-Src-self
      by (cases t1) simp-all
    next
    fix t1 t2
    assume ind1: Ide t1 ⟹ Coinitial (head-strategy t1) t1
    assume ind2: Ide t2 ⟹ Coinitial (head-strategy t2) t2
    assume t: Ide (λ[t1] • t2)
    show Coinitial (head-strategy (λ[t1] • t2)) (λ[t1] • t2)
      using t by auto
  qed
qed
```

The following function tests whether a term is an elementary reduction of the head redex.

```
fun is-head-reduction
where is-head-reduction «-» ⟷ False
    | is-head-reduction λ[t] ⟷ is-head-reduction t
    | is-head-reduction (λ[-] ⋅ -) ⟷ False
    | is-head-reduction (t ⋅ u) ⟷ is-head-reduction t ∧ Ide u
    | is-head-reduction (λ[t] • u) ⟷ Ide t ∧ Ide u
    | is-head-reduction ♯ ⟷ False

lemma is-head-reduction-char:
shows is-head-reduction t ⟷ elementary-reduction t ∧ head-strategy (Src t) = t
  apply (induct t)
     apply simp-all
proof −
  fix t1 t2
  assume ind: is-head-reduction t1 ⟷
              elementary-reduction t1 ∧ head-strategy (Src t1) = t1
  show is-head-reduction (t1 ⋅ t2) ⟷
        (elementary-reduction t1 ∧ Ide t2 ∨ Ide t1 ∧ elementary-reduction t2) ∧
         head-strategy (Src t1 ⋅ Src t2) = t1 ⋅ t2
    using ind Ide-implies-Arr Ide-iff-Src-self Ide-Src elementary-reduction-not-ide
          ide-char
    apply (cases t1)
```

```
      apply simp-all
     apply (metis Ide-Src arr-char elementary-reduction-is-arr)
    apply (metis Ide-Src arr-char elementary-reduction-is-arr)
   by metis
 next
 fix t1 t2
 show Ide t1 ∧ Ide t2 ⟷ Ide t1 ∧ Ide t2 ∧ Src (Src t1) = t1 ∧ Src (Src t2) = t2
   by (metis Ide-iff-Src-self Ide-implies-Arr)
qed

lemma is-head-reductionI:
assumes Arr t and elementary-reduction t and head-strategy (Src t) = t
shows is-head-reduction t
 using assms is-head-reduction-char by blast
```

The following function tests whether a redex in the head position of a term is marked.

```
fun contains-head-reduction
where contains-head-reduction «-» ⟷ False
   | contains-head-reduction λ[t] ⟷ contains-head-reduction t
   | contains-head-reduction (λ[-] ∘ -) ⟷ False
   | contains-head-reduction (t ∘ u) ⟷ contains-head-reduction t ∧ Arr u
   | contains-head-reduction (λ[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
 using Ide-implies-Arr
 apply (induct t)
    apply auto
proof −
 fix t1 t2
 assume ind1: is-head-reduction t1 ⟹ contains-head-reduction t1
 assume ind2: is-head-reduction t2 ⟹ contains-head-reduction t2
 assume t: is-head-reduction (t1 ∘ t2)
 show contains-head-reduction (t1 ∘ t2)
   using t ind1 ind2 Ide-implies-Arr
   by (cases t1) auto
qed
```

An *internal reduction* is one that does not contract any redex at the head position.

```
fun is-internal-reduction
where is-internal-reduction «-» ⟷ True
   | is-internal-reduction λ[t] ⟷ is-internal-reduction t
   | is-internal-reduction (λ[t] ∘ u) ⟷ Arr t ∧ Arr u
   | is-internal-reduction (t ∘ u) ⟷ is-internal-reduction t ∧ Arr u
   | is-internal-reduction (λ[-] • -) ⟷ False
   | is-internal-reduction ♯ ⟷ False

lemma is-internal-reduction-iff:
```

shows *is-internal-reduction t* $\longleftrightarrow$ *Arr t* $\wedge$ $\neg$ *contains-head-reduction t*
apply (*induct t*)
apply *simp-all*
proof −
fix *t1 t2*
assume *ind1*: *is-internal-reduction t1* $\longleftrightarrow$ *Arr t1* $\wedge$ $\neg$ *contains-head-reduction t1*
assume *ind2*: *is-internal-reduction t2* $\longleftrightarrow$ *Arr t2* $\wedge$ $\neg$ *contains-head-reduction t2*
show *is-internal-reduction* (*t1* ⋅ *t2*) $\longleftrightarrow$
Arr t1 $\wedge$ *Arr t2* $\wedge$ $\neg$ *contains-head-reduction* (*t1* ⋅ *t2*)
using *ind1 ind2*
apply (*cases t1*)
apply *simp-all*
by *blast*
qed

Head reduction steps are either $\lesssim$-prefixes of, or are preserved by, residuation along arbitrary reductions.

lemma *is-head-reduction-resid*:
shows ⟦*is-head-reduction t*; *Arr u*; *Src t* = *Src u*⟧ $\Longrightarrow$ *t* $\lesssim$ *u* $\vee$ *is-head-reduction* (*t* \ *u*)
proof (*induct t arbitrary*: *u*)
show $\bigwedge$*u*. ⟦*is-head-reduction* ♯; *Arr u*; *Src* ♯ = *Src u*⟧
$\Longrightarrow$ ♯ $\lesssim$ *u* $\vee$ *is-head-reduction* (♯ \ *u*)
by *auto*
show $\bigwedge$*x u*. ⟦*is-head-reduction «x»*; *Arr u*; *Src «x»* = *Src u*⟧
$\Longrightarrow$ *«x»* $\lesssim$ *u* $\vee$ *is-head-reduction* (*«x»* \ *u*)
by *auto*
fix *t u*
assume *ind*: $\bigwedge$*u*. ⟦*is-head-reduction t*; *Arr u*; *Src t* = *Src u*⟧
$\Longrightarrow$ *t* $\lesssim$ *u* $\vee$ *is-head-reduction* (*t* \ *u*)
assume *t*: *is-head-reduction* **λ**[*t*]
assume *u*: *Arr u*
assume *tu*: *Src* **λ**[*t*] = *Src u*
have *1*: *Arr t*
by (*metis Arr-head-strategy head-strategy-Src is-head-reduction-char Arr.simps*(*3*) *t tu u*)
show **λ**[*t*] $\lesssim$ *u* $\vee$ *is-head-reduction* (**λ**[*t*] \ *u*)
using *t u tu 1 ind*
by (*cases u*) *auto*
next
fix *t1 t2 u*
assume *ind1*: $\bigwedge$*u1*. ⟦*is-head-reduction t1*; *Arr u1*; *Src t1* = *Src u1*⟧
$\Longrightarrow$ *t1* $\lesssim$ *u1* $\vee$ *is-head-reduction* (*t1* \ *u1*)
assume *ind2*: $\bigwedge$*u2*. ⟦*is-head-reduction t2*; *Arr u2*; *Src t2* = *Src u2*⟧
$\Longrightarrow$ *t2* $\lesssim$ *u2* $\vee$ *is-head-reduction* (*t2* \ *u2*)
assume *t*: *is-head-reduction* (**λ**[*t1*] • *t2*)
assume *u*: *Arr u*
assume *tu*: *Src* (**λ**[*t1*] • *t2*) = *Src u*
show **λ**[*t1*] • *t2* $\lesssim$ *u* $\vee$ *is-head-reduction* ((**λ**[*t1*] • *t2*) \ *u*)
using *t u tu ind1 ind2 Coinitial-iff-Con Ide-implies-Arr ide-char resid-Ide-Arr Ide-Subst*
by (*cases u*; *cases un-App1 u*) *auto*

```
next
fix t1 t2 u
assume ind1: ⋀u1. ⟦is-head-reduction t1; Arr u1; Src t1 = Src u1⟧
                  ⟹ t1 ≲ u1 ∨ is-head-reduction (t1 \ u1)
assume ind2: ⋀u2. ⟦is-head-reduction t2; Arr u2; Src t2 = Src u2⟧
                  ⟹ t2 ≲ u2 ∨ is-head-reduction (t2 \ u2)
assume t: is-head-reduction (t1 ⋅ t2)
assume u: Arr u
assume tu: Src (t1 ⋅ t2) = Src u
have Arr (t1 ⋅ t2)
  using is-head-reduction-char elementary-reduction-is-arr t by blast
hence t1: Arr t1 and t2: Arr t2
  by auto
have 0: ¬ is-Lam t1
  using t is-Lam-def by fastforce
have 1: is-head-reduction t1
  using t t1 by force
show t1 ⋅ t2 ≲ u ∨ is-head-reduction ((t1 ⋅ t2) \ u)
proof −
  have ¬ Ide ((t1 ⋅ t2) \ u) ⟹ is-head-reduction ((t1 ⋅ t2) \ u)
  proof (intro is-head-reductionI)
    assume 2: ¬ Ide ((t1 ⋅ t2) \ u)
    have 3: is-App u ⟹ ¬ Ide (t1 \ un-App1 u) ∨ ¬ Ide (t2 \ un-App2 u)
    by (metis 2 ide-char lambda.collapse(3) lambda.discI(3) lambda.sel(3−4) prfx-App-iff)
    have 4: is-Beta u ⟹ ¬ Ide (t1 \ un-Beta1 u) ∨ ¬ Ide (t2 \ un-Beta2 u)
      using u tu 2
      by (metis 0 ConI Con-implies-is-Lam-iff-is-Lam ‹Arr (t1 ⋅ t2)›
          ConD(4) lambda.collapse(4) lambda.disc(8))
    show 5: Arr ((t1 ⋅ t2) \ u)
      using Arr-resid ‹Arr (t1 ⋅ t2)› tu u by auto
    show head-strategy (Src ((t1 ⋅ t2) \ u)) = (t1 ⋅ t2) \ u
    proof (cases u)
      show u = ♯ ⟹ head-strategy (Src ((t1 ⋅ t2) \ u)) = (t1 ⋅ t2) \ u
        by simp
      show ⋀x. u = «x» ⟹ head-strategy (Src ((t1 ⋅ t2) \ u)) = (t1 ⋅ t2) \ u
        by auto
      show ⋀v. u = λ[v] ⟹ head-strategy (Src ((t1 ⋅ t2) \ u)) = (t1 ⋅ t2) \ u
        by simp
      show ⋀u1 u2. u = λ[u1] • u2 ⟹ head-strategy (Src ((t1 ⋅ t2) \ u)) = (t1 ⋅ t2) \ u
        by (metis 0 5 Arr-not-Nil ConD(4) Con-implies-is-Lam-iff-is-Lam lambda.disc(8))
      show ⋀u1 u2. u = App u1 u2 ⟹ head-strategy (Src ((t1 ⋅ t2) \ u)) = (t1 ⋅ t2) \ u
      proof −
        fix u1 u2
        assume u1u2: u = u1 ⋅ u2
        have head-strategy (Src ((t1 ⋅ t2) \ u)) =
              head-strategy (Src (t1 \ u1) ⋅ Src (t2 \ u2))
          using u u1u2 tu t1 t2 Coinitial-iff-Con by auto
        also have ... = head-strategy (Trg u1 ⋅ Trg u2)
          using 5 u1u2 Src-resid
```

```
        by (metis Arr-not-Nil ConD(1))
      also have ... = (t1 ⋆ t2) \ u
      proof (cases Trg u1)
        show Trg u1 = ♯ ⟹ head-strategy (Trg u1 ⋆ Trg u2) = (t1 ⋆ t2) \ u
          using Arr-not-Nil u u1u2 by force
        show ⋀x. Trg u1 = «x» ⟹ head-strategy (Trg u1 ⋆ Trg u2) = (t1 ⋆ t2) \ u
          using tu t u t1 t2 u1u2 Arr-not-Nil Ide-iff-Src-self
          by (cases u1; cases t1) auto
        show ⋀v. Trg u1 = λ[v] ⟹ head-strategy (Trg u1 ⋆ Trg u2) = (t1 ⋆ t2) \ u
          using tu t u t1 t2 u1u2 Arr-not-Nil Ide-iff-Src-self
          apply (cases u1; cases t1)
                        apply auto
          by (metis 2 5 Src-resid Trg.simps(3−4) resid.simps(3−4) resid-Src-Arr)
        show ⋀u11 u12. Trg u1 = u11 ⋆ u12
                       ⟹ head-strategy (Trg u1 ⋆ Trg u2) = (t1 ⋆ t2) \ u
        proof −
          fix u11 u12
          assume u1: Trg u1 = u11 ⋆ u12
          show head-strategy (Trg u1 ⋆ Trg u2) = (t1 ⋆ t2) \ u
          proof (cases Trg u1)
            show Trg u1 = ♯ ⟹ ?thesis
              using u1 by simp
            show ⋀x. Trg u1 = «x» ⟹ ?thesis
              apply simp
              using u1 by force
            show ⋀v. Trg u1 = λ[v] ⟹ ?thesis
              using u1 by simp
            show ⋀u11 u12. Trg u1 = u11 ⋆ u12 ⟹ ?thesis
              using t u tu u1u2 1 2 ind1 elementary-reduction-not-ide
                    is-head-reduction-char Src-resid Ide-iff-Src-self
                    ‹Arr (t1 ⋆ t2)› Coinitial-iff-Con
              by fastforce
            show ⋀u11 u12. Trg u1 = λ[u11] • u12 ⟹ ?thesis
              using u1 by simp
          qed
        qed
        show ⋀u11 u12. Trg u1 = λ[u11] • u12 ⟹ ?thesis
          using u1u2 u Ide-Trg by fastforce
      qed
      finally show head-strategy (Src ((t1 ⋆ t2) \ u)) = (t1 ⋆ t2) \ u
        by simp
    qed
  qed
  thus elementary-reduction ((t1 ⋆ t2) \ u)
    by (metis 2 5 Ide-Src Ide-implies-Arr head-strategy-is-elementary)
 qed
 thus ?thesis by blast
qed
qed
```

Internal reductions are closed under residuation.

```
lemma is-internal-reduction-resid:
shows ⟦is-internal-reduction t; is-internal-reduction u; Src t = Src u⟧
        ⟹ is-internal-reduction (t \ u)
  apply (induct t arbitrary: u)
     apply auto
  apply (metis Con-implies-Arr2 con-char weak-extensionality Arr.simps(2) Src.simps(2)
               parallel-strategy.simps(1) prfx-implies-con resid-Arr-Src subs-Ide
               subs-implies-prfx subs-parallel-strategy-Src)
proof −
  fix t u
  assume ind: ⋀u. ⟦is-internal-reduction u; Src t = Src u⟧ ⟹ is-internal-reduction (t \ u)
  assume t: is-internal-reduction t
  assume u: is-internal-reduction u
  assume tu: λ[Src t] = Src u
  show is-internal-reduction (λ[t] \ u)
    using t u tu ind
    apply (cases u)
    by auto fastforce
  next
  fix t1 t2 u
  assume ind1: ⋀u. ⟦is-internal-reduction t1; is-internal-reduction u; Src t1 = Src u⟧
                     ⟹ is-internal-reduction (t1 \ u)
  assume t: is-internal-reduction (t1 ⋅ t2)
  assume u: is-internal-reduction u
  assume tu: Src t1 ⋅ Src t2 = Src u
  show is-internal-reduction ((t1 ⋅ t2) \ u)
    using t u tu ind1 Coinitial-resid-resid Coinitial-iff-Con Arr-Src
          is-internal-reduction-iff
    apply auto
     apply (metis Arr.simps(4) Src.simps(4))
  proof −
    assume t1: Arr t1 and t2: Arr t2 and u: Arr u
    assume tu: Src t1 ⋅ Src t2 = Src u
    assume 1: ¬ contains-head-reduction u
    assume 2: ¬ contains-head-reduction (t1 ⋅ t2)
    assume 3: contains-head-reduction ((t1 ⋅ t2) \ u)
    show False
      using t1 t2 u tu 1 2 3 is-internal-reduction-iff
      apply (cases u)
         apply simp-all
      apply (cases t1; cases un-App1 u)
                       apply simp-all
      by (metis Coinitial-iff-Con ind1 Arr.simps(4) Src.simps(4) resid.simps(3))
  qed
qed
```

A head reduction is preserved by residuation along an internal reduction, so a head reduction can only be canceled by a transition that contains a head reduction.

lemma *is-head-reduction-resid'*:
shows ⟦*is-head-reduction t*; *is-internal-reduction u*; *Src t = Src u*⟧
  ⟹ *is-head-reduction* (*t* \ *u*)
proof (*induct t arbitrary*: *u*)
  show ⋀*u*. ⟦*is-head-reduction* ♯; *is-internal-reduction u*; *Src* ♯ = *Src u*⟧
    ⟹ *is-head-reduction* (♯ \ *u*)
    by *simp*
  show ⋀*x u*. ⟦*is-head-reduction «x»*; *is-internal-reduction u*; *Src «x» = Src u*⟧
    ⟹ *is-head-reduction* (*«x»* \ *u*)
    by *simp*
  show ⋀*t*. ⟦⋀*u*. ⟦*is-head-reduction t*; *is-internal-reduction u*; *Src t = Src u*⟧
      ⟹ *is-head-reduction* (*t* \ *u*);
    is-head-reduction **λ**[*t*]; *is-internal-reduction u*; *Src* **λ**[*t*] = *Src u*⟧
    ⟹ *is-head-reduction* (**λ**[*t*] \ *u*)
    for *u*
    by (*cases u*, *simp-all*) *fastforce*
  fix *t1 t2 u*
  assume *ind1*: ⋀*u*. ⟦*is-head-reduction t1*; *is-internal-reduction u*; *Src t1 = Src u*⟧
      ⟹ *is-head-reduction* (*t1* \ *u*)
  assume *t*: *is-head-reduction* (*t1* ∘ *t2*)
  assume *u*: *is-internal-reduction u*
  assume *tu*: *Src* (*t1* ∘ *t2*) = *Src u*
  show *is-head-reduction* ((*t1* ∘ *t2*) \ *u*)
    using *t u tu ind1*
    apply (*cases u*)
     apply *simp-all*
  proof (*intro conjI impI*)
    fix *u1 u2*
    assume *u1u2*: *u = u1* ∘ *u2*
    show *1*: *Con t1 u1*
      using *Coinitial-iff-Con tu u1u2 ide-char*
      by (*metis ConD*(*1*) *Ide.simps*(*1*) *is-head-reduction.simps*(*9*) *is-head-reduction-resid*
          is-internal-reduction.simps(*9*) *is-internal-reduction-resid t u*)
    show *Con t2 u2*
      using *Coinitial-iff-Con tu u1u2 ide-char*
      by (*metis ConD*(*1*) *Ide.simps*(*1*) *is-head-reduction.simps*(*9*) *is-head-reduction-resid*
          is-internal-reduction.simps(*9*) *is-internal-reduction-resid t u*)
    show *is-head-reduction* (*t1* \ *u1* ∘ *t2* \ *u2*)
      using *t u u1u2 1 Coinitial-iff-Con* ‹*Con t2 u2*› *ide-char ind1 resid-Ide-Arr*
      apply (*cases t1*; *simp-all*; *cases u1*; *simp-all*; *cases un-App1 u1*)
           apply *auto*
      by (*metis 1 ind1 is-internal-reduction.simps*(*6*) *resid.simps*(*3*))
  qed
  next
  fix *t1 t2 u*
  assume *ind1*: ⋀*u*. ⟦*is-head-reduction t1*; *is-internal-reduction u*; *Src t1 = Src u*⟧
      ⟹ *is-head-reduction* (*t1* \ *u*)
  assume *t*: *is-head-reduction* (**λ**[*t1*] • *t2*)
  assume *u*: *is-internal-reduction u*

```
  assume tu: Src (λ[t1] • t2) = Src u
  show is-head-reduction ((λ[t1] • t2) \ u)
    using t u tu ind1
    apply (cases u)
      apply simp-all
    by (metis Con-implies-Arr1 is-head-reduction-resid is-internal-reduction.simps(9)
        is-internal-reduction-resid lambda.disc(15) prfx-App-iff t tu)
qed
```

The following function differs from *head-strategy* in that it only selects an already-marked redex, whereas *head-strategy* marks the redex at the head position.

```
fun head-redex
where head-redex ♯ = ♯
    | head-redex «x» = «x»
    | head-redex λ[t] = λ[head-redex t]
    | head-redex (λ[t] ∘ u) = λ[Src t] ∘ Src u
    | head-redex (t ∘ u) = head-redex t ∘ Src u
    | head-redex (λ[t] • u) = (λ[Src t] • Src u)

lemma elementary-reduction-head-redex:
shows ⟦Arr t; ¬ Ide (head-redex t)⟧ ⟹ elementary-reduction (head-redex t)
  using Ide-Src
  apply (induct t)
    apply auto
proof −
  show ⋀t2. ⟦¬ Ide (head-redex t1) ⟹ elementary-reduction (head-redex t1);
             ¬ Ide (head-redex (t1 ∘ t2));
             ⋀t. Arr t ⟹ Ide (Src t); Arr t1; Arr t2⟧
               ⟹ elementary-reduction (head-redex (t1 ∘ t2))
    for t1
    using Ide-Src
    by (cases t1) auto
qed

lemma subs-head-redex:
shows Arr t ⟹ head-redex t ⊑ t
  using Ide-Src subs-Ide
  apply (induct t)
    apply simp-all
proof −
  show ⋀t2. ⟦head-redex t1 ⊑ t1; head-redex t2 ⊑ t2;
             Arr t1 ∧ Arr t2; ⋀t. Arr t ⟹ Ide (Src t);
             ⋀u t. ⟦Ide u; Src t = Src u⟧ ⟹ u ⊑ t⟧
               ⟹ head-redex (t1 ∘ t2) ⊑ t1 ∘ t2
    for t1
    using Ide-Src subs-Ide
    by (cases t1) auto
qed
```

lemma *contains-head-reduction-iff*:
shows *contains-head-reduction t* $\longleftrightarrow$ *Arr t* $\wedge$ $\neg$ *Ide* (*head-redex t*)
apply (*induct t*)
apply *simp-all*
proof −
show $\bigwedge$*t2*. *contains-head-reduction t1* = (*Arr t1* $\wedge$ $\neg$ *Ide* (*head-redex t1*))
$\Longrightarrow$ *contains-head-reduction* (*t1* ∘ *t2*) =
(*Arr t1* $\wedge$ *Arr t2* $\wedge$ $\neg$ *Ide* (*head-redex* (*t1* ∘ *t2*)))
for *t1*
using *Ide-Src*
by (*cases t1*) *auto*
qed

lemma *head-redex-is-head-reduction*:
shows ⟦*Arr t*; *contains-head-reduction t*⟧ $\Longrightarrow$ *is-head-reduction* (*head-redex t*)
using *Ide-Src*
apply (*induct t*)
apply *simp-all*
proof −
show $\bigwedge$*t2*. ⟦*contains-head-reduction t1* $\Longrightarrow$ *is-head-reduction* (*head-redex t1*);
Arr t1 $\wedge$ *Arr t2*;
contains-head-reduction (*t1* ∘ *t2*); $\bigwedge$*t*. *Arr t* $\Longrightarrow$ *Ide* (*Src t*)⟧
$\Longrightarrow$ *is-head-reduction* (*head-redex* (*t1* ∘ *t2*))
for *t1*
using *Ide-Src contains-head-reduction-iff subs-implies-prfx*
by (*cases t1*) *auto*
qed

lemma *Arr-head-redex*:
assumes *Arr t*
shows *Arr* (*head-redex t*)
using *assms Ide-implies-Arr elementary-reduction-head-redex elementary-reduction-is-arr*
by *blast*

lemma *Src-head-redex*:
assumes *Arr t*
shows *Src* (*head-redex t*) = *Src t*
using *assms*
by (*metis Coinitial-iff-Con Ide.simps*(*1*) *ide-char subs-head-redex subs-implies-prfx*)

lemma *Con-Arr-head-redex*:
assumes *Arr t*
shows *Con t* (*head-redex t*)
using *assms*
by (*metis Con-sym Ide.simps*(*1*) *ide-char subs-head-redex subs-implies-prfx*)

lemma *is-head-reduction-if*:
shows ⟦*contains-head-reduction u*; *elementary-reduction u*⟧ $\Longrightarrow$ *is-head-reduction u*
apply (*induct u*)

```
    apply auto
  using contains-head-reduction.elims(2)
   apply fastforce
proof −
  fix u1 u2
  assume u1: Ide u1
  assume u2: elementary-reduction u2
  assume 1: contains-head-reduction (u1 ⋅ u2)
  have False
    using u1 u2 1
    apply (cases u1)
       apply auto
    by (metis Arr-head-redex Ide-iff-Src-self Src-head-redex contains-head-reduction-iff
        ide-char resid-Arr-Src subs-head-redex subs-implies-prfx u1)
  thus is-head-reduction (u1 ⋅ u2)
    by blast
qed

lemma (in reduction-paths) head-redex-decomp:
assumes Λ.Arr t
shows [Λ.head-redex t] @ [t \ Λ.head-redex t] *∼* [t]
  using assms prfx-decomp Λ.subs-head-redex Λ.subs-implies-prfx
  by (metis Ide.simps(2) Resid.simps(3) Λ.prfx-implies-con ide-char)
```

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))⟧
        ⟹ Ide (head-strategy (Trg u))
  apply (induct u)
     apply simp-all
proof −
  fix u1 u2
  assume ind1: ⟦is-internal-reduction u1; Ide (head-strategy (Src u1))⟧
                ⟹ Ide (head-strategy (Trg u1))
  assume ind2: ⟦is-internal-reduction u2; Ide (head-strategy (Src u2))⟧
                ⟹ Ide (head-strategy (Trg u2))
  assume u: is-internal-reduction (u1 ⋅ u2)
  assume 1: Ide (head-strategy (Src u1 ⋅ Src u2))
  show Ide (head-strategy (Trg u1 ⋅ Trg u2))
    using u 1 ind1 ind2 Ide-Src Ide-Trg Ide-implies-Arr
    by (cases u1) auto
qed

lemma head-reduction-unique:
shows ⟦is-head-reduction t; is-head-reduction u; coinitial t u⟧ ⟹ t = u
  by (metis Coinitial-iff-Con con-def confluence is-head-reduction-char null-char)
```

Residuation along internal reductions preserves head reductions.

```
lemma resid-head-strategy-internal:
```

shows *is-internal-reduction u* $\Longrightarrow$ *head-strategy* (*Src u*) \ *u* = *head-strategy* (*Trg u*)
using *internal-reduction-preserves-no-head-redex Arr-head-strategy Ide-iff-Src-self Src-head-strategy Src-resid head-strategy-is-elementary is-head-reduction-char is-head-reduction-resid′ is-internal-reduction-iff*
apply (*cases u*)
apply *simp-all*
apply (*metis head-strategy-Src resid-Src-Arr*)
apply (*metis head-strategy-Src Arr.simps*(4) *Src.simps*(4) *Trg.simps*(3) *resid-Src-Arr*)
by *blast*

An internal reduction followed by a head reduction can be expressed as a join of the internal reduction with a head reduction.

lemma *resid-head-strategy-Src*:
assumes *is-internal-reduction t* **and** *is-head-reduction u*
and *seq t u*
shows *head-strategy* (*Src t*) \ *t* = *u*
and *composite-of t u* (*Join* (*head-strategy* (*Src t*)) *t*)
proof −
show *1*: *head-strategy* (*Src t*) \ *t* = *u*
using *assms internal-reduction-preserves-no-head-redex resid-head-strategy-internal elementary-reduction-not-ide ide-char is-head-reduction-char seq-char*
by *force*
show *composite-of t u* (*Join* (*head-strategy* (*Src t*)) *t*)
using *assms*(3) *1 Arr-head-strategy Src-head-strategy join-of-Join join-of-def seq-char*
by *force*
qed

lemma *App-Var-contains-no-head-reduction*:
shows ¬ *contains-head-reduction* (*«x»* ∘ *u*)
by *simp*

lemma *hgt-resid-App-head-redex*:
assumes *Arr* (*t* ∘ *u*) **and** ¬ *Ide* (*head-redex* (*t* ∘ *u*))
shows *hgt* ((*t* ∘ *u*) \ *head-redex* (*t* ∘ *u*)) < *hgt* (*t* ∘ *u*)
using *assms contains-head-reduction-iff elementary-reduction-decreases-hgt elementary-reduction-head-redex subs-head-redex*
by *blast*

3.5.3 Leftmost Reduction

Leftmost (or normal-order) reduction is the strategy that produces an elementary reduction path by contracting the leftmost redex at each step. It agrees with head reduction as long as there is a head redex, otherwise it continues on with the next subterm to the right.

fun *leftmost-strategy*
where *leftmost-strategy «x»* = *«x»*
| *leftmost-strategy* **λ**[*t*] = **λ**[*leftmost-strategy t*]
| *leftmost-strategy* (**λ**[*t*] ∘ *u*) = **λ**[*t*] • *u*

```
  | leftmost-strategy (t ∘ u) =
      (if ¬ Ide (leftmost-strategy t)
       then leftmost-strategy t ∘ u
       else t ∘ leftmost-strategy u)
  | leftmost-strategy (λ[t] • u) = λ[t] • u
  | leftmost-strategy ♯ = ♯


definition is-leftmost-reduction
where is-leftmost-reduction t ⟷ elementary-reduction t ∧ leftmost-strategy (Src t) = t

lemma leftmost-strategy-is-reduction-strategy:
shows reduction-strategy leftmost-strategy
proof (unfold reduction-strategy-def, intro allI impI)
  fix t
  show Ide t ⟹ Coinitial (leftmost-strategy t) t
  proof (induct t, auto)
    show ⋀t2. ⟦Arr (leftmost-strategy t1); Arr (leftmost-strategy t2);
               Ide t1; Ide t2;
               Arr t1; Src (leftmost-strategy t1) = Src t1;
               Arr t2; Src (leftmost-strategy t2) = Src t2⟧
                 ⟹ Arr (leftmost-strategy (t1 ∘ t2))
         for t1
      by (cases t1) auto
  qed
qed

lemma elementary-reduction-leftmost-strategy:
shows Ide t ⟹ elementary-reduction (leftmost-strategy t) ∨ Ide (leftmost-strategy t)
  apply (induct t)
     apply simp-all
proof −
  fix t1 t2
  show ⟦elementary-reduction (leftmost-strategy t1) ∨ Ide (leftmost-strategy t1);
        elementary-reduction (leftmost-strategy t2) ∨ Ide (leftmost-strategy t2);
        Ide t1 ∧ Ide t2⟧
          ⟹ elementary-reduction (leftmost-strategy (t1 ∘ t2)) ∨
             Ide (leftmost-strategy (t1 ∘ t2))
    by (cases t1) auto
qed

lemma (in lambda-calculus) leftmost-strategy-selects-head-reduction:
shows is-head-reduction t ⟹ t = leftmost-strategy (Src t)
proof (induct t)
  show ⋀t1 t2. ⟦is-head-reduction t1 ⟹ t1 = leftmost-strategy (Src t1);
                is-head-reduction (t1 ∘ t2)⟧
                  ⟹ t1 ∘ t2 = leftmost-strategy (Src (t1 ∘ t2))
  proof −
    fix t1 t2
```

```
    assume ind1: is-head-reduction t1 ⟹ t1 = leftmost-strategy (Src t1)
    assume t: is-head-reduction (t1 ∘ t2)
    show t1 ∘ t2 = leftmost-strategy (Src (t1 ∘ t2))
      using t ind1
      apply (cases t1)
         apply simp-all
       apply (cases Src t1)
          apply simp-all
      using ind1
           apply force
      using ind1
          apply force
      using ind1
        apply force
       apply (metis Ide-iff-Src-self Ide-implies-Arr elementary-reduction-not-ide
         ide-char ind1 is-head-reduction-char)
      using ind1
       apply force
      by (metis Ide-iff-Src-self Ide-implies-Arr)
  qed
  show ⋀t1 t2. ⟦is-head-reduction t1 ⟹ t1 = leftmost-strategy (Src t1);
                is-head-reduction (λ[t1] • t2)⟧
                  ⟹ λ[t1] • t2 = leftmost-strategy (Src (λ[t1] • t2))
    by (metis Ide-iff-Src-self Ide-implies-Arr Src.simps(5)
        is-head-reduction.simps(8) leftmost-strategy.simps(3))
qed auto

lemma has-redex-iff-not-Ide-leftmost-strategy:
shows Arr t ⟹ has-redex t ⟷ ¬ Ide (leftmost-strategy (Src t))
  apply (induct t)
     apply simp-all
proof −
  fix t1 t2
  assume ind1: Ide (parallel-strategy t1) ⟷ Ide (leftmost-strategy (Src t1))
  assume ind2: Ide (parallel-strategy t2) ⟷ Ide (leftmost-strategy (Src t2))
  assume t: Arr t1 ∧ Arr t2
  show Ide (parallel-strategy (t1 ∘ t2)) ⟷
       Ide (leftmost-strategy (Src t1 ∘ Src t2))
    using t ind1 ind2 Ide-Src Ide-iff-Src-self
    by (cases t1) auto
qed

lemma leftmost-reduction-preservation:
shows ⟦is-leftmost-reduction t; elementary-reduction u; ¬ is-leftmost-reduction u;
       coinitial t u⟧ ⟹ is-leftmost-reduction (t \ u)
proof (induct t arbitrary: u)
  show ⋀u. coinitial ♯ u ⟹ is-leftmost-reduction (♯ \ u)
    by simp
  show ⋀x u. is-leftmost-reduction «x» ⟹ is-leftmost-reduction («x» \ u)
```

```
  by (simp add: is-leftmost-reduction-def)
fix t u
show ⟦⋀u. ⟦is-leftmost-reduction t; elementary-reduction u;
           ¬ is-leftmost-reduction u; coinitial t u⟧ ⟹ is-leftmost-reduction (t \ u);
      is-leftmost-reduction (Lam t); elementary-reduction u;
      ¬ is-leftmost-reduction u; coinitial λ[t] u⟧
        ⟹ is-leftmost-reduction (λ[t] \ u)
  using is-leftmost-reduction-def
  by (cases u) auto
next
fix t1 t2 u
show ⟦is-leftmost-reduction (λ[t1] • t2); elementary-reduction u; ¬ is-leftmost-reduction u;
      coinitial (λ[t1] • t2) u⟧
        ⟹ is-leftmost-reduction ((λ[t1] • t2) \ u)
  using is-leftmost-reduction-def Src-resid Ide-Trg Ide-iff-Src-self Arr-Trg Arr-not-Nil
  apply (cases u)
     apply simp-all
  by (cases un-App1 u) auto
assume ind1: ⋀u. ⟦is-leftmost-reduction t1; elementary-reduction u;
                  ¬ is-leftmost-reduction u; coinitial t1 u⟧
                    ⟹ is-leftmost-reduction (t1 \ u)
assume ind2: ⋀u. ⟦is-leftmost-reduction t2; elementary-reduction u;
                  ¬ is-leftmost-reduction u; coinitial t2 u⟧
                    ⟹ is-leftmost-reduction (t2 \ u)
assume 1: is-leftmost-reduction (t1 ∘ t2)
assume 2: elementary-reduction u
assume 3: ¬ is-leftmost-reduction u
assume 4: coinitial (t1 ∘ t2) u
show is-leftmost-reduction ((t1 ∘ t2) \ u)
  using 1 2 3 4 ind1 ind2 is-leftmost-reduction-def Src-resid
  apply (cases u)
     apply auto[3]
proof −
  show ⋀u1 u2. u = λ[u1] • u2 ⟹ is-leftmost-reduction ((t1 ∘ t2) \ u)
    by (metis 2 3 is-leftmost-reduction-def elementary-reduction.simps(5)
        is-head-reduction.simps(8) leftmost-strategy-selects-head-reduction)
  fix u1 u2
  assume u: u = u1 ∘ u2
  show is-leftmost-reduction ((t1 ∘ t2) \ u)
    using u 1 2 3 4 ind1 ind2 is-leftmost-reduction-def Src-resid Ide-Trg
          elementary-reduction-not-ide
    apply (cases u)
       apply simp-all
    apply (cases u1)
       apply simp-all
      apply auto[1]
    using Ide-iff-Src-self
     apply simp-all
  proof −
```

fix *u11 u12*
assume *u*: $u = u11 \circ u12 \circ u2$
assume *u1*: $u1 = u11 \circ u12$
have *A*: $(\mathit{elementary\text{-}reduction}\ t1 \wedge \mathit{Src}\ u2 = t2 \vee$
$\mathit{Src}\ u11 \circ \mathit{Src}\ u12 = t1 \wedge \mathit{elementary\text{-}reduction}\ t2) \wedge$
$(\mathit{if} \neg \mathit{Ide}\ (\mathit{leftmost\text{-}strategy}\ (\mathit{Src}\ u11 \circ \mathit{Src}\ u12))$
$\mathit{then}\ \mathit{leftmost\text{-}strategy}\ (\mathit{Src}\ u11 \circ \mathit{Src}\ u12) \circ \mathit{Src}\ u2$
$\mathit{else}\ \mathit{Src}\ u11 \circ \mathit{Src}\ u12 \circ \mathit{leftmost\text{-}strategy}\ (\mathit{Src}\ u2)) = t1 \circ t2$
using *1 4 Ide-iff-Src-self is-leftmost-reduction-def u* **by** *auto*
have *B*: $(\mathit{elementary\text{-}reduction}\ u11 \wedge \mathit{Src}\ u12 = u12 \vee$
$\mathit{Src}\ u11 = u11 \wedge \mathit{elementary\text{-}reduction}\ u12) \wedge \mathit{Src}\ u2 = u2 \vee$
$\mathit{Src}\ u11 = u11 \wedge \mathit{Src}\ u12 = u12 \wedge \mathit{elementary\text{-}reduction}\ u2$
using *2 4 Ide-iff-Src-self u* **by** *force*
have *C*: $t1 = u11 \circ u12 \longrightarrow t2 \neq u2$
using *1 3 u* **by** *fastforce*
have *D*: $\mathit{Arr}\ t1 \wedge \mathit{Arr}\ t2 \wedge \mathit{Arr}\ u11 \wedge \mathit{Arr}\ u12 \wedge \mathit{Arr}\ u2 \wedge$
$\mathit{Src}\ t1 = \mathit{Src}\ u11 \circ \mathit{Src}\ u12 \wedge \mathit{Src}\ t2 = \mathit{Src}\ u2$
using *4 u* **by** *force*
have *E*: $\bigwedge u.\ [\![\mathit{elementary\text{-}reduction}\ t1 \wedge \mathit{leftmost\text{-}strategy}\ (\mathit{Src}\ u) = t1;$
$\mathit{elementary\text{-}reduction}\ u;$
$t1 \neq u;$
$\mathit{Arr}\ u \wedge \mathit{Src}\ u11 \circ \mathit{Src}\ u12 = \mathit{Src}\ u]\!]$
$\Longrightarrow \mathit{elementary\text{-}reduction}\ (t1 \setminus u) \wedge$
$\mathit{leftmost\text{-}strategy}\ (\mathit{Trg}\ u) = t1 \setminus u$
using *D Src-resid ind1 is-leftmost-reduction-def* **by** *auto*
have *F*: $\bigwedge u.\ [\![\mathit{elementary\text{-}reduction}\ t2 \wedge \mathit{leftmost\text{-}strategy}\ (\mathit{Src}\ u) = t2;$
$\mathit{elementary\text{-}reduction}\ u;$
$t2 \neq u;$
$\mathit{Arr}\ u \wedge \mathit{Src}\ u2 = \mathit{Src}\ u]\!]$
$\Longrightarrow \mathit{elementary\text{-}reduction}\ (t2 \setminus u) \wedge$
$\mathit{leftmost\text{-}strategy}\ (\mathit{Trg}\ u) = t2 \setminus u$
using *D Src-resid ind2 is-leftmost-reduction-def* **by** *auto*
have *G*: $\bigwedge t.\ \mathit{elementary\text{-}reduction}\ t \Longrightarrow \neg\ \mathit{Ide}\ t$
using *elementary-reduction-not-ide ide-char* **by** *blast*
have *H*: $\mathit{elementary\text{-}reduction}\ (t1 \setminus (u11 \circ u12)) \wedge \mathit{Ide}\ (t2 \setminus u2) \vee$
$\mathit{Ide}\ (t1 \setminus (u11 \circ u12)) \wedge \mathit{elementary\text{-}reduction}\ (t2 \setminus u2)$
proof $(\mathit{cases}\ \mathit{Ide}\ (t2 \setminus u2))$
assume *1*: $\mathit{Ide}\ (t2 \setminus u2)$
hence $\mathit{elementary\text{-}reduction}\ (t1 \setminus (u11 \circ u12))$
by (*metis A B C D E F G Ide-Src Arr.simps*(*4*) *Src.simps*(*4*)
elementary-reduction.simps(*4*) *lambda.inject*(*3*) *resid-Arr-Src*)
thus *?thesis*
using *1* **by** *auto*
next
assume *1*: $\neg\ \mathit{Ide}\ (t2 \setminus u2)$
hence $\mathit{Ide}\ (t1 \setminus (u11 \circ u12)) \wedge \mathit{elementary\text{-}reduction}\ (t2 \setminus u2)$
apply (*intro conjI*)
apply (*metis 1 A D Ide-Src Arr.simps*(*4*) *Src.simps*(*4*) *resid-Ide-Arr*)
by (*metis A B C D F Ide-iff-Src-self lambda.inject*(*3*) *resid-Arr-Src resid-Ide-Arr*)

```
  thus ?thesis by simp
qed
show (¬ Ide (leftmost-strategy (Trg u11 ∘ Trg u12)) ⟶
       (elementary-reduction (t1 \ (u11 ∘ u12)) ∧ Ide (t2 \ u2) ∨
        Ide (t1 \ (u11 ∘ u12)) ∧ elementary-reduction (t2 \ u2)) ∧
       leftmost-strategy (Trg u11 ∘ Trg u12) = t1 \ (u11 ∘ u12) ∧ Trg u2 = t2 \ u2) ∧
      (Ide (leftmost-strategy (Trg u11 ∘ Trg u12)) ⟶
       (elementary-reduction (t1 \ (u11 ∘ u12)) ∧ Ide (t2 \ u2) ∨
        Ide (t1 \ (u11 ∘ u12)) ∧ elementary-reduction (t2 \ u2)) ∧
        Trg u11 ∘ Trg u12 = t1 \ (u11 ∘ u12) ∧ leftmost-strategy (Trg u2) = t2 \ u2)
proof (intro conjI impI)
  show H: elementary-reduction (t1 \ (u11 ∘ u12)) ∧ Ide (t2 \ u2) ∨
            Ide (t1 \ (u11 ∘ u12)) ∧ elementary-reduction (t2 \ u2)
    by fact
  show H: elementary-reduction (t1 \ (u11 ∘ u12)) ∧ Ide (t2 \ u2) ∨
            Ide (t1 \ (u11 ∘ u12)) ∧ elementary-reduction (t2 \ u2)
    by fact
  assume K: ¬ Ide (leftmost-strategy (Trg u11 ∘ Trg u12))
  show J: Trg u2 = t2 \ u2
    using A B D G K has-redex-iff-not-Ide-leftmost-strategy
          NF-def NF-iff-has-no-redex NF-App-iff resid-Arr-Src resid-Src-Arr
    by (metis lambda.inject(3))
  show leftmost-strategy (Trg u11 ∘ Trg u12) = t1 \ (u11 ∘ u12)
    using 2 A B C D E G H J u Ide-Trg Src-Src
      has-redex-iff-not-Ide-leftmost-strategy resid-Arr-Ide resid-Src-Arr
    by (metis Arr.simps(4) Ide.simps(4) Src.simps(4) Trg.simps(3)
      elementary-reduction.simps(4) lambda.inject(3))
  next
  assume K: Ide (leftmost-strategy (Trg u11 ∘ Trg u12))
  show I: Trg u11 ∘ Trg u12 = t1 \ (u11 ∘ u12)
    using 2 A D E K u Coinitial-resid-resid ConI resid-Arr-self resid-Ide-Arr
          resid-Arr-Ide Ide-iff-Src-self Src-resid
    apply (cases Ide (leftmost-strategy (Src u11 ∘ Src u12)))
     apply simp
    using lambda-calculus.Con-Arr-Src(2)
     apply force
    apply simp
    using u1 G H Coinitial-iff-Con
    apply (cases elementary-reduction u11;
           cases elementary-reduction u12)
      apply simp-all
      apply metis
     apply (metis Src.simps(4) Trg.simps(3) elementary-reduction.simps(1,4))
    apply (metis Src.simps(4) Trg.simps(3) elementary-reduction.simps(1,4))
    by (metis Trg-Src)
  show leftmost-strategy (Trg u2) = t2 \ u2
    using 2 A C D F G H I u Ide-Trg Ide-iff-Src-self NF-def NF-iff-has-no-redex
          has-redex-iff-not-Ide-leftmost-strategy resid-Ide-Arr
    by (metis Arr.simps(4) Src.simps(4) Trg.simps(3) elementary-reduction.simps(4)
```

```
            lambda.inject(3))
      qed
    qed
  qed
qed

end
```

3.6 Standard Reductions

In this section, we define the notion of a *standard reduction*, which is an elementary reduction path that performs reductions from left to right, possibly skipping some redexes that could be contracted. Once a redex has been skipped, neither that redex nor any redex to its left will subsequently be contracted. We then define and prove correct a function that transforms an arbitrary elementary reduction path into a congruent standard reduction path. Using this function, we prove the Standardization Theorem, which says that every elementary reduction path is congruent to a standard reduction path. We then show that a standard reduction path that reaches a normal form is in fact a leftmost reduction path. From this fact and the Standardization Theorem we prove the Leftmost Reduction Theorem: leftmost reduction is a normalizing strategy.

The Standardization Theorem was first proved by Curry and Feys [3], with subsequent proofs given by a number of authors. Formalized proofs have also been given; a recent one (using Agda) is presented in [2], with references to earlier work. The version of the theorem that we formalize here is a "strong" version, which asserts the existence of a standard reduction path congruent to a 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 λ[t] λ[t'] = sseq t t'
     | sseq (t ∘ u) (t' ∘ 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 (λ[t] ∘ u) (λ[t'] • u') = False
     | sseq ((λ[t] • u) ∘ v) w =
           (Ide t ∧ Ide u ∧ Ide v ∧ elementary-reduction w ∧ seq ((λ[t] • u) ∘ v) w)
     | sseq (λ[t] • u) v = (Ide t ∧ Ide u ∧ elementary-reduction v ∧ seq (λ[t] • u) v)
     | sseq - - = False

  lemma sseq-imp-seq:
  shows sseq t u ⟹ seq t u
  proof (induct t arbitrary: u)
    show ⋀u. sseq ♯ u ⟹ seq ♯ u
      using sseq.elims(1) by blast
    fix u
    show ⋀x. sseq «x» u ⟹ seq «x» u
      using sseq.elims(1) by blast
    show ⋀t. ⟦⋀u. sseq t u ⟹ seq t u; sseq λ[t] u⟧ ⟹ seq λ[t] u
      using seq-char by (cases u) auto
    show ⋀t1 t2. ⟦⋀u. sseq t1 u ⟹ seq t1 u; ⋀u. sseq t2 u ⟹ seq t2 u;
                  sseq (λ[t1] • t2) u⟧
                    ⟹ seq (λ[t1] • t2) u
      using seq-char Ide-implies-Arr
      by (cases u) auto
    fix t1 t2
    show ⟦⋀u. sseq t1 u ⟹ seq t1 u; ⋀u. sseq t2 u ⟹ seq t2 u; sseq (t1 ∘ t2) u⟧
            ⟹ seq (t1 ∘ t2) u
    proof −
      assume ind1: ⋀u. sseq t1 u ⟹ seq t1 u
      assume ind2: ⋀u. sseq t2 u ⟹ seq t2 u
      assume 1: sseq (t1 ∘ t2) u
      show ?thesis
        using 1 ind1 ind2 seq-char arr-char elementary-reduction-is-arr
              Ide-Src Ide-Trg Ide-implies-Arr Coinitial-iff-Con resid-Arr-self
```

```
      apply (cases u, simp-all)
        apply (cases t1, simp-all)
       apply (cases t1, simp-all)
      apply (cases Ide t1; cases Ide t2)
        apply simp-all
       apply (metis Ide-iff-Src-self Ide-iff-Trg-self)
      apply (metis Ide-iff-Src-self Ide-iff-Trg-self)
     apply (metis Ide-iff-Trg-self Src-Trg)
    by (cases t1) auto
  qed
qed

lemma sseq-imp-elementary-reduction1:
shows sseq t u ⟹ elementary-reduction t
proof (induct u arbitrary: t)
  show ⋀t. sseq t ♯ ⟹ elementary-reduction t
    by simp
  show ⋀x t. sseq t «x» ⟹ elementary-reduction t
    using elementary-reduction.simps(2) sseq.elims(1) by blast
  show ⋀u. ⟦⋀t. sseq t u ⟹ elementary-reduction t; sseq t λ[u]⟧
             ⟹ elementary-reduction t for t
    using seq-cases sseq-imp-seq
    apply (cases t, simp-all)
    by force
  show ⋀u1 u2. ⟦⋀t. sseq t u1 ⟹ elementary-reduction t;
                ⋀t. sseq t u2 ⟹ elementary-reduction t;
                sseq t (u1 ∘ u2)⟧
                  ⟹ elementary-reduction t for t
    using seq-cases sseq-imp-seq Ide-Src elementary-reduction-is-arr
    apply (cases t, simp-all)
    by blast
  show ⋀u1 u2.
   ⟦⋀t. sseq t u1 ⟹ elementary-reduction t; ⋀t. sseq t u2 ⟹ elementary-reduction t;
    sseq t (λ[u1] • u2)⟧
   ⟹ elementary-reduction t for t
    using seq-cases sseq-imp-seq
    apply (cases t, simp-all)
    by fastforce
qed

lemma sseq-imp-elementary-reduction2:
shows sseq t u ⟹ elementary-reduction u
proof (induct u arbitrary: t)
  show ⋀t. sseq t ♯ ⟹ elementary-reduction ♯
    by simp
  show ⋀x t. sseq t «x» ⟹ elementary-reduction «x»
    using elementary-reduction.simps(2) sseq.elims(1) by blast
  show ⋀u. ⟦⋀t. sseq t u ⟹ elementary-reduction u; sseq t λ[u]⟧
             ⟹ elementary-reduction λ[u] for t
```

using *seq-cases sseq-imp-seq*
apply (*cases t*, *simp-all*)
by *force*
show $\bigwedge u1\ u2.\ [\![\bigwedge t.\ sseq\ t\ u1 \Longrightarrow elementary\text{-}reduction\ u1;$
$\bigwedge t.\ sseq\ t\ u2 \Longrightarrow elementary\text{-}reduction\ u2;$
$sseq\ t\ (u1 \circ u2)]\!]$
$\Longrightarrow elementary\text{-}reduction\ (u1 \circ u2)$ **for** *t*
using *seq-cases sseq-imp-seq Ide-Trg elementary-reduction-is-arr*
by (*cases t*) *auto*
show $\bigwedge u1\ u2.\ [\![\bigwedge t.\ sseq\ t\ u1 \Longrightarrow elementary\text{-}reduction\ u1;$
$\bigwedge t.\ sseq\ t\ u2 \Longrightarrow elementary\text{-}reduction\ u2;$
$sseq\ t\ (\boldsymbol{\lambda}[u1] \bullet u2)]\!]$
$\Longrightarrow elementary\text{-}reduction\ (\boldsymbol{\lambda}[u1] \bullet u2)$ **for** *t*
using *seq-cases sseq-imp-seq*
apply (*cases t*, *simp-all*)
by *fastforce*
qed

lemma *sseq-Beta*:
shows $sseq\ (\boldsymbol{\lambda}[t] \bullet u)\ v \longleftrightarrow Ide\ t \wedge Ide\ u \wedge elementary\text{-}reduction\ v \wedge seq\ (\boldsymbol{\lambda}[t] \bullet u)\ v$
by (*cases v*) *auto*

lemma *sseq-BetaI* [*intro*]:
assumes *Ide t* **and** *Ide u* **and** *elementary-reduction v* **and** $seq\ (\boldsymbol{\lambda}[t] \bullet u)\ v$
shows $sseq\ (\boldsymbol{\lambda}[t] \bullet u)\ v$
using *assms sseq-Beta* **by** *simp*

A head reduction is standardly sequential with any elementary reduction that can be performed after it.

lemma *sseq-head-reductionI*:
shows $[\![is\text{-}head\text{-}reduction\ t;\ elementary\text{-}reduction\ u;\ seq\ t\ u]\!] \Longrightarrow sseq\ t\ u$
proof (*induct t arbitrary*: *u*)
show $\bigwedge u.\ [\![is\text{-}head\text{-}reduction\ \sharp;\ elementary\text{-}reduction\ u;\ seq\ \sharp\ u]\!] \Longrightarrow sseq\ \sharp\ u$
by *simp*
show $\bigwedge x\ u.\ [\![is\text{-}head\text{-}reduction\ «x»;\ elementary\text{-}reduction\ u;\ seq\ «x»\ u]\!] \Longrightarrow sseq\ «x»\ u$
by *auto*
show $\bigwedge t.\ [\![\bigwedge u.\ [\![is\text{-}head\text{-}reduction\ t;\ elementary\text{-}reduction\ u;\ seq\ t\ u]\!] \Longrightarrow sseq\ t\ u;$
$is\text{-}head\text{-}reduction\ \boldsymbol{\lambda}[t];\ elementary\text{-}reduction\ u;\ seq\ \boldsymbol{\lambda}[t]\ u]\!]$
$\Longrightarrow sseq\ \boldsymbol{\lambda}[t]\ u$ **for** *u*
by (*cases u*) *auto*
show $\bigwedge t2.\ [\![\bigwedge u.\ [\![is\text{-}head\text{-}reduction\ t1;\ elementary\text{-}reduction\ u;\ seq\ t1\ u]\!] \Longrightarrow sseq\ t1\ u;$
$\bigwedge u.\ [\![is\text{-}head\text{-}reduction\ t2;\ elementary\text{-}reduction\ u;\ seq\ t2\ u]\!] \Longrightarrow sseq\ t2\ u;$
$is\text{-}head\text{-}reduction\ (t1 \circ t2);\ elementary\text{-}reduction\ u;\ seq\ (t1 \circ t2)\ u]\!]$
$\Longrightarrow sseq\ (t1 \circ t2)\ u$ **for** *t1 u*
using *seq-char*
apply (*cases u*)
apply *simp-all*
apply (*metis ArrE Ide-iff-Src-self Ide-iff-Trg-self App-Var-contains-no-head-reduction*
is-head-reduction-char is-head-reduction-imp-contains-head-reduction

```
      is-head-reduction.simps(3,6-7))
   by (cases t1) auto
 show ⋀t1 t2 u. ⟦⋀u. ⟦is-head-reduction t1; elementary-reduction u; seq t1 u⟧ ⟹ sseq t1 u;
                 ⋀u. ⟦is-head-reduction t2; elementary-reduction u; seq t2 u⟧ ⟹ sseq t2 u;
                 is-head-reduction (λ[t1] • t2); elementary-reduction u; seq (λ[t1] • t2) u⟧
                   ⟹ sseq (λ[t1] • t2) u
   by auto
qed
```

Once a head reduction is skipped in an application, then all terms that follow it in a standard reduction path are also applications that do not contain head reductions.

```
lemma sseq-preserves-App-and-no-head-reduction:
shows ⟦sseq t u; is-App t ∧ ¬ contains-head-reduction t⟧
        ⟹ is-App u ∧ ¬ contains-head-reduction u
 apply (induct t arbitrary: u)
    apply simp-all
proof −
 fix t1 t2 u
 assume ind1: ⋀u. ⟦sseq t1 u; is-App t1 ∧ ¬ contains-head-reduction t1⟧
                  ⟹ is-App u ∧ ¬ contains-head-reduction u
 assume ind2: ⋀u. ⟦sseq t2 u; is-App t2 ∧ ¬ contains-head-reduction t2⟧
                  ⟹ is-App u ∧ ¬ contains-head-reduction u
 assume sseq: sseq (t1 ∘ t2) u
 assume t: ¬ contains-head-reduction (t1 ∘ t2)
 have u: ¬ is-Beta u
  using sseq t sseq-imp-seq seq-cases
  by (cases t1; cases u) auto
 have 1: is-App u
   using u sseq sseq-imp-seq
   apply (cases u)
      apply simp-all
   by fastforce+
 moreover have ¬ contains-head-reduction u
 proof (cases u)
   show ⋀v. u = λ[v] ⟹ ¬ contains-head-reduction u
     using 1 by auto
   show ⋀v w. u = λ[v] • w ⟹ ¬ contains-head-reduction u
     using u by auto
   fix u1 u2
   assume u: u = u1 ∘ u2
   have 1: (sseq t1 u1 ∧ Ide t2 ∧ t2 = u2) ∨ (Ide t1 ∧ t1 = u1 ∧ sseq t2 u2) ∨
           (elementary-reduction t1 ∧ u1 = Trg t1 ∧ t2 = Src u2 ∧ elementary-reduction u2)
     using sseq u by force
   moreover have Ide t1 ∧ t1 = u1 ∧ sseq t2 u2 ⟹ ?thesis
     using Ide-implies-Arr ide-char sseq-imp-seq t u by fastforce
   moreover have elementary-reduction t1 ∧ u1 = Trg t1 ∧ t2 = Src u2 ∧
                 elementary-reduction u2
                   ⟹ ?thesis
   proof −
```

```
      assume 2: elementary-reduction t1 ∧ u1 = Trg t1 ∧ t2 = Src u2 ∧
                elementary-reduction u2
      have contains-head-reduction u ⟹ contains-head-reduction u1
        using u
        apply simp
        using contains-head-reduction.elims(2) by fastforce
      hence contains-head-reduction u ⟹ ¬ Ide u1
        using contains-head-reduction-iff
        by (metis Coinitial-iff-Con Ide-iff-Src-self Ide-implies-Arr ide-char resid-Arr-Src
            subs-head-redex subs-implies-prfx)
      thus ?thesis
        using 2
        by (metis Arr.simps(4) Ide-Trg seq-char sseq sseq-imp-seq)
    qed
    moreover have sseq t1 u1 ∧ Ide t2 ∧ t2 = u2 ⟹ ?thesis
      using t u ind1 [of u1] Ide-implies-Arr sseq-imp-elementary-reduction1
      apply (cases t1, simp-all)
      using elementary-reduction.simps(1)
        apply blast
      using elementary-reduction.simps(2)
       apply blast
      using contains-head-reduction.elims(2)
       apply fastforce
      apply (metis contains-head-reduction.simps(6) is-App-def)
      using sseq-Beta by blast
    ultimately show ?thesis by blast
  qed
  ultimately show is-App u ∧ ¬ contains-head-reduction u
    by blast
qed

end
```

Standard Reduction Paths

```
context reduction-paths
begin
```

A *standard reduction path* is an elementary reduction path in which successive reductions are standardly sequential.

```
  fun Std
  where Std [] = True
      | Std [t] = Λ.elementary-reduction t
      | Std (t # U) = (Λ.sseq t (hd U) ∧ Std U)

  lemma Std-consE [elim]:
  assumes Std (t # U)
  and ⟦Λ.Arr t; U ≠ [] ⟹ Λ.sseq t (hd U); Std U⟧ ⟹ thesis
  shows thesis
```

```
  using assms
  by (metis Λ.arr-char Λ.elementary-reduction-is-arr Λ.seq-char Λ.sseq-imp-seq
      list.exhaust-sel list.sel(1) Std.simps(1−3))

lemma Std-imp-Arr [simp]:
shows ⟦Std T; T ≠ []⟧ ⟹ Arr T
proof (induct T)
  show [] ≠ [] ⟹ Arr []
    by simp
  fix t U
  assume ind: ⟦Std U; U ≠ []⟧ ⟹ Arr U
  assume tU: Std (t # U)
  show Arr (t # U)
  proof (cases U = [])
    show U = [] ⟹ Arr (t # U)
      using Λ.elementary-reduction-is-arr tU Λ.Ide-implies-Arr Std.simps(2) Arr.simps(2)
      by blast
    assume U: U ≠ []
    show Arr (t # U)
    proof −
      have Λ.sseq t (hd U)
        using tU U
        by (metis list.exhaust-sel reduction-paths.Std.simps(3))
      thus ?thesis
        using U ind Λ.sseq-imp-seq
        apply auto
        using reduction-paths.Std.elims(3) tU
        by fastforce
    qed
  qed
qed

lemma Std-imp-sseq-last-hd:
shows ⟦Std (T @ U); T ≠ []; U ≠ []⟧ ⟹ Λ.sseq (last T) (hd U)
  apply (induct T arbitrary: U)
   apply simp-all
  by (metis Std.elims(3) Std.simps(3) append-self-conv2 neq-Nil-conv)

lemma Std-implies-set-subset-elementary-reduction:
shows Std U ⟹ set U ⊆ Collect Λ.elementary-reduction
  apply (induct U)
    apply auto
  by (metis Std.simps(2) Std.simps(3) neq-Nil-conv Λ.sseq-imp-elementary-reduction1)

lemma Std-map-Lam:
shows Std T ⟹ Std (map Λ.Lam T)
proof (induct T)
  show Std [] ⟹ Std (map Λ.Lam [])
    by simp
```

fix *t U*
assume *ind*: *Std U* $\Longrightarrow$ *Std* (*map* Λ.*Lam U*)
assume *tU*: *Std* (*t* # *U*)
have *Std* (*map* Λ.*Lam* (*t* # *U*)) $\longleftrightarrow$ *Std* ($\boldsymbol{\lambda}$[*t*] # *map* Λ.*Lam U*)
by *auto*
also have ... = *True*
apply (*cases U* = [])
apply *simp-all*
using *Arr.simps*(*3*) *Std.simps*(*2*) *arr-char tU*
apply *presburger*
proof −
assume *U*: *U* ≠ []
have *Std* ($\boldsymbol{\lambda}$[*t*] # *map* Λ.*Lam U*) $\longleftrightarrow$ Λ.*sseq* $\boldsymbol{\lambda}$[*t*] $\boldsymbol{\lambda}$[*hd U*] ∧ *Std* (*map* Λ.*Lam U*)
using *U*
by (*metis Nil-is-map-conv Std.simps*(*3*) *hd-map list.exhaust-sel*)
also have ... $\longleftrightarrow$ Λ.*sseq t* (*hd U*) ∧ *Std* (*map* Λ.*Lam U*)
by *auto*
also have ... = *True*
using *ind tU U*
by (*metis Std.simps*(*3*) *list.exhaust-sel*)
finally show *Std* ($\boldsymbol{\lambda}$[*t*] # *map* Λ.*Lam U*) **by** *blast*
qed
finally show *Std* (*map* Λ.*Lam* (*t* # *U*)) **by** *blast*
qed

lemma *Std-map-App1*:
shows ⟦Λ.*Ide b*; *Std T*⟧ $\Longrightarrow$ *Std* (*map* (λ*X*. *X* ∘ *b*) *T*)
proof (*induct T*)
show ⟦Λ.*Ide b*; *Std* []⟧ $\Longrightarrow$ *Std* (*map* (λ*X*. *X* ∘ *b*) [])
by *simp*
fix *t U*
assume *ind*: ⟦Λ.*Ide b*; *Std U*⟧ $\Longrightarrow$ *Std* (*map* (λ*X*. *X* ∘ *b*) *U*)
assume *b*: Λ.*Ide b*
assume *tU*: *Std* (*t* # *U*)
show *Std* (*map* (λ*v*. *v* ∘ *b*) (*t* # *U*))
proof (*cases U* = [])
show *U* = [] $\Longrightarrow$ *?thesis*
using *Ide-implies-Arr b* Λ.*arr-char tU* **by** *force*
assume *U*: *U* ≠ []
have *Std* (*map* (λ*v*. *v* ∘ *b*) (*t* # *U*)) = *Std* ((*t* ∘ *b*) # *map* (λ*X*. *X* ∘ *b*) *U*)
by *simp*
also have ... = (Λ.*sseq* (*t* ∘ *b*) (*hd U* ∘ *b*) ∧ *Std* (*map* (λ*X*. *X* ∘ *b*) *U*))
using *U reduction-paths.Std.simps*(*3*) *hd-map*
by (*metis Nil-is-map-conv neq-Nil-conv*)
also have ... = *True*
using *b tU U ind*
by (*metis Std.simps*(*3*) *list.exhaust-sel* Λ.*sseq.simps*(*4*))
finally show *Std* (*map* (λ*v*. *v* ∘ *b*) (*t* # *U*)) **by** *blast*
qed

qed

lemma *Std-map-App2*:
shows $[\![\Lambda.Ide\ a;\ Std\ T]\!] \Longrightarrow Std\ (map\ (\lambda u.\ a \circ u)\ T)$
proof (*induct T*)
  show $[\![\Lambda.Ide\ a;\ Std\ []]\!] \Longrightarrow Std\ (map\ (\lambda u.\ a \circ u)\ [])$
    by *simp*
  fix *t U*
  assume *ind*: $[\![\Lambda.Ide\ a;\ Std\ U]\!] \Longrightarrow Std\ (map\ (\lambda u.\ a \circ u)\ U)$
  assume *a*: $\Lambda.Ide\ a$
  assume *tU*: $Std\ (t\ \#\ U)$
  show $Std\ (map\ (\lambda u.\ a \circ u)\ (t\ \#\ U))$
  proof (*cases* $U = []$)
    show $U = [] \Longrightarrow ?thesis$
      using *a tU* **by** *force*
    assume *U*: $U \neq []$
    have $Std\ (map\ (\lambda u.\ a \circ u)\ (t\ \#\ U)) = Std\ ((a \circ t)\ \#\ map\ (\lambda u.\ a \circ u)\ U)$
      by *simp*
    also have $... = (\Lambda.sseq\ (a \circ t)\ (a \circ hd\ U) \wedge Std\ (map\ (\lambda u.\ a \circ u)\ U))$
      using *U*
      by (*metis Nil-is-map-conv Std.simps*(*3*) *hd-map list.exhaust-sel*)
    also have $... = True$
      using *a tU U ind*
      by (*metis Std.simps*(*3*) *list.exhaust-sel* $\Lambda.sseq.simps(4)$)
    finally show $Std\ (map\ (\lambda u.\ a \circ u)\ (t\ \#\ U))$ **by** *blast*
  qed
qed

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 (*induct T*)
  show $[\![Std\ []; set\ [] \subseteq Collect\ \Lambda.is\text{-}Lam]\!] \Longrightarrow Std\ (map\ \Lambda.un\text{-}Lam\ [])$
    by *simp*
  fix *t T*
  assume *ind*: $[\![Std\ T;\ set\ T \subseteq Collect\ \Lambda.is\text{-}Lam]\!] \Longrightarrow Std\ (map\ \Lambda.un\text{-}Lam\ T)$
  assume *tT*: $Std\ (t\ \#\ T)$
  assume *1*: $set\ (t\ \#\ T) \subseteq Collect\ \Lambda.is\text{-}Lam$
  show $Std\ (map\ \Lambda.un\text{-}Lam\ (t\ \#\ T))$
  proof (*cases* $T = []$)
    show $T = [] \Longrightarrow Std\ (map\ \Lambda.un\text{-}Lam\ (t\ \#\ T))$
    by (*metis 1 Std.simps*(*2*) $\Lambda.elementary\text{-}reduction.simps(3)$ $\Lambda.lambda.collapse(2)$
        list.set-intros(*1*) *list.simps*(*8*) *list.simps*(*9*) *mem-Collect-eq subset-code*(*1*) *tT*)
    assume *T*: $T \neq []$
    show $Std\ (map\ \Lambda.un\text{-}Lam\ (t\ \#\ T))$
     using *T tT 1 ind Std.simps*(*3*) [*of* $\Lambda.un\text{-}Lam\ t$ $\Lambda.un\text{-}Lam\ (hd\ T)$ *map* $\Lambda.un\text{-}Lam\ (tl\ T)$]
     by (*metis* $\Lambda.lambda.collapse(2)$ $\Lambda.sseq.simps(3)$ *list.exhaust-sel list.sel*(*1*)
        list.set-intros(*1*) *map-eq-Cons-conv mem-Collect-eq reduction-paths.Std.simps*(*3*)
        set-subset-Cons subset-code(*1*))
  qed

qed

lemma *Std-append-single*:
shows $[\![Std\ T;\ T \neq [];\ \Lambda.sseq\ (last\ T)\ u]\!] \Longrightarrow Std\ (T\ @\ [u])$
proof (*induct T*)
  show $[\![Std\ [];\ [] \neq [];\ \Lambda.sseq\ (last\ [])\ u]\!] \Longrightarrow Std\ ([]\ @\ [u])$
    by *blast*
  fix *t T*
  assume *ind*: $[\![Std\ T;\ T \neq [];\ \Lambda.sseq\ (last\ T)\ u]\!] \Longrightarrow Std\ (T\ @\ [u])$
  assume *tT*: $Std\ (t\ \#\ T)$
  assume *sseq*: $\Lambda.sseq\ (last\ (t\ \#\ T))\ u$
  have $Std\ (t\ \#\ (T\ @\ [u]))$
    using $\Lambda.sseq\text{-}imp\text{-}elementary\text{-}reduction2$ *sseq ind tT*
    apply (*cases* $T = []$)
     apply *simp*
    by (*metis append-Cons last-ConsR list.sel*(*1*) *neq-Nil-conv reduction-paths.Std.simps*(*3*))
  thus $Std\ ((t\ \#\ T)\ @\ [u])$ **by** *simp*
qed

lemma *Std-append*:
shows $[\![Std\ T;\ Std\ U;\ T = [] \vee U = [] \vee \Lambda.sseq\ (last\ T)\ (hd\ U)]\!] \Longrightarrow Std\ (T\ @\ U)$
proof (*induct U arbitrary*: *T*)
  show $\bigwedge T.\ [\![Std\ T;\ Std\ [];\ T = [] \vee [] = [] \vee \Lambda.sseq\ (last\ T)\ (hd\ [])]\!] \Longrightarrow Std\ (T\ @\ [])$
    by *simp*
  fix *u T U*
  assume *ind*: $\bigwedge T.\ [\![Std\ T;\ Std\ U;\ T = [] \vee U = [] \vee \Lambda.sseq\ (last\ T)\ (hd\ U)]\!]$
                $\Longrightarrow Std\ (T\ @\ U)$
  assume *T*: $Std\ T$
  assume *uU*: $Std\ (u\ \#\ U)$
  have *U*: $Std\ U$
    using *uU Std.elims*(*3*) **by** *fastforce*
  assume *seq*: $T = [] \vee u\ \#\ U = [] \vee \Lambda.sseq\ (last\ T)\ (hd\ (u\ \#\ U))$
  show $Std\ (T\ @\ (u\ \#\ U))$
    by (*metis Std-append-single T U append.assoc append.left-neutral append-Cons*
        ind last-snoc list.distinct(*1*) *list.exhaust-sel list.sel*(*1*) *Std.simps*(*3*) *seq uU*)
qed

Projections of Standard 'App Paths'

Given a standard reduction path, all of whose transitions have App as their top-level constructor, we can apply *un-App1* or *un-App2* to each transition to project the path onto paths formed from the "rator" and the "rand" of each application. These projected paths are not standard, since the projection operation will introduce identities, in general. However, in this section we show that if we remove the identities, then in fact we do obtain standard reduction paths.

abbreviation *notIde*
where $notIde \equiv \lambda u.\ \neg\ \Lambda.Ide\ u$

lemma *filter-notIde-Ide*:
shows *Ide U* $\Longrightarrow$ *filter notIde U* = []
by (*induct U*) *auto*

lemma *cong-filter-notIde*:
shows [[*Arr U*; ¬ *Ide U*]] $\Longrightarrow$ *filter notIde U* $^*\sim^*$ *U*
proof (*induct U*)
show [[*Arr* []; ¬ *Ide* []]] $\Longrightarrow$ *filter notIde* [] $^*\sim^*$ []
by *simp*
fix *u U*
assume *ind*: [[*Arr U*; ¬ *Ide U*]] $\Longrightarrow$ *filter notIde U* $^*\sim^*$ *U*
assume *Arr*: *Arr* (*u* # *U*)
assume *1*: ¬ *Ide* (*u* # *U*)
show *filter notIde* (*u* # *U*) $^*\sim^*$ (*u* # *U*)
proof (*cases* Λ.*Ide u*)
assume *u*: Λ.*Ide u*
have *U*: *Arr U* ∧ ¬ *Ide U*
using *Arr u 1 Ide.elims*(*3*) **by** *fastforce*
have *filter notIde* (*u* # *U*) = *filter notIde U*
using *u* **by** *simp*
also have ... $^*\sim^*$ *U*
using *U ind* **by** *blast*
also have *U* $^*\sim^*$ [*u*] @ *U*
using *u*
by (*metis* (*full-types*) *Arr Arr-has-Src Cons-eq-append-conv Ide.elims*(*3*) *Ide.simps*(*2*)
Srcs.simps(*1*) *U arrI*$_P$ *arr-append-imp-seq cong-append-ideI*(*3*) *ide-char*
Λ.*ide-char not-Cons-self2*)
also have [*u*] @ *U* = *u* # *U*
by *simp*
finally show *?thesis* **by** *blast*
next
assume *u*: ¬ Λ.*Ide u*
show *?thesis*
proof (*cases Ide U*)
assume *U*: *Ide U*
have *filter notIde* (*u* # *U*) = [*u*]
using *u U filter-notIde-Ide* **by** *simp*
moreover have [*u*] $^*\sim^*$ [*u*] @ *U*
using *u U cong-append-ideI*(*4*) [*of* [*u*] *U*]
by (*metis Arr Con-Arr-self Cons-eq-appendI Resid-Ide*(*1*) *arr-append-imp-seq*
arr-char ide-char ide-implies-arr neq-Nil-conv self-append-conv2)
moreover have [*u*] @ *U* = *u* # *U*
by *simp*
ultimately show *?thesis* **by** *auto*
next
assume *U*: ¬ *Ide U*
have *filter notIde* (*u* # *U*) = [*u*] @ *filter notIde U*
using *u U Arr* **by** *simp*
also have ... $^*\sim^*$ [*u*] @ *U*

```
    proof (cases U = [])
      show U = [] ⟹ ?thesis
        by (metis Arr arr-char cong-reflexive append-Nil2 filter.simps(1))
      assume 1: U ≠ []
      have seq [u] (filter notIde U)
        by (metis (full-types) 1 Arr Arr.simps(2−3) Con-imp-eq-Srcs Con-implies-Arr(1)
            Ide.elims(3) Ide.simps(1) Trgs.simps(2) U ide-char ind seq-char
            seq-implies-Trgs-eq-Srcs)
      thus ?thesis
        using u U Arr ind cong-append [of [u] filter notIde U [u] U]
        by (meson 1 Arr-consE cong-reflexive seqE)
    qed
    also have [u] @ U = u # U
      by simp
    finally show ?thesis by argo
  qed
 qed
qed

lemma Std-filter-map-un-App1:
shows ⟦Std U; set U ⊆ Collect Λ.is-App⟧ ⟹ Std (filter notIde (map Λ.un-App1 U))
proof (induct U)
 show ⟦Std []; set [] ⊆ Collect Λ.is-App⟧ ⟹ Std (filter notIde (map Λ.un-App1 []))
   by simp
 fix u U
 assume ind: ⟦Std U; set U ⊆ Collect Λ.is-App⟧ ⟹ Std (filter notIde (map Λ.un-App1 U))
 assume 1: Std (u # U)
 assume 2: set (u # U) ⊆ Collect Λ.is-App
 show Std (filter notIde (map Λ.un-App1 (u # U)))
   using 1 2 ind
   apply (cases u)
     apply simp-all
 proof −
   fix u1 u2
   assume uU: Std ((u1 ∘ u2) # U)
   assume set: set U ⊆ Collect Λ.is-App
   assume ind: Std U ⟹ Std (filter notIde (map Λ.un-App1 U))
   assume u: u = u1 ∘ u2
   show (¬ Λ.Ide u1 ⟶ Std (u1 # filter notIde (map Λ.un-App1 U))) ∧
        (Λ.Ide u1 ⟶ Std (filter notIde (map Λ.un-App1 U)))
   proof (intro conjI impI)
     assume u1: Λ.Ide u1
     show Std (filter notIde (map Λ.un-App1 U))
       by (metis 1 Std.simps(1) Std.simps(3) ind neq-Nil-conv)
     next
     assume u1: ¬ Λ.Ide u1
     show Std (u1 # filter notIde (map Λ.un-App1 U))
     proof (cases Ide (map Λ.un-App1 U))
       show Ide (map Λ.un-App1 U) ⟹ ?thesis
```

```
proof −
  assume U: Ide (map Λ.un-App1 U)
  have filter notIde (map Λ.un-App1 U) = []
    by (metis U Ide-char filter-False Λ.ide-char
        mem-Collect-eq subsetD)
  thus ?thesis
    by (metis Std.elims(1) Std.simps(2) Λ.elementary-reduction.simps(4) list.discI
        list.sel(1) Λ.sseq-imp-elementary-reduction1 u1 uU)
qed
assume U: ¬ Ide (map Λ.un-App1 U)
show ?thesis
proof (cases U = [])
  show U = [] ⟹ ?thesis
    using 1 u u1 by fastforce
  assume U ≠ []
  hence U: U ≠ [] ∧ ¬ Ide (map Λ.un-App1 U)
    using U by simp
  have Λ.sseq u1 (hd (filter notIde (map Λ.un-App1 U)))
  proof −
    have ⋀u1 u2. ⟦set U ⊆ Collect Λ.is-App; ¬ Ide (map Λ.un-App1 U); U ≠ [];
                  Std ((u1 ∘ u2) # U); ¬ Λ.Ide u1⟧
                    ⟹ Λ.sseq u1 (hd (filter notIde (map Λ.un-App1 U)))
      for U
      apply (induct U)
       apply simp-all
      apply (intro conjI impI)
    proof −
      fix u U u1 u2
      assume ind: ⋀u1 u2. ⟦¬ Ide (map Λ.un-App1 U); U ≠ [];
                           Std ((u1 ∘ u2) # U); ¬ Λ.Ide u1⟧
                             ⟹ Λ.sseq u1 (hd (filter notIde (map Λ.un-App1 U)))
      assume 1: Λ.is-App u ∧ set U ⊆ Collect Λ.is-App
      assume 2: ¬ Ide (Λ.un-App1 u # map Λ.un-App1 U)
      assume 3: Λ.sseq (u1 ∘ u2) u ∧ Std (u # U)
      show ¬ Λ.Ide (Λ.un-App1 u) ⟹ Λ.sseq u1 (Λ.un-App1 u)
        by (metis 1 3 Λ.Arr.simps(4) Λ.Ide-Trg Λ.lambda.collapse(3) Λ.seq-char
            Λ.sseq.simps(4) Λ.sseq-imp-seq)
      assume 4: ¬ Λ.Ide u1
      assume 5: Λ.Ide (Λ.un-App1 u)
      have u1: Λ.elementary-reduction u1
        using 3 4 Λ.elementary-reduction.simps(4) Λ.sseq-imp-elementary-reduction1
        by blast
      have 6: Arr (Λ.un-App1 u # map Λ.un-App1 U)
        using 1 3 Std-imp-Arr Arr-map-un-App1 [of u # U] by auto
      have 7: Arr (map Λ.un-App1 U)
        using 1 2 3 5 6 Arr-map-un-App1 Std-imp-Arr Λ.ide-char by fastforce
      have 8: ¬ Ide (map Λ.un-App1 U)
        using 2 5 6 set-Ide-subset-ide by fastforce
      have 9: Λ.seq u (hd U)
```

```
  by (metis 3 7 Std.simps(3) Arr.simps(1) list.collapse list.simps(8)
      Λ.sseq-imp-seq)
show Λ.sseq u1 (hd (filter notIde (map Λ.un-App1 U)))
proof −
  have Λ.sseq (u1 ∘ Λ.Trg (Λ.un-App2 u)) (hd U)
  proof (cases Λ.Ide (Λ.un-App1 (hd U)))
    assume 10: Λ.Ide (Λ.un-App1 (hd U))
    hence Λ.elementary-reduction (Λ.un-App2 (hd U))
      by (metis (full-types) 1 3 7 Std.elims(2) Arr.simps(1)
          Λ.elementary-reduction-App-iff Λ.elementary-reduction-not-ide
          Λ.ide-char list.sel(2) list.sel(3) list.set-sel(1) list.simps(8)
          mem-Collect-eq Λ.sseq-imp-elementary-reduction2 subsetD)
    moreover have Λ.Trg u1 = Λ.un-App1 (hd U)
    proof −
      have Λ.Trg u1 = Λ.Src (Λ.un-App1 u)
        by (metis 1 3 5 Λ.Ide-iff-Src-self Λ.Ide-implies-Arr Λ.Trg-Src
            Λ.elementary-reduction-not-ide Λ.ide-char Λ.lambda.collapse(3)
            Λ.sseq.simps(4) Λ.sseq-imp-elementary-reduction2)
      also have ... = Λ.Trg (Λ.un-App1 u)
        by (metis 5 Λ.Ide-iff-Src-self Λ.Ide-iff-Trg-self
            Λ.Ide-implies-Arr)
      also have ... = Λ.un-App1 (hd U)
        using 1 3 5 7 Λ.Ide-iff-Trg-self
        by (metis 9 10 Arr.simps(1) lambda-calculus.Ide-iff-Src-self
            Λ.Ide-implies-Arr Λ.Src-Src Λ.Src-eq-iff(2) Λ.Trg.simps(3)
            Λ.lambda.collapse(3) Λ.seqE_Λ list.set-sel(1) list.simps(8)
            mem-Collect-eq subsetD)
      finally show ?thesis by argo
    qed
    moreover have Λ.Trg (Λ.un-App2 u) = Λ.Src (Λ.un-App2 (hd U))
      by (metis 1 7 9 Arr.simps(1) hd-in-set Λ.Src.simps(4) Λ.Src-Src
          Λ.Trg.simps(3) Λ.lambda.collapse(3) Λ.lambda.sel(4)
          Λ.seq-char list.simps(8) mem-Collect-eq subset-code(1))
    ultimately show ?thesis
      using Λ.sseq.simps(4)
      by (metis 1 7 u1 Arr.simps(1) hd-in-set Λ.lambda.collapse(3)
          list.simps(8) mem-Collect-eq subsetD)
    next
    assume 10: ¬ Λ.Ide (Λ.un-App1 (hd U))
    have False
    proof −
      have Λ.elementary-reduction (Λ.un-App2 u)
        using 1 3 5 Λ.elementary-reduction-App-iff
              Λ.elementary-reduction-not-ide Λ.sseq-imp-elementary-reduction2
        by blast
      moreover have Λ.sseq u (hd U)
        by (metis 3 7 Std.simps(3) Arr.simps(1)
            hd-Cons-tl list.simps(8))
      moreover have Λ.elementary-reduction (Λ.un-App1 (hd U))
```

```
            by (metis 1 7 10 Nil-is-map-conv Arr.simps(1)
                calculation(2) Λ.elementary-reduction-App-iff hd-in-set Λ.ide-char
                mem-Collect-eq Λ.sseq-imp-elementary-reduction2 subset-iff)
          ultimately show ?thesis
            using Λ.sseq.simps(4)
            by (metis 1 5 7 Arr.simps(1) Λ.elementary-reduction-not-ide
                hd-in-set Λ.ide-char Λ.lambda.collapse(3) list.simps(8)
                mem-Collect-eq subset-iff)
        qed
        thus ?thesis by argo
      qed
      hence Std ((u1 ∘ Λ.Trg (Λ.un-App2 u)) # U)
        by (metis 3 7 Std.simps(3) Arr.simps(1) list.exhaust-sel list.simps(8))
      thus ?thesis
        using ind
        by (metis 7 8 u1 Arr.simps(1) Λ.elementary-reduction-not-ide Λ.ide-char
            list.simps(8))
    qed
  qed
  thus ?thesis
    using U set u1 uU by blast
qed
thus ?thesis
  by (metis 1 Std.simps(2-3) ‹U ≠ []› ind list.exhaust-sel list.sel(1)
      Λ.sseq-imp-elementary-reduction1)
qed
qed
qed
qed

lemma Std-filter-map-un-App2:
shows ⟦Std U; set U ⊆ Collect Λ.is-App⟧ ⟹ Std (filter notIde (map Λ.un-App2 U))
proof (induct U)
  show ⟦Std []; set [] ⊆ Collect Λ.is-App⟧ ⟹ Std (filter notIde (map Λ.un-App2 []))
    by simp
  fix u U
 assume ind: ⟦Std U; set U ⊆ Collect Λ.is-App⟧ ⟹ Std (filter notIde (map Λ.un-App2 U))
  assume 1: Std (u # U)
  assume 2: set (u # U) ⊆ Collect Λ.is-App
  show Std (filter notIde (map Λ.un-App2 (u # U)))
    using 1 2 ind
    apply (cases u)
      apply simp-all
  proof -
    fix u1 u2
    assume uU: Std ((u1 ∘ u2) # U)
    assume set: set U ⊆ Collect Λ.is-App
    assume ind: Std U ⟹ Std (filter notIde (map Λ.un-App2 U))
```

```
assume u: u = u1 ∘ u2
show (¬ Λ.Ide u2 ⟶ Std (u2 # filter notIde (map Λ.un-App2 U))) ∧
     (Λ.Ide u2 ⟶ Std (filter notIde (map Λ.un-App2 U)))
proof (intro conjI impI)
  assume u2: Λ.Ide u2
  show Std (filter notIde (map Λ.un-App2 U))
    by (metis 1 Std.simps(1) Std.simps(3) ind neq-Nil-conv)
  next
  assume u2: ¬ Λ.Ide u2
  show Std (u2 # filter notIde (map Λ.un-App2 U))
  proof (cases Ide (map Λ.un-App2 U))
    show Ide (map Λ.un-App2 U) ⟹ ?thesis
    proof −
      assume U: Ide (map Λ.un-App2 U)
      have filter notIde (map Λ.un-App2 U) = []
        by (metis U Ide-char filter-False Λ.ide-char mem-Collect-eq subsetD)
      thus ?thesis
        by (metis Std.elims(1) Std.simps(2) Λ.elementary-reduction.simps(4) list.discI
            list.sel(1) Λ.sseq-imp-elementary-reduction1 u2 uU)
    qed
    assume U: ¬ Ide (map Λ.un-App2 U)
    show ?thesis
    proof (cases U = [])
      show U = [] ⟹ ?thesis
        using 1 u u2 by fastforce
      assume U ≠ []
      hence U: U ≠ [] ∧ ¬ Ide (map Λ.un-App2 U)
        using U by simp
      have Λ.sseq u2 (hd (filter notIde (map Λ.un-App2 U)))
      proof −
        have ⋀u1 u2. ⟦set U ⊆ Collect Λ.is-App; ¬ Ide (map Λ.un-App2 U); U ≠ [];
                      Std ((u1 ∘ u2) # U); ¬ Λ.Ide u2⟧
                        ⟹ Λ.sseq u2 (hd (filter notIde (map Λ.un-App2 U)))
          for U
          apply (induct U)
          apply simp-all
          apply (intro conjI impI)
        proof −
          fix u U u1 u2
          assume ind: ⋀u1 u2. ⟦¬ Ide (map Λ.un-App2 U); U ≠ [];
                               Std ((u1 ∘ u2) # U); ¬ Λ.Ide u2⟧
                                 ⟹ Λ.sseq u2 (hd (filter notIde (map Λ.un-App2 U)))
          assume 1: Λ.is-App u ∧ set U ⊆ Collect Λ.is-App
          assume 2: ¬ Ide (Λ.un-App2 u # map Λ.un-App2 U)
          assume 3: Λ.sseq (u1 ∘ u2) u ∧ Std (u # U)
          assume 4: ¬ Λ.Ide u2
          show ¬ Λ.Ide (Λ.un-App2 u) ⟹ Λ.sseq u2 (Λ.un-App2 u)
            by (metis 1 3 4 Λ.elementary-reduction.simps(4)
                Λ.elementary-reduction-not-ide Λ.ide-char Λ.lambda.collapse(3)
```

```
            Λ.sseq.simps(4) Λ.sseq-imp-elementary-reduction1)
        assume 5: Λ.Ide (Λ.un-App2 u)
        have False
          by (metis 1 3 4 5 Λ.elementary-reduction-not-ide Λ.ide-char
              Λ.lambda.collapse(3) Λ.sseq.simps(4) Λ.sseq-imp-elementary-reduction2)
        thus Λ.sseq u2 (hd (filter notIde (map Λ.un-App2 U))) by argo
      qed
      thus ?thesis
        using U set u2 uU by blast
    qed
    thus ?thesis
      by (metis 1 Std.simps(2) Std.simps(3) ‹U ≠ []› ind list.exhaust-sel list.sel(1)
          Λ.sseq-imp-elementary-reduction1)
  qed
 qed
qed
qed
```

If the first step in a standard reduction path contracts a redex that is not at the head position, then all subsequent terms have *App* as their top-level operator.

```
lemma seq-App-Std-implies:
shows ⟦Std (t # U); Λ.is-App t ∧ ¬ Λ.contains-head-reduction t⟧
        ⟹ set U ⊆ Collect Λ.is-App
proof (induct U arbitrary: t)
  show ⋀t. ⟦Std [t]; Λ.is-App t ∧ ¬ Λ.contains-head-reduction t⟧
             ⟹ set [] ⊆ Collect Λ.is-App
    by simp
  fix t u U
  assume ind: ⋀t. ⟦Std (t # U); Λ.is-App t ∧ ¬ Λ.contains-head-reduction t⟧
                    ⟹ set U ⊆ Collect Λ.is-App
  assume Std: Std (t # u # U)
  assume t: Λ.is-App t ∧ ¬ Λ.contains-head-reduction t
  have U: set (u # U) ⊆ Collect Λ.elementary-reduction
    using Std Std-implies-set-subset-elementary-reduction by fastforce
  have u: Λ.elementary-reduction u
    using U by simp
  have set U ⊆ Collect Λ.elementary-reduction
    using U by simp
  show set (u # U) ⊆ Collect Λ.is-App
  proof (cases U = [])
    show U = [] ⟹ ?thesis
      by (metis Std empty-set empty-subsetI insert-subset
          Λ.sseq-preserves-App-and-no-head-reduction list.sel(1) list.simps(15)
          mem-Collect-eq reduction-paths.Std.simps(3) t)
    assume U: U ≠ []
    have Λ.sseq t u
      using Std by auto
    hence Λ.is-App u ∧ ¬ Λ.Ide u ∧ ¬ Λ.contains-head-reduction u
```

```
      using t u U Λ.sseq-preserves-App-and-no-head-reduction [of t u]
          Λ.elementary-reduction-not-ide
      by blast
    thus ?thesis
      using Std ind [of u] ‹set U ⊆ Collect Λ.elementary-reduction› by simp
  qed
qed
```

3.6.2 Standard Developments

The following function takes a term t (representing a parallel reduction) and produces a standard reduction path that is a complete development of t and is thus congruent to $[t]$. The proof of termination makes use of the Finite Development Theorem.

```
function (sequential) standard-development
where standard-development ♯ = []
  | standard-development «-» = []
  | standard-development λ[t] = map Λ.Lam (standard-development t)
  | standard-development (t ∘ u) =
      (if Λ.Arr t ∧ Λ.Arr u then
         map (λv. v ∘ Λ.Src u) (standard-development t) @
         map (λv. 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 [])
  by pat-completeness auto

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 −
  have (λ[t] • u) \ (λ[Src t] • Src u) = subst u t
    using assms
    by (metis Arr-not-Nil Ide-Src Ide-iff-Src-self Ide-implies-Arr resid.simps(4)
        resid-Arr-Ide)
  moreover have elementary-reduction (λ[Src t] • Src u)
    by (simp add: assms Ide-Src)
  moreover have λ[Src t] • Src u ⊑ λ[t] • u
    by (metis assms Arr.simps(5) head-redex.simps(9) subs-head-redex)
  ultimately show ?thesis
    using assms elementary-reduction-decreases-hgt [of λ[Src t] • Src u λ[t] • u]
    by (metis mlex-less)
qed

termination standard-development
```

proof (*relation* Λ.*stddev-term-rel*)
  show *wf* Λ.*stddev-term-rel*
    using Λ.*wf-subterm-rel wf-mlex* **by** *blast*
  show ⋀*t*. (*t*, **λ**[*t*]) ∈ Λ.*stddev-term-rel*
    by (*simp add*: Λ.*subterm-lemmas*(*1*) *mlex-prod-def*)
  show ⋀*t u*. (*t*, *t* ∘ *u*) ∈ Λ.*stddev-term-rel*
    using Λ.*subterm-lemmas*(*3*)
    by (*metis antisym-conv1* Λ.*hgt.simps*(*4*) *le-add1 mem-Collect-eq mlex-iff old.prod.case*)
  show ⋀*t u*. (*u*, *t* ∘ *u*) ∈ Λ.*stddev-term-rel*
    using Λ.*subterm-lemmas*(*3*) **by** (*simp add*: *mlex-leq*)
  show ⋀*t u*. Λ.*Arr t* ∧ Λ.*Arr u* ⟹ (Λ.*subst u t*, **λ**[*t*] • *u*) ∈ Λ.*stddev-term-rel*
    using Λ.*subst-lt-Beta* **by** *simp*
qed

lemma *Ide-iff-standard-development-empty*:
shows Λ.*Arr t* ⟹ Λ.*Ide t* ⟷ *standard-development t* = []
  by (*induct t*) *auto*

lemma *set-standard-development*:
shows Λ.*Arr t* ⟶ *set* (*standard-development t*) ⊆ *Collect* Λ.*elementary-reduction*
  apply (*rule standard-development.induct*)
  using Λ.*Ide-Src* Λ.*Ide-Trg* Λ.*Arr-Subst* **by** *auto*

lemma *cong-standard-development*:
shows Λ.*Arr t* ∧ ¬ Λ.*Ide t* ⟶ *standard-development t* *∼* [*t*]
proof (*rule standard-development.induct*)
 show Λ.*Arr* ♯ ∧ ¬ Λ.*Ide* ♯ ⟶ *standard-development* ♯ *∼* [♯]
    by *simp*
  show ⋀*x*. Λ.*Arr* «*x*» ∧ ¬ Λ.*Ide* «*x*»
            ⟶ *standard-development* «*x*» *∼* [«*x*»]
    by *simp*
  show ⋀*t*. Λ.*Arr t* ∧ ¬ Λ.*Ide t* ⟶ *standard-development t* *∼* [*t*] ⟹
          Λ.*Arr* **λ**[*t*] ∧ ¬ Λ.*Ide* **λ**[*t*] ⟶ *standard-development* **λ**[*t*] *∼* [**λ**[*t*]]
    by (*metis* (*mono-tags*, *lifting*) *cong-map-Lam* Λ.*Arr.simps*(*3*) Λ.*Ide.simps*(*3*)
        list.simps(*8*,*9*) *standard-development.simps*(*3*))
  show ⋀*t u*. ⟦Λ.*Arr t* ∧ Λ.*Arr u*
              ⟹ Λ.*Arr t* ∧ ¬ Λ.*Ide t* ⟶ *standard-development t* *∼* [*t*];
            Λ.*Arr t* ∧ Λ.*Arr u*
              ⟹ Λ.*Arr u* ∧ ¬ Λ.*Ide u* ⟶ *standard-development u* *∼* [*u*]⟧
                ⟹ Λ.*Arr* (*t* ∘ *u*) ∧ ¬ Λ.*Ide* (*t* ∘ *u*) ⟶
                      standard-development (*t* ∘ *u*) *∼* [*t* ∘ *u*]
  proof
    fix *t u*
    assume *ind1*: Λ.*Arr t* ∧ Λ.*Arr u*
                  ⟹ Λ.*Arr t* ∧ ¬ Λ.*Ide t* ⟶ *standard-development t* *∼* [*t*]
    assume *ind2*: Λ.*Arr t* ∧ Λ.*Arr u*
                  ⟹ Λ.*Arr u* ∧ ¬ Λ.*Ide u* ⟶ *standard-development u* *∼* [*u*]
    assume *1*: Λ.*Arr* (*t* ∘ *u*) ∧ ¬ Λ.*Ide* (*t* ∘ *u*)
    show *standard-development* (*t* ∘ *u*) *∼* [*t* ∘ *u*]

```
proof (cases standard-development t = [])
  show standard-development t = [] ⟹ ?thesis
    using 1 ind2 cong-map-App1 Ide-iff-standard-development-empty Λ.Ide-iff-Trg-self
    apply simp
    by (metis (no-types, opaque-lifting) list.simps(8,9))
  assume t: standard-development t ≠ []
  show ?thesis
  proof (cases standard-development u = [])
    assume u: standard-development u = []
    have standard-development (t ∘ u) = map (λX. X ∘ u) (standard-development t)
      using u 1 Λ.Ide-iff-Src-self ide-char ind2 by auto
    also have ... *∼* map (λa. a ∘ u) [t]
      using cong-map-App2 [of u]
      by (meson 1 Λ.Arr.simps(4) Ide-iff-standard-development-empty t u ind1)
    also have map (λa. a ∘ u) [t] = [t ∘ u]
      by simp
    finally show ?thesis by blast
    next
    assume u: standard-development u ≠ []
    have standard-development (t ∘ u) =
          map (λa. a ∘ Λ.Src u) (standard-development t) @
          map (λb. Λ.Trg t ∘ b) (standard-development u)
      using 1 by force
    moreover have map (λa. a ∘ Λ.Src u) (standard-development t) *∼* [t ∘ Λ.Src u]
    proof −
     have map (λa. a ∘ Λ.Src u) (standard-development t) *∼* map (λa. a ∘ Λ.Src u) [t]
       using t u 1 ind1 Λ.Ide-Src Ide-iff-standard-development-empty cong-map-App2
       by (metis Λ.Arr.simps(4))
     also have map (λa. a ∘ Λ.Src u) [t] = [t ∘ Λ.Src u]
       by simp
     finally show ?thesis by blast
    qed
    moreover have map (λb. Λ.Trg t ∘ b) (standard-development u) *∼* [Λ.Trg t ∘ u]
      using t u 1 ind2 Λ.Ide-Trg Ide-iff-standard-development-empty cong-map-App1
      by (metis (mono-tags, opaque-lifting) Λ.Arr.simps(4) list.simps(8,9))
    moreover have seq (map (λa. a ∘ Λ.Src u) (standard-development t))
                      (map (λb. Λ.Trg t ∘ b) (standard-development u))
    proof
      show Arr (map (λa. a ∘ Λ.Src u) (standard-development t))
        by (metis Con-implies-Arr(1) Ide.simps(1) calculation(2) ide-char)
      show Arr (map ((∘) (Λ.Trg t)) (standard-development u))
        by (metis Con-implies-Arr(1) Ide.simps(1) calculation(3) ide-char)
      show Λ.Trg (last (map (λa. a ∘ Λ.Src u) (standard-development t))) =
           Λ.Src (hd (map ((∘) (Λ.Trg t)) (standard-development u)))
        using 1 Src-hd-eqI Trg-last-eqI calculation(2) calculation(3) by auto
    qed
    ultimately have standard-development (t ∘ u) *∼* [t ∘ Λ.Src u] @ [Λ.Trg t ∘ u]
      using cong-append [of map (λa. a ∘ Λ.Src u) (standard-development t)
                            map (λb. Λ.Trg t ∘ b) (standard-development u)
```

```
                    [t ∘ Λ.Src u] [Λ.Trg t ∘ u]]
      by simp
    moreover have [t ∘ Λ.Src u] @ [Λ.Trg t ∘ u] *∼* [t ∘ u]
      using 1 Λ.Ide-Trg Λ.resid-Arr-Src Λ.resid-Arr-self Λ.null-char
            ide-char Λ.Arr-not-Nil
      by simp
    ultimately show ?thesis
      using cong-transitive by blast
  qed
 qed
qed
show ⋀t u. (Λ.Arr t ∧ Λ.Arr u ⟹
              Λ.Arr (Λ.subst u t) ∧ ¬ Λ.Ide (Λ.subst u t)
                ⟶ standard-development (Λ.subst u t) *∼* [Λ.subst u t]) ⟹
            Λ.Arr (λ[t] • u) ∧ ¬ Λ.Ide (λ[t] • u) ⟶
              standard-development (λ[t] • u) *∼* [λ[t] • u]
proof
  fix t u
  assume 1: Λ.Arr (λ[t] • u) ∧ ¬ Λ.Ide (λ[t] • u)
  assume ind: Λ.Arr t ∧ Λ.Arr u ⟹
                Λ.Arr (Λ.subst u t) ∧ ¬ Λ.Ide (Λ.subst u t)
                  ⟶ standard-development (Λ.subst u t) *∼* [Λ.subst u t]
  show standard-development (λ[t] • u) *∼* [λ[t] • u]
  proof (cases Λ.Ide (Λ.subst u t))
    assume 2: Λ.Ide (Λ.subst u t)
    have standard-development (λ[t] • u) = [λ[Λ.Src t] • Λ.Src u]
      using 1 2 Ide-iff-standard-development-empty [of Λ.subst u t] Λ.Arr-Subst
      by simp
    also have [λ[Λ.Src t] • Λ.Src u] *∼* [λ[t] • u]
      using 1 2 Λ.Ide-Src Λ.Ide-implies-Arr ide-char Λ.resid-Arr-Ide
      apply (intro conjI)
       apply simp-all
       apply (metis Λ.Ide.simps(1) Λ.Ide-Subst-iff Λ.Ide-Trg)
      by fastforce
    finally show ?thesis by blast
    next
    assume 2: ¬ Λ.Ide (Λ.subst u t)
    have standard-development (λ[t] • u) =
          [λ[Λ.Src t] • Λ.Src u] @ standard-development (Λ.subst u t)
      using 1 by auto
    also have [λ[Λ.Src t] • Λ.Src u] @ standard-development (Λ.subst u t) *∼*
              [λ[Λ.Src t] • Λ.Src u] @ [Λ.subst u t]
    proof (intro cong-append)
      show seq [Λ.Beta (Λ.Src t) (Λ.Src u)] (standard-development (Λ.subst u t))
        using 1 2 ind arr-char ide-implies-arr Λ.Arr-Subst Con-implies-Arr(1) Src-hd-eqI
        apply (intro seqI_ΛP)
          apply simp-all
        by (metis Arr.simps(1))
      show [λ[Λ.Src t] • Λ.Src u] *∼* [λ[Λ.Src t] • Λ.Src u]
```

```
    using 1
    by (metis Λ.Arr.simps(5) Λ.Ide-Src Λ.Ide-implies-Arr Arr.simps(2) Resid-Arr-self
        ide-char Λ.arr-char)
  show standard-development (Λ.subst u t) *∼* [Λ.subst u t]
    using 1 2 Λ.Arr-Subst ind by simp
qed
also have [λ[Λ.Src t] • Λ.Src u] @ [Λ.subst u t] *∼* [λ[t] • u]
proof
  show [λ[Λ.Src t] • Λ.Src u] @ [Λ.subst u t] *≲* [λ[t] • u]
  proof −
    have t \ Λ.Src t ≠ ♯ ∧ u \ Λ.Src u ≠ ♯
      by (metis 1 Λ.Arr.simps(5) Λ.Coinitial-iff-Con Λ.Ide-Src Λ.Ide-iff-Src-self
          Λ.Ide-implies-Arr)
    moreover have Λ.con (λ[Λ.Src t] • Λ.Src u) (λ[t] • u)
      by (metis 1 Λ.head-redex.simps(9) Λ.prfx-implies-con Λ.subs-head-redex
          Λ.subs-implies-prfx)
    ultimately have ([λ[Λ.Src t] • Λ.Src u] @ [Λ.subst u t]) *\* [λ[t] • u] =
                    [λ[Λ.Src t] • Λ.Src u] *\* [λ[t] • u] @
                      [Λ.subst u t] *\* ([λ[t] • u] *\* [λ[Λ.Src t] • Λ.Src u])
      using Resid-append(1)
            [of [λ[Λ.Src t] • Λ.Src u] [Λ.subst u t] [λ[t] • u]]
      apply simp
      by (metis Λ.Arr-Subst Λ.Coinitial-iff-Con Λ.Ide-Src Λ.resid-Arr-Ide)
    also have ... = [Λ.subst (Λ.Trg u) (Λ.Trg t)] @ ([Λ.subst u t] *\* [Λ.subst u t])
    proof −
      have t \ Λ.Src t ≠ ♯ ∧ u \ Λ.Src u ≠ ♯
        by (metis 1 Λ.Arr.simps(5) Λ.Coinitial-iff-Con Λ.Ide-Src
            Λ.Ide-iff-Src-self Λ.Ide-implies-Arr)
      moreover have Λ.Src t \ t ≠ ♯ ∧ Λ.Src u \ u ≠ ♯
        using Λ.Con-sym calculation(1) by presburger
      moreover have Λ.con (Λ.subst u t) (Λ.subst u t)
        by (meson Λ.Arr-Subst Λ.Con-implies-Arr2 Λ.arr-char Λ.arr-def calculation(2))
      moreover have Λ.con (λ[t] • u) (λ[Λ.Src t] • Λ.Src u)
        using ‹Λ.con (λ[Λ.Src t] • Λ.Src u) (λ[t] • u)› Λ.con-sym by blast
      moreover have Λ.con (λ[Λ.Src t] • Λ.Src u) (λ[t] • u)
        using ‹Λ.con (λ[Λ.Src t] • Λ.Src u) (λ[t] • u)› by blast
      moreover have Λ.con (Λ.subst u t) (Λ.subst (u \ Λ.Src u) (t \ Λ.Src t))
        by (metis Λ.Coinitial-iff-Con Λ.Ide-Src calculation(1−3) Λ.resid-Arr-Ide)
      ultimately show ?thesis
        using 1 by auto
    qed
    finally have ([λ[Λ.Src t] • Λ.Src u] @ [Λ.subst u t]) *\* [λ[t] • u] =
                  [Λ.subst (Λ.Trg u) (Λ.Trg t)] @ [Λ.subst u t] *\* [Λ.subst u t]
      by blast
    moreover have Ide ...
      by (metis 1 2 Λ.Arr.simps(5) Λ.Arr-Subst Λ.Ide-Subst Λ.Ide-Trg
          Nil-is-append-conv Arr-append-iff_PWE Con-implies-Arr(2) Ide.simps(1−2)
          Ide-appendI_PWE Resid-Arr-self ide-char calculation Λ.ide-char ind
          Con-imp-Arr-Resid)
```

```
      ultimately show ?thesis
        using ide-char by presburger
    qed
    show [λ[t] • u] *≲* [λ[Λ.Src t] • Λ.Src u] @ [Λ.subst u t]
    proof −
      have [λ[t] • u] *\* ([λ[Λ.Src t] • Λ.Src u] @ [Λ.subst u t]) =
           ([λ[t] • u] *\* [λ[Λ.Src t] • Λ.Src u]) *\* [Λ.subst u t]
        by fastforce
      also have ... = [Λ.subst u t] *\* [Λ.subst u t]
      proof −
        have t \ Λ.Src t ≠ ♯ ∧ u \ Λ.Src u ≠ ♯
          by (metis 1 Λ.Arr.simps(5) Λ.Coinitial-iff-Con Λ.Ide-Src
              Λ.Ide-iff-Src-self Λ.Ide-implies-Arr)
        moreover have Λ.con (Λ.subst u t) (Λ.subst u t)
          by (metis 1 Λ.Arr.simps(5) Λ.Arr-Subst Λ.Coinitial-iff-Con
              Λ.con-def Λ.null-char)
        moreover have Λ.con (λ[t] • u) (λ[Λ.Src t] • Λ.Src u)
          by (metis 1 Λ.Con-sym Λ.con-def Λ.head-redex.simps(9) Λ.null-char
              Λ.prfx-implies-con Λ.subs-head-redex Λ.subs-implies-prfx)
        moreover have Λ.con (Λ.subst (u \ Λ.Src u) (t \ Λ.Src t)) (Λ.subst u t)
          by (metis Λ.Coinitial-iff-Con Λ.Ide-Src calculation(1) calculation(2)
              Λ.resid-Arr-Ide)
        ultimately show ?thesis
          using Λ.resid-Arr-Ide
          apply simp
          by (metis Λ.Coinitial-iff-Con Λ.Ide-Src)
      qed
      finally have [λ[t] • u] *\* ([λ[Λ.Src t] • Λ.Src u] @ [Λ.subst u t]) =
                   [Λ.subst u t] *\* [Λ.subst u t]
        by blast
      moreover have Ide ...
        by (metis 1 2 Λ.Arr.simps(5) Λ.Arr-Subst Con-implies-Arr(2) Resid-Arr-self
            ind ide-char)
      ultimately show ?thesis
        using ide-char by presburger
    qed
  qed
  finally show ?thesis by blast
 qed
qed

lemma Src-hd-standard-development:
assumes Λ.Arr t and ¬ Λ.Ide t
shows Λ.Src (hd (standard-development t)) = Λ.Src t
  by (metis assms Src-hd-eqI cong-standard-development list.sel(1))

lemma Trg-last-standard-development:
assumes Λ.Arr t and ¬ Λ.Ide t
```

shows $\Lambda.Trg\ (last\ (standard\text{-}development\ t)) = \Lambda.Trg\ t$
by (*metis assms Trg-last-eqI cong-standard-development last-ConsL*)

lemma *Srcs-standard-development*:
shows $[\![\Lambda.Arr\ t;\ standard\text{-}development\ t \neq []]\!]$
$\Longrightarrow Srcs\ (standard\text{-}development\ t) = \{\Lambda.Src\ t\}$
by (*metis Con-implies-Arr(1) Ide.simps(1) Ide-iff-standard-development-empty*
Src-hd-standard-development Srcs-simp$_{\Lambda P}$ cong-standard-development ide-char)

lemma *Trgs-standard-development*:
shows $[\![\Lambda.Arr\ t;\ standard\text{-}development\ t \neq []]\!]$
$\Longrightarrow Trgs\ (standard\text{-}development\ t) = \{\Lambda.Trg\ t\}$
by (*metis Con-implies-Arr(2) Ide.simps(1) Ide-iff-standard-development-empty*
Trg-last-standard-development Trgs-simp$_{\Lambda P}$ cong-standard-development ide-char)

lemma *development-standard-development*:
shows $\Lambda.Arr\ t \longrightarrow development\ t\ (standard\text{-}development\ t)$
apply (*rule standard-development.induct*)
apply *blast*
apply *simp*
apply (*simp add*: *development-map-Lam*)
proof
fix *t1 t2*
assume *ind1*: $\Lambda.Arr\ t1 \wedge \Lambda.Arr\ t2$
$\Longrightarrow \Lambda.Arr\ t1 \longrightarrow development\ t1\ (standard\text{-}development\ t1)$
assume *ind2*: $\Lambda.Arr\ t1 \wedge \Lambda.Arr\ t2$
$\Longrightarrow \Lambda.Arr\ t2 \longrightarrow development\ t2\ (standard\text{-}development\ t2)$
assume *t*: $\Lambda.Arr\ (t1 \circ t2)$
show $development\ (t1 \circ t2)\ (standard\text{-}development\ (t1 \circ t2))$
proof (*cases standard-development t1 = []*)
show $standard\text{-}development\ t1 = []$
$\Longrightarrow development\ (t1 \circ t2)\ (standard\text{-}development\ (t1 \circ t2))$
using *t ind2* $\Lambda.Ide\text{-}Src\ \Lambda.Ide\text{-}Trg\ \Lambda.Ide\text{-}iff\text{-}Src\text{-}self\ \Lambda.Ide\text{-}iff\text{-}Trg\text{-}self$
Ide-iff-standard-development-empty
development-map-App-2 [*of* $\Lambda.Src\ t1\ t2\ standard\text{-}development\ t2$]
by *fastforce*
assume *t1*: $standard\text{-}development\ t1 \neq []$
show $development\ (t1 \circ t2)\ (standard\text{-}development\ (t1 \circ t2))$
proof (*cases standard-development t2 = []*)
assume *t2*: $standard\text{-}development\ t2 = []$
show *?thesis*
using *t t2 ind1 Ide-iff-standard-development-empty development-map-App-1* **by** *simp*
next
assume *t2*: $standard\text{-}development\ t2 \neq []$
have $development\ (t1 \circ t2)\ (map\ (\lambda a.\ a \circ \Lambda.Src\ t2)\ (standard\text{-}development\ t1))$
using $\Lambda.Arr.simps(4)$ *development-map-App-1 ind1 t* **by** *presburger*
moreover have $development\ ((t1 \circ t2)\ ^{1}\backslash^{*}$
$map\ (\lambda a.\ a \circ \Lambda.Src\ t2)\ (standard\text{-}development\ t1))$
$(map\ (\lambda a.\ \Lambda.Trg\ t1 \circ a)\ (standard\text{-}development\ t2))$

```
    proof −
      have Λ.App t1 t2 ¹\* map (λa. a ∘ Λ.Src t2) (standard-development t1) =
          Λ.Trg t1 ∘ t2
      proof −
        have map (λa. a ∘ Λ.Src t2) (standard-development t1) *∼* [t1 ∘ Λ.Src t2]
        proof −
          have map (λa. a ∘ Λ.Src t2) (standard-development t1) =
              standard-development (t1 ∘ Λ.Src t2)
            by (metis Λ.Arr.simps(4) Λ.Ide-Src Λ.Ide-iff-Src-self
              Ide-iff-standard-development-empty Λ.Ide-implies-Arr Nil-is-map-conv
              append-Nil2 standard-development.simps(4) t)
          also have standard-development (t1 ∘ Λ.Src t2) *∼* [t1 ∘ Λ.Src t2]
            by (metis Λ.Arr.simps(4) Λ.Ide.simps(4) Λ.Ide-Src Λ.Ide-implies-Arr
              cong-standard-development development-Ide ind1 t t1)
          finally show ?thesis by blast
        qed
        hence [t1 ∘ t2] *\* map (λa. a ∘ Λ.Src t2) (standard-development t1) =
            [t1 ∘ t2] *\* [t1 ∘ Λ.Src t2]
          by (metis Resid-parallel con-imp-coinitial prfx-implies-con calculation
            development-implies map-is-Nil-conv t1)
        also have [t1 ∘ t2] *\* [t1 ∘ Λ.Src t2] = [Λ.Trg t1 ∘ t2]
          using t Λ.arr-resid-iff-con Λ.resid-Arr-self
          by simp force
        finally have [t1 ∘ t2] *\* map (λa. a ∘ Λ.Src t2) (standard-development t1) =
                [Λ.Trg t1 ∘ t2]
          by blast
        thus ?thesis
          by (simp add: Resid1x-as-Resid')
      qed
      thus ?thesis
        by (metis ind2 Λ.Arr.simps(4) Λ.Ide-Trg Λ.Ide-iff-Src-self development-map-App-2
          Λ.reduction-strategy-def Λ.head-strategy-is-reduction-strategy t)
    qed
    ultimately show ?thesis
      using t development-append [of t1 ∘ t2
                                  map (λa. a ∘ Λ.Src t2) (standard-development t1)
                                  map (λb. Λ.Trg t1 ∘ b) (standard-development t2)]
      by auto
  qed
qed
next
fix t1 t2
assume ind: Λ.Arr t1 ∧ Λ.Arr t2 ⟹
            Λ.Arr (Λ.subst t2 t1)
              ⟶ development (Λ.subst t2 t1) (standard-development (Λ.subst t2 t1))
show Λ.Arr (λ[t1] • t2) ⟶ development (λ[t1] • t2) (standard-development (λ[t1] • t2))
proof
  assume 1: Λ.Arr (λ[t1] • t2)
  have development (Λ.subst t2 t1) (standard-development (Λ.subst t2 t1))
```

```
      using 1 ind by (simp add: Λ.Arr-Subst)
    thus development (λ[t1] • t2) (standard-development (λ[t1] • t2))
      using 1 Λ.Ide-Src Λ.subs-Ide by auto
  qed
qed

lemma Std-standard-development:
shows Std (standard-development t)
  apply (rule standard-development.induct)
     apply simp-all
  using Std-map-Lam
    apply blast
proof
  fix t u
  assume t: Λ.Arr t ∧ Λ.Arr u ⟹ Std (standard-development t)
  assume u: Λ.Arr t ∧ Λ.Arr u ⟹ Std (standard-development u)
  assume 0: Λ.Arr t ∧ Λ.Arr u
  show Std (map (λa. a ∘ Λ.Src u) (standard-development t) @
            map (λb. Λ.Trg t ∘ b) (standard-development u))
  proof (cases Λ.Ide t)
    show Λ.Ide t ⟹ ?thesis
      using 0 Λ.Ide-iff-Trg-self Ide-iff-standard-development-empty u Std-map-App2
      by fastforce
    assume 1: ¬ Λ.Ide t
    show ?thesis
    proof (cases Λ.Ide u)
      show Λ.Ide u ⟹ ?thesis
        using t u 0 1 Std-map-App1 [of Λ.Src u standard-development t] Λ.Ide-Src
        by (metis Ide-iff-standard-development-empty append-Nil2 list.simps(8))
      assume 2: ¬ Λ.Ide u
      show ?thesis
      proof (intro Std-append)
        show 3: Std (map (λa. a ∘ Λ.Src u) (standard-development t))
          using t 0 Std-map-App1 Λ.Ide-Src by blast
        show Std (map (λb. Λ.Trg t ∘ b) (standard-development u))
          using u 0 Std-map-App2 Λ.Ide-Trg by simp
        show map (λa. a ∘ Λ.Src u) (standard-development t) = [] ∨
             map (λb. Λ.Trg t ∘ b) (standard-development u) = [] ∨
             Λ.sseq (last (map (λa. a ∘ Λ.Src u) (standard-development t)))
                    (hd (map (λb. Λ.Trg t ∘ b) (standard-development u)))
        proof −
          have Λ.sseq (last (map (λa. a ∘ Λ.Src u) (standard-development t)))
                      (hd (map (λb. Λ.Trg t ∘ b) (standard-development u)))
          proof −
            obtain x where x: last (map (λa. a ∘ Λ.Src u) (standard-development t)) =
                              x ∘ Λ.Src u
              using 0 1 Ide-iff-standard-development-empty last-map by auto
            obtain y where y: hd (map (λb. Λ.Trg t ∘ b) (standard-development u)) =
                              Λ.Trg t ∘ y
```

```
      using 0 2 Ide-iff-standard-development-empty list.map-sel(1) by auto
    have Λ.elementary-reduction x
    proof −
      have Λ.elementary-reduction (x ∘ Λ.Src u)
        using x
      by (metis 0 1 3 Ide-iff-standard-development-empty Nil-is-map-conv Std.simps(2)
          Std-imp-sseq-last-hd append-butlast-last-id append-self-conv2 list.discI
          list.sel(1) Λ.sseq-imp-elementary-reduction2)
      thus ?thesis
        using 0 Λ.Ide-Src Λ.elementary-reduction-not-ide by auto
    qed
    moreover have Λ.elementary-reduction y
    proof −
      have Λ.elementary-reduction (Λ.Trg t ∘ y)
        using y
        by (metis 0 2 Λ.Ide-Trg Ide-iff-standard-development-empty
            u Std.elims(2) Λ.elementary-reduction.simps(4) list.map-sel(1) list.sel(1)
            Λ.sseq-imp-elementary-reduction1)
      thus ?thesis
        using 0 Λ.Ide-Trg Λ.elementary-reduction-not-ide by auto
    qed
    moreover have Λ.Trg t = Λ.Trg x
      by (metis 0 1 Ide-iff-standard-development-empty Trg-last-standard-development
          x Λ.lambda.inject(3) last-map)
    moreover have Λ.Src u = Λ.Src y
      using y
      by (metis 0 2 Λ.Arr-not-Nil Λ.Coinitial-iff-Con
          Ide-iff-standard-development-empty development.elims(2) development-imp-Arr
          development-standard-development Λ.lambda.inject(3) list.map-sel(1)
          list.sel(1))
    ultimately show ?thesis
      using x y by simp
  qed
  thus ?thesis by blast
 qed
 qed
qed
qed
next
fix t u
assume ind: Λ.Arr t ∧ Λ.Arr u ⟹ Std (standard-development (Λ.subst u t))
show Λ.Arr t ∧ Λ.Arr u
      ⟶ Std ((λ[Λ.Src t] • Λ.Src u) # standard-development (Λ.subst u t))
proof
  assume 1: Λ.Arr t ∧ Λ.Arr u
  show Std ((λ[Λ.Src t] • Λ.Src u) # standard-development (Λ.subst u t))
  proof (cases Λ.Ide (Λ.subst u t))
    show Λ.Ide (Λ.subst u t)
          ⟹ Std ((λ[Λ.Src t] • Λ.Src u) # standard-development (Λ.subst u t))
```

```
    using 1 Λ.Arr-Subst Λ.Ide-Src Ide-iff-standard-development-empty by simp
  assume 2: ¬ Λ.Ide (Λ.subst u t)
  show Std ((λ[Λ.Src t] • Λ.Src u) # standard-development (Λ.subst u t))
  proof −
    have Λ.sseq (λ[Λ.Src t] • Λ.Src u) (hd (standard-development (Λ.subst u t)))
    proof −
      have Λ.elementary-reduction (hd (standard-development (Λ.subst u t)))
        using ind
        by (metis 1 2 Λ.Arr-Subst Ide-iff-standard-development-empty
            Std.elims(2) list.sel(1) Λ.sseq-imp-elementary-reduction1)
      moreover have Λ.seq (λ[Λ.Src t] • Λ.Src u)
                          (hd (standard-development (Λ.subst u t)))
        using 1 2 Src-hd-standard-development calculation Λ.Arr.simps(5)
              Λ.Arr-Src Λ.Arr-Subst Λ.Src-Subst Λ.Trg.simps(4) Λ.Trg-Src Λ.arr-char
              Λ.elementary-reduction-is-arr Λ.seq-char
        by presburger
      ultimately show ?thesis
        using 1 Λ.Ide-Src Λ.sseq-Beta by auto
    qed
    moreover have Std (standard-development (Λ.subst u t))
      using 1 ind by blast
    ultimately show ?thesis
      by (metis 1 2 Λ.Arr-Subst Ide-iff-standard-development-empty Std.simps(3)
          list.collapse)
  qed
 qed
qed
qed
```

3.6.3 Standardization

In this section, we define and prove correct a function that takes an arbitrary reduction path and produces a standard reduction path congruent to it. The method is roughly analogous to insertion sort: given a path, recursively standardize the tail and then "insert" the head into to the result. A complication is that in general the head may be a parallel reduction instead of an elementary reduction, and in any case elementary reductions are not preserved under residuation so we need to be able to handle the parallel reductions that arise from permuting elementary reductions. In general, this means that parallel reduction steps have to be decomposed into factors, and then each factor has to be inserted at its proper position. Another issue is that reductions don't all happen at the top level of a term, so we need to be able to descend recursively into terms during the insertion procedure. The key idea here is: in a standard reduction, once a step has occurred that is not a head reduction, then all subsequent terms will have *App* as their top-level constructor. So, once we have passed a step that is not a head reduction, we can recursively descend into the subsequent applications and treat the "rator" and the "rand" parts independently.

The following function performs the core insertion part of the standardization al-

gorithm. It assumes that it is given an arbitrary parallel reduction t and an already-standard reduction path U, and it inserts t into U, producing a standard reduction path that is congruent to $t \mathbin{\#} U$. 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 $t \mathbin{\#} U$ consists completely of identities, then its standardization is the empty list $[]$, which is not a path and cannot be congruent to $t \mathbin{\#} U$. So in order to be able to apply the induction hypotheses in the correctness proof, we need to make sure that we don't make recursive calls when U itself would consist entirely of identities. Finally, when we descend through an application, the step t and the path U are projected to their "rator" and "rand" components, which are treated separately and the results concatenated. However, the projection operations can introduce identities and therefore do not preserve elementary reductions. To handle this, we need to filter out identities after projection but before the recursive call.

Ensuring termination also involves some care: we make recursive calls in which the length of the second argument is increased, but the "height" of the first argument is decreased. So we use a lexicographic order that makes the height of the first argument more significant and the length of the second argument secondary. The base cases either discard paths that consist entirely of identities, or else they expand a single parallel reduction t into a standard development.

```
function (sequential) stdz-insert
where stdz-insert t [] = standard-development t
  | stdz-insert «-» U = stdz-insert (hd U) (tl U)
  | stdz-insert λ[t] U =
    (if Λ.Ide t then
       stdz-insert (hd U) (tl U)
     else
       map Λ.Lam (stdz-insert t (map Λ.un-Lam U)))
  | stdz-insert (λ[t] ∘ u) ((λ[-] • -) # U) = stdz-insert (λ[t] • u) U
  | stdz-insert (t ∘ u) U =
    (if Λ.Ide (t ∘ u) then
       stdz-insert (hd U) (tl U)
     else if Λ.seq (t ∘ u) (hd U) then
       if Λ.contains-head-reduction (t ∘ u) then
         if Λ.Ide ((t ∘ u) \ Λ.head-redex (t ∘ u)) then
           Λ.head-redex (t ∘ u) # stdz-insert (hd U) (tl U)
         else
           Λ.head-redex (t ∘ u) # stdz-insert ((t ∘ u) \ Λ.head-redex (t ∘ u)) U
       else if Λ.contains-head-reduction (hd U) then
         if Λ.Ide ((t ∘ u) \ Λ.head-strategy (t ∘ u)) then
           stdz-insert (Λ.head-strategy (t ∘ u)) (tl U)
         else
```

```
            Λ.head-strategy (t ∘ u) # stdz-insert ((t ∘ u) \ Λ.head-strategy (t ∘ u)) (tl U)
          else
            map (λa. a ∘ Λ.Src u)
              (stdz-insert t (filter notIde (map Λ.un-App1 U))) @
            map (λb. Λ.Trg (Λ.un-App1 (last U)) ∘ b)
              (stdz-insert u (filter notIde (map Λ.un-App2 U)))
        else [])
    | stdz-insert (λ[t] • u) U =
      (if Λ.Arr t ∧ Λ.Arr u then
        (λ[Λ.Src t] • Λ.Src u) # stdz-insert (Λ.subst u t) U
      else [])
    | stdz-insert - - = []
  by pat-completeness auto



  fun standardize
  where standardize [] = []
    | standardize U = stdz-insert (hd U) (standardize (tl U))

  abbreviation stdzins-rel
   where stdzins-rel ≡ mlex-prod (length o snd) (inv-image (mlex-prod Λ.hgt Λ.subterm-rel)
fst)

  termination stdz-insert
  using Λ.subterm.intros(2−3) Λ.hgt-Subst less-Suc-eq-le Λ.elementary-reduction-decreases-hgt
        Λ.elementary-reduction-head-redex Λ.contains-head-reduction-iff
        Λ.elementary-reduction-is-arr Λ.Src-head-redex Λ.App-Var-contains-no-head-reduction
        Λ.hgt-resid-App-head-redex Λ.seq-char
    apply (relation stdzins-rel)
    apply (auto simp add: wf-mlex Λ.wf-subterm-rel mlex-iff mlex-less Λ.subterm-lemmas(1))
    by (meson dual-order.eq-iff length-filter-le not-less-eq-eq)+

  lemma stdz-insert-Ide:
  shows Ide (t # U) ⟹ stdz-insert t U = []
  proof (induct U arbitrary: t)
    show ⋀t. Ide [t] ⟹ stdz-insert t [] = []
      by (metis Ide-iff-standard-development-empty Λ.Ide-implies-Arr Ide.simps(2)
          Λ.ide-char stdz-insert.simps(1))
    show ⋀U. ⟦⋀t. Ide (t # U) ⟹ stdz-insert t U = []; Ide (t # u # U)⟧
                  ⟹ stdz-insert t (u # U) = []
      for t u
      using Λ.ide-char
      apply (cases t; cases u)
        apply simp-all
      by fastforce
  qed

  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 (*induct U arbitrary*: *u*)
show $\bigwedge u.\ [\![\Lambda.Ide\ u;\ seq\ [u]\ [];\ Std\ []]\!] \Longrightarrow stdz\text{-}insert\ u\ [] = stdz\text{-}insert\ (hd\ [])\ (tl\ [])$
by (*simp add*: *seq-char*)
fix *u v U*
assume *u*: $\Lambda.Ide\ u$
assume *seq*: $seq\ [u]\ (v\ \#\ U)$
assume *Std*: $Std\ (v\ \#\ U)$
assume *ind*: $\bigwedge u.\ [\![\Lambda.Ide\ u;\ seq\ [u]\ U;\ Std\ U]\!]$
$\Longrightarrow stdz\text{-}insert\ u\ U = stdz\text{-}insert\ (hd\ U)\ (tl\ U)$
show $stdz\text{-}insert\ u\ (v\ \#\ U) = stdz\text{-}insert\ (hd\ (v\ \#\ U))\ (tl\ (v\ \#\ U))$
using *u ind stdz-insert-Ide Ide-implies-Arr*
apply (*cases u*; *cases v*)
apply *simp-all*
proof −
fix *x y a b*
assume *xy*: $\Lambda.Ide\ x \wedge \Lambda.Ide\ y$
assume *u'*: $u = x \circ y$
assume *v'*: $v = \boldsymbol{\lambda}[a] \bullet b$
have *ab*: $\Lambda.Ide\ a \wedge \Lambda.Ide\ b$
using $Std\ \langle v = \boldsymbol{\lambda}[a] \bullet b\rangle$ *Std.elims*(*2*) *Λ.sseq-Beta*
by (*metis Std-consE Λ.elementary-reduction.simps*(*5*) *Std.simps*(*2*))
have $x = \boldsymbol{\lambda}[a] \wedge y = b$
using *xy ab u u' v' seq seq-char*
by (*metis Λ.Ide-iff-Src-self Λ.Ide-iff-Trg-self Λ.Ide-implies-Arr Λ.Src.simps*(*5*)
Srcs-simp$_{\Lambda P}$ *Trgs.simps*(*2*) *Λ.lambda.inject*(*3*) *list.sel*(*1*) *singleton-insert-inj-eq*
Λ.targets-char$_{\Lambda}$)
thus $stdz\text{-}insert\ (x \circ y)\ ((\boldsymbol{\lambda}[a] \bullet b)\ \#\ U) = stdz\text{-}insert\ (\boldsymbol{\lambda}[a] \bullet b)\ U$
using *u u' stdz-insert.simps*(*4*) **by** *presburger*
qed
qed

Insertion of a term with *Beta* as its top-level constructor always leaves such a term at the head of the result. Stated another way, *Beta* at the top-level must always come first in a standard reduction path.

lemma *stdz-insert-Beta-ind*:
shows $[\![\Lambda.hgt\ t + length\ U \leq n;\ \Lambda.is\text{-}Beta\ t;\ seq\ [t]\ U]\!]$
$\Longrightarrow \Lambda.is\text{-}Beta\ (hd\ (stdz\text{-}insert\ t\ U))$
proof (*induct n arbitrary*: *t U*)
show $\bigwedge t\ U.\ [\![\Lambda.hgt\ t + length\ U \leq 0;\ \Lambda.is\text{-}Beta\ t;\ seq\ [t]\ U]\!]$
$\Longrightarrow \Lambda.is\text{-}Beta\ (hd\ (stdz\text{-}insert\ t\ U))$
using *Arr.simps*(*1*) *seq-char* **by** *blast*
fix *n t U*
assume *ind*: $\bigwedge t\ U.\ [\![\Lambda.hgt\ t + length\ U \leq n;\ \Lambda.is\text{-}Beta\ t;\ seq\ [t]\ U]\!]$
$\Longrightarrow \Lambda.is\text{-}Beta\ (hd\ (stdz\text{-}insert\ t\ U))$
assume *seq*: $seq\ [t]\ U$
assume *n*: $\Lambda.hgt\ t + length\ U \leq Suc\ n$
assume *t*: $\Lambda.is\text{-}Beta\ t$
show $\Lambda.is\text{-}Beta\ (hd\ (stdz\text{-}insert\ t\ U))$

```
    using t seq seq-char
    by (cases U; cases t; cases hd U) auto
  qed

  lemma stdz-insert-Beta:
  assumes Λ.is-Beta t and seq [t] U
  shows Λ.is-Beta (hd (stdz-insert t U))
    using assms stdz-insert-Beta-ind by blast
```

This is the correctness lemma for insertion: Given a term t and standard reduction path U sequential with it, the result of insertion is a standard reduction path which is congruent to $t \# U$ unless $t \# U$ consists entirely of identities.

The proof is very long. Its structure parallels that of the definition of the function *stdz-insert*. For really understanding the details, I strongly suggest viewing the proof in Isabelle/JEdit and using the code folding feature to unfold the proof a little bit at a time.

```
  lemma stdz-insert-correctness:
  shows seq [t] U ∧ Std U ⟶
          Std (stdz-insert t U) ∧ (¬ Ide (t # U) ⟶ cong (stdz-insert t U) (t # U))
        (is ?P t U)
  proof (rule stdz-insert.induct [of ?P])
    show ⋀t. ?P t []
      using seq-char by simp
    show ⋀u U. ?P ♯ (u # U)
      using seq-char not-arr-null null-char by auto
    show ⋀x u U. ?P (hd (u # U)) (tl (u # U)) ⟹ ?P «x» (u # U)
    proof −
      fix x u U
      assume ind: ?P (hd (u # U)) (tl (u # U))
      show ?P «x» (u # U)
      proof (intro impI, elim conjE, intro conjI)
        assume seq: seq [«x»] (u # U)
        assume Std: Std (u # U)
        have 1: stdz-insert «x» (u # U) = stdz-insert u U
          by simp
        have 2: U ≠ [] ⟹ seq [u] U
          using Std Std-imp-Arr
          by (simp add: arrI_P arr-append-imp-seq)
        show Std (stdz-insert «x» (u # U))
          using ind
          by (metis 1 2 Std Std-standard-development list.exhaust-sel list.sel(1) list.sel(3)
              reduction-paths.Std.simps(3) reduction-paths.stdz-insert.simps(1))
        show ¬ Ide («x» # u # U) ⟶ stdz-insert «x» (u # U) *∼* «x» # u # U
        proof (cases U = [])
          show U = [] ⟹ ?thesis
            using cong-standard-development cong-cons-ideI(1)
            apply simp
            by (metis Arr.simps(1−2) Arr-iff-Con-self Con-rec(3) Λ.in-sourcesI con-char
                cong-transitive ideE Λ.Ide.simps(2) Λ.arr-char Λ.ide-char seq)
```

```
      assume U: U ≠ []
      show ?thesis
        using 1 2 ind seq seq-char cong-cons-ideI(1)
        apply simp
        by (metis Std Std-consE U Λ.Arr.simps(2) Λ.Ide.simps(2) Λ.targets-simps(2)
            prfx-transitive)
    qed
  qed
qed
show ⋀M u U. ⟦Λ.Ide M ⟹ ?P (hd (u # U)) (tl (u # U));
              ¬ Λ.Ide M ⟹ ?P M (map Λ.un-Lam (u # U))⟧
                 ⟹ ?P λ[M] (u # U)
proof −
  fix M u U
  assume ind1: Λ.Ide M ⟹ ?P (hd (u # U)) (tl (u # U))
  assume ind2: ¬ Λ.Ide M ⟹ ?P M (map Λ.un-Lam (u # U))
  show ?P λ[M] (u # U)
  proof (intro impI, elim conjE)
    assume seq: seq [λ[M]] (u # U)
    assume Std: Std (u # U)
    have u: Λ.is-Lam u
      using seq
      by (metis insert-subset Λ.lambda.disc(8) list.simps(15) mem-Collect-eq
          seq-Lam-Arr-implies)
    have U: set U ⊆ Collect Λ.is-Lam
      using u seq
      by (metis insert-subset Λ.lambda.disc(8) list.simps(15) seq-Lam-Arr-implies)
    show Std (stdz-insert λ[M] (u # U)) ∧
          (¬ Ide (λ[M] # u # U) ⟶ stdz-insert λ[M] (u # U) *∼* λ[M] # u # U)
    proof (cases Λ.Ide M)
      assume M: Λ.Ide M
      have 1: stdz-insert λ[M] (u # U) = stdz-insert u U
        using M by simp
      show ?thesis
      proof (cases Ide (u # U))
        show Ide (u # U) ⟹ ?thesis
          using 1 Std-standard-development Ide-iff-standard-development-empty
          by (metis Ide-imp-Ide-hd Std Std-implies-set-subset-elementary-reduction
              Λ.elementary-reduction-not-ide list.sel(1) list.set-intros(1)
              mem-Collect-eq subset-code(1))
        assume 2: ¬ Ide (u # U)
        show ?thesis
        proof (cases U = [])
          assume 3: U = []
          have 4: standard-development u *∼* [λ[M]] @ [u]
            using M 2 3 seq ide-char cong-standard-development [of u]
                  cong-append-ideI(1) [of [λ[M]] [u]]
            by (metis Arr-imp-arr-hd Ide.simps(2) Std Std-imp-Arr cong-transitive
                Λ.Ide.simps(3) Λ.arr-char Λ.ide-char list.sel(1) not-Cons-self2)
```

```
    show ?thesis
      using 1 3 4 Std-standard-development by force
    next
    assume 3: U ≠ []
    have stdz-insert λ[M] (u # U) = stdz-insert u U
      using M 3 by simp
    have 5: Λ.Arr u ∧ ¬ Λ.Ide u
      by (meson 3 Std Std-consE Λ.elementary-reduction-not-ide Λ.ide-char
          Λ.sseq-imp-elementary-reduction1)
    have 4: standard-development u @ U *∼* ([λ[M]] @ [u]) @ U
    proof (intro cong-append seqI_ΛP)
      show Arr (standard-development u)
     using 5 Std-standard-development Std-imp-Arr Ide-iff-standard-development-empty
        by force
      show Arr U
        using Std 3 by auto
      show Λ.Trg (last (standard-development u)) = Λ.Src (hd U)
        by (metis 3 5 Std Std-consE Trg-last-standard-development Λ.seq-char
            Λ.sseq-imp-seq)
      show standard-development u *∼* [λ[M]] @ [u]
        using M 5 Std Std-imp-Arr cong-standard-development [of u]
              cong-append-ideI(3) [of [λ[M]] [u]]
        by (metis (no-types, lifting) Arr.simps(2) Ide.simps(2) arr-char ide-char
            Λ.Ide.simps(3) Λ.arr-char Λ.ide-char prfx-transitive seq seq-def
            sources-cons)
      show U *∼* U
        by (simp add: ‹Arr U› arr-char prfx-reflexive)
    qed
    show ?thesis
    proof (intro conjI)
      show Std (stdz-insert λ[M] (u # U))
        by (metis (no-types, lifting) 1 3 M Std Std-consE append-Cons
            append-eq-append-conv2 append-self-conv arr-append-imp-seq ind1
            list.sel(1) list.sel(3) not-Cons-self2 seq seq-def)
      show ¬ Ide (λ[M] # u # U) ⟶ stdz-insert λ[M] (u # U) *∼* λ[M] # u # U
      proof
        have seq [u] U ∧ Std U
          using 2 3 Std
          by (metis Cons-eq-appendI Std-consE arr-append-imp-seq neq-Nil-conv
              self-append-conv2 seq seqE)
        thus stdz-insert λ[M] (u # U) *∼* λ[M] # u # U
          using M 1 2 3 4 ind1 cong-cons-ideI(1) [of λ[M] u # U]
          apply simp
          by (meson cong-transitive seq)
      qed
    qed
  qed
qed
next
```

```
      assume M: ¬ Λ.Ide M
      have 1: stdz-insert λ[M] (u # U) =
              map Λ.Lam (stdz-insert M (Λ.un-Lam u # map Λ.un-Lam U))
        using M by simp
      show ?thesis
      proof (intro conjI)
        show Std (stdz-insert λ[M] (u # U))
          by (metis 1 M Std Std-map-Lam Std-map-un-Lam ind2 Λ.lambda.disc(8)
              list.simps(9) seq seq-Lam-Arr-implies seq-map-un-Lam)
        show ¬ Ide (λ[M] # u # U) ⟶ stdz-insert λ[M] (u # U) *∼* λ[M] # u # U
        proof
          have map Λ.Lam (stdz-insert M (Λ.un-Lam u # map Λ.un-Lam U)) *∼*
               λ[M] # u # U
          proof −
            have map Λ.Lam (stdz-insert M (Λ.un-Lam u # map Λ.un-Lam U)) *∼*
                 map Λ.Lam (M # Λ.un-Lam u # map Λ.un-Lam U)
              by (metis (mono-tags, opaque-lifting) Ide-imp-Ide-hd M Std Std-map-un-Lam
                  cong-map-Lam ind2 Λ.ide-char Λ.lambda.discI(2)
                  list.sel(1) list.simps(9) seq seq-Lam-Arr-implies seq-map-un-Lam)
            thus ?thesis
              using u U
              by (simp add: map-idI subset-code(1))
          qed
          thus stdz-insert λ[M] (u # U) *∼* λ[M] # u # U
            using 1 by presburger
        qed
      qed
    qed
  qed
qed
show ⋀M N A B U. ?P (λ[M] • N) U ⟹ ?P (λ[M] ∘ N) ((λ[A] • B) # U)
proof −
  fix M N A B U
  assume ind: ?P (λ[M] • N) U
  show ?P (λ[M] ∘ N) ((λ[A] • B) # U)
  proof (intro impI, elim conjE)
    assume seq: seq [λ[M] ∘ N] ((λ[A] • B) # U)
    assume Std: Std ((λ[A] • B) # U)
    have MN: Λ.Arr M ∧ Λ.Arr N
      using seq
      by (simp add: seq-char)
    have AB: Λ.Trg M = A ∧ Λ.Trg N = B
    proof −
      have 1: Λ.Ide A ∧ Λ.Ide B
        using Std
        by (metis Std.simps(2) Std.simps(3) Λ.elementary-reduction.simps(5)
            list.exhaust-sel Λ.sseq-Beta)
      moreover have Trgs [λ[M] ∘ N] = Srcs [λ[A] • B]
        using 1 seq seq-char
```

```
    by (simp add: Λ.Ide-implies-Arr Srcs-simpΛP)
  ultimately show ?thesis
    by (metis Λ.Ide-iff-Src-self Λ.Ide-implies-Arr Λ.Src.simps(5) Srcs-simpΛP
        Λ.Trg.simps(2−3) Trgs-simpΛP Λ.lambda.inject(2) Λ.lambda.sel(3−4)
        last.simps list.sel(1) seq-char seq the-elem-eq)
qed
have 1: stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U) = stdz-insert (λ[M] • N) U
  by auto
show Std (stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U)) ∧
     (¬ Ide ((λ[M] ∘ N) # (λ[A] • B) # U) ⟶
       stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U) *∼* (λ[M] ∘ N) # (λ[A] • B) # U)
proof (cases U = [])
  assume U: U = []
  have 1: stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U) =
          standard-development (λ[M] • N)
    using U by simp
  show ?thesis
  proof (intro conjI)
    show Std (stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U))
      using 1 Std-standard-development by presburger
    show ¬ Ide ((λ[M] ∘ N) # (λ[A] • B) # U) ⟶
          stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U) *∼* (λ[M] ∘ N) # (λ[A] • B) # U
    proof (intro impI)
      assume 2: ¬ Ide ((λ[M] ∘ N) # (λ[A] • B) # U)
      have stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U) =
           (λ[Λ.Src M] • Λ.Src N) # standard-development (Λ.subst N M)
        using 1 MN by simp
      also have ... *∼* [λ[M] • N]
        using MN AB cong-standard-development
        by (metis 1 calculation Λ.Arr.simps(5) Λ.Ide.simps(5))
      also have [λ[M] • N] *∼* (λ[M] ∘ N) # (λ[A] • B) # U
        using AB MN U Beta-decomp(2) [of M N] by simp
      finally show stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U) *∼*
                   (λ[M] ∘ N) # (λ[A] • B) # U
        by blast
    qed
  qed
  next
  assume U: U ≠ []
  have 1: stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U) = stdz-insert (λ[M] • N) U
    using U by simp
  have 2: seq [λ[M] • N] U
    using MN AB U Std Λ.sseq-imp-seq
    apply (intro seqIΛP)
     apply auto
    by fastforce
  have 3: Std U
    using Std by fastforce
  show ?thesis
```

```
      proof (intro conjI)
        show Std (stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U))
          using 2 3 ind by simp
        show ¬ Ide ((λ[M] ∘ N) # (λ[A] • B) # U) ⟶
              stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U) *∼* (λ[M] ∘ N) # (λ[A] • B) # U
        proof
          have stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U) *∼* [λ[M] • N] @ U
            by (metis 1 2 3 Λ.Ide.simps(5) U Ide.simps(3) append.left-neutral
                append-Cons Λ.ide-char ind list.exhaust)
          also have [λ[M] • N] @ U *∼* ([λ[M] ∘ N] @ [λ[A] • B]) @ U
            using MN AB Beta-decomp
            by (meson 2 cong-append cong-reflexive seqE)
          also have ([λ[M] ∘ N] @ [λ[A] • B]) @ U = (λ[M] ∘ N) # (λ[A] • B) # U
            by simp
          finally show stdz-insert (λ[M] ∘ N) ((λ[A] • B) # U) *∼*
                       (λ[M] ∘ N) # (λ[A] • B) # U
            by argo
        qed
      qed
    qed
  qed
qed
show ⋀M N u U. (Λ.Arr M ∧ Λ.Arr N ⟹ ?P (Λ.subst N M) (u # U))
                ⟹ ?P (λ[M] • N) (u # U)
proof −
  fix M N u U
  assume ind: Λ.Arr M ∧ Λ.Arr N ⟹ ?P (Λ.subst N M) (u # U)
  show ?P (λ[M] • N) (u # U)
  proof (intro impI, elim conjE)
    assume seq: seq [λ[M] • N] (u # U)
    assume Std: Std (u # U)
    have MN: Λ.Arr M ∧ Λ.Arr N
      using seq seq-char by simp
    show Std (stdz-insert (λ[M] • N) (u # U)) ∧
         (¬ Ide (Λ.Beta M N # u # U) ⟶
            cong (stdz-insert (λ[M] • N) (u # U)) ((λ[M] • N) # u # U))
    proof (cases Λ.Ide (Λ.subst N M))
      assume 1: Λ.Ide (Λ.subst N M)
      have 2: ¬ Ide (u # U)
       using Std Std-implies-set-subset-elementary-reduction Λ.elementary-reduction-not-ide
        by force
      have 3: stdz-insert (λ[M] • N) (u # U) = (λ[Λ.Src M] • Λ.Src N) # stdz-insert u U
        using MN 1 seq seq-char Std stdz-insert-Ide-Std [of Λ.subst N M u # U]
              Λ.Ide-implies-Arr
        by (cases U = []) auto
      show ?thesis
      proof (cases U = [])
        assume U: U = []
        have 3: stdz-insert (λ[M] • N) (u # U) =
```

```
        (λ[Λ.Src M] • Λ.Src N) # standard-development u
  using 2 3 U by force
have 4: Λ.seq (λ[Λ.Src M] • Λ.Src N) (hd (standard-development u))
proof
  show Λ.Arr (λ[Λ.Src M] • Λ.Src N)
    using MN by simp
  show Λ.Arr (hd (standard-development u))
    by (metis 2 Arr-imp-arr-hd Ide.simps(2) Ide-iff-standard-development-empty
        Std Std-consE Std-imp-Arr Std-standard-development U Λ.arr-char
        Λ.ide-char)
  show Λ.Trg (λ[Λ.Src M] • Λ.Src N) = Λ.Src (hd (standard-development u))
    by (metis 1 2 Ide.simps(2) MN Src-hd-standard-development Std Std-consE
        Trg-last-Src-hd-eqI U Λ.Ide-iff-Src-self Λ.Ide-implies-Arr Λ.Src-Subst
        Λ.Trg.simps(4) Λ.Trg-Src Λ.Trg-Subst Λ.ide-char last-ConsL list.sel(1) seq)
qed
show ?thesis
proof (intro conjI)
  show Std (stdz-insert (λ[M] • N) (u # U))
  proof −
    have Λ.sseq (λ[Λ.Src M] • Λ.Src N) (hd (standard-development u))
      using MN 2 4 U Λ.Ide-Src
      apply (intro Λ.sseq-BetaI)
        apply auto
      by (metis Ide.simps(1) Resid.simps(2) Std Std-consE
          Std-standard-development cong-standard-development hd-Cons-tl ide-char
          Λ.sseq-imp-elementary-reduction1 Std.simps(2))
    thus ?thesis
      by (metis 3 Std.simps(2−3) Std-standard-development hd-Cons-tl
          Λ.sseq-imp-elementary-reduction1)
  qed
  show ¬ Ide ((λ[M] • N) # u # U)
           ⟶ stdz-insert (λ[M] • N) (u # U) *∼* (λ[M] • N) # u # U
  proof
    have stdz-insert (λ[M] • N) (u # U) =
         [λ[Λ.Src M] • Λ.Src N] @ standard-development u
      using 3 by simp
    also have 5: [λ[Λ.Src M] • Λ.Src N] @ standard-development u *∼*
                 [λ[Λ.Src M] • Λ.Src N] @ [u]
    proof (intro cong-append)
      show seq [λ[Λ.Src M] • Λ.Src N] (standard-development u)
        by (metis 2 3 Ide.simps(2) Ide-iff-standard-development-empty
            Std Std-consE Std-imp-Arr U ‹Std (stdz-insert (Λ.Beta M N) (u # U))›
            arr-append-imp-seq arr-char calculation Λ.ide-char neq-Nil-conv)
      thus [λ[Λ.Src M] • Λ.Src N] *∼* [λ[Λ.Src M] • Λ.Src N]
        using cong-reflexive by blast
      show standard-development u *∼* [u]
        by (metis 2 Arr.simps(2) Ide.simps(2) Std Std-imp-Arr U
            cong-standard-development Λ.arr-char Λ.ide-char not-Cons-self2)
    qed
```

```
    also have [λ[Λ.Src M] • Λ.Src N] @ [u] *∼*
              ([λ[Λ.Src M] • Λ.Src N] @ [Λ.subst N M]) @ [u]
    proof (intro cong-append)
      show seq [λ[Λ.Src M] • Λ.Src N] [u]
        by (metis 5 Con-implies-Arr(1) Ide.simps(1) arr-append-imp-seq
            arr-char ide-char not-Cons-self2)
      show [λ[Λ.Src M] • Λ.Src N] *∼* [λ[Λ.Src M] • Λ.Src N] @ [Λ.subst N M]
        by (metis (full-types) 1 MN Ide-iff-standard-development-empty
            cong-standard-development cong-transitive Λ.Arr.simps(5) Λ.Arr-Subst
            Λ.Ide.simps(5) Beta-decomp(1) standard-development.simps(5))
      show [u] *∼* [u]
        using Resid-Arr-self Std Std-imp-Arr U ide-char by blast
    qed
   also have ([λ[Λ.Src M] • Λ.Src N] @ [Λ.subst N M]) @ [u] *∼* [λ[M] • N] @ [u]
      by (metis Beta-decomp(1) MN U Resid-Arr-self cong-append
          ide-char seq-char seq)
    also have [λ[M] • N] @ [u] = (λ[M] • N) # u # U
      using U by simp
    finally show stdz-insert (λ[M] • N) (u # U) *∼* (λ[M] • N) # u # U
      by blast
  qed
qed
next
assume U: U ≠ []
have 4: seq [u] U
  by (simp add: Std U arrI_P arr-append-imp-seq)
have 5: Std U
  using Std by auto
have 6: Std (stdz-insert u U) ∧
        set (stdz-insert u U) ⊆ {a. Λ.elementary-reduction a} ∧
        (¬ Ide (u # U) ⟶
        cong (stdz-insert u U) (u # U))
proof −
  have seq [Λ.subst N M] (u # U) ∧ Std (u # U)
    using MN Std Std-imp-Arr Λ.Arr-Subst
    apply (intro conjI seqI_ΛP)
      apply simp-all
    by (metis Trg-last-Src-hd-eqI Λ.Trg.simps(4) last-ConsL list.sel(1) seq)
  thus ?thesis
    using MN 1 2 3 4 5 ind Std-implies-set-subset-elementary-reduction
          stdz-insert-Ide-Std
    apply simp
    by (meson cong-cons-ideI(1) cong-transitive lambda-calculus.ide-char)
qed
have 7: Λ.seq (λ[Λ.Src M] • Λ.Src N) (hd (stdz-insert u U))
  using MN 1 2 6 Arr-imp-arr-hd Con-implies-Arr(2) ide-char Λ.arr-char
        Ide-iff-standard-development-empty Src-hd-eqI Trg-last-Src-hd-eqI
        Trg-last-standard-development Λ.Ide-implies-Arr seq
  apply (intro Λ.seqI_Λ)
```

```
      apply simp
     apply (metis Ide.simps(1))
  by (metis Λ.Arr.simps(5) Λ.Ide.simps(5) last.simps standard-development.simps(5))
 have 8: seq [λ[Λ.Src M] • Λ.Src N] (stdz-insert u U)
  by (metis 2 6 7 seqI_ΛP Arr.simps(2) Con-implies-Arr(2)
      Ide.simps(1) ide-char last.simps Λ.seqE Λ.seq-char)
 show ?thesis
 proof (intro conjI)
  show Std (stdz-insert (λ[M] • N) (u # U))
  proof −
   have Λ.sseq (λ[Λ.Src M] • Λ.Src N) (hd (stdz-insert u U))
    by (metis MN 2 6 7 Λ.Ide-Src Std.elims(2) Ide.simps(1)
        Resid.simps(2) ide-char list.sel(1) Λ.sseq-BetaI
        Λ.sseq-imp-elementary-reduction1)
   thus ?thesis
    by (metis 2 3 6 Std.simps(3) Resid.simps(1) con-char prfx-implies-con
        list.exhaust-sel)
  qed
  show ¬ Ide ((λ[M] • N) # u # U)
         ⟶ stdz-insert (λ[M] • N) (u # U) *∼* (λ[M] • N) # u # U
  proof
   have stdz-insert (λ[M] • N) (u # U) = [λ[Λ.Src M] • Λ.Src N] @ stdz-insert u U
    using 3 by simp
   also have ... *∼* [λ[Λ.Src M] • Λ.Src N] @ u # U
    using MN 2 3 6 8 cong-append
    by (meson cong-reflexive seqE)
   also have [λ[Λ.Src M] • Λ.Src N] @ u # U *∼*
             ([λ[Λ.Src M] • Λ.Src N] @ [Λ.subst N M]) @ u # U
    using MN 1 2 6 8 Beta-decomp(1) Std Src-hd-eqI Trg-last-Src-hd-eqI
          Λ.Arr-Subst Λ.ide-char ide-char
    apply (intro cong-append cong-append-ideI seqI_ΛP)
          apply auto[2]
         apply metis
        apply auto[4]
    by (metis cong-transitive)
   also have ([λ[Λ.Src M] • Λ.Src N] @ [Λ.subst N M]) @ u # U *∼*
             [λ[M] • N] @ u # U
    by (meson MN 2 6 Beta-decomp(1) cong-append prfx-transitive seq)
   also have [λ[M] • N] @ u # U = (λ[M] • N) # u # U
    by simp
   finally show stdz-insert (λ[M] • N) (u # U) *∼* (λ[M] • N) # u # U
    by simp
  qed
 qed
qed
next
assume 1: ¬ Λ.Ide (Λ.subst N M)
have 2: stdz-insert (λ[M] • N) (u # U) =
        (λ[Λ.Src M] • Λ.Src N) # stdz-insert (Λ.subst N M) (u # U)
```

```
  using 1 MN by simp
have 3: seq [Λ.subst N M] (u # U)
  using Λ.Arr-Subst MN seq-char seq by force
have 4: Std (stdz-insert (Λ.subst N M) (u # U)) ∧
        set (stdz-insert (Λ.subst N M) (u # U)) ⊆ {a. Λ.elementary-reduction a} ∧
        stdz-insert (Λ.Subst 0 N M) (u # U) *∼* Λ.subst N M # u # U
  using 1 3 Std ind MN Ide.simps(3) Λ.ide-char
        Std-implies-set-subset-elementary-reduction
  by presburger
have 5: Λ.seq (λ[Λ.Src M] • Λ.Src N) (hd (stdz-insert (Λ.subst N M) (u # U)))
  using MN 4
  apply (intro Λ.seqI_Λ)
   apply simp
  apply (metis Arr-imp-arr-hd Con-implies-Arr(1) Ide.simps(1) ide-char Λ.arr-char)
  using Src-hd-eqI
  by force
show ?thesis
proof (intro conjI)
  show Std (stdz-insert (λ[M] • N) (u # U))
  proof −
    have Λ.sseq (λ[Λ.Src M] • Λ.Src N) (hd (stdz-insert (Λ.subst N M) (u # U)))
      using 5
      by (metis 4 MN Λ.Ide-Src Std.elims(2) Ide.simps(1) Resid.simps(2)
          ide-char list.sel(1) Λ.sseq-BetaI Λ.sseq-imp-elementary-reduction1)
    thus ?thesis
      by (metis 2 4 Std.simps(3) Arr.simps(1) Con-implies-Arr(2)
          Ide.simps(1) ide-char list.exhaust-sel)
  qed
  show ¬ Ide ((λ[M] • N) # u # U)
          ⟶ stdz-insert (λ[M] • N) (u # U) *∼* (λ[M] • N) # u # U
  proof
    have stdz-insert (λ[M] • N) (u # U) =
         [λ[Λ.Src M] • Λ.Src N] @ stdz-insert (Λ.subst N M) (u # U)
      using 2 by simp
    also have ... *∼* [λ[Λ.Src M] • Λ.Src N] @ Λ.subst N M # u # U
    proof (intro cong-append)
      show seq [λ[Λ.Src M] • Λ.Src N] (stdz-insert (Λ.subst N M) (u # U))
        by (metis 4 5 Arr.simps(2) Con-implies-Arr(1) Ide.simps(1) ide-char
            Λ.arr-char Λ.seq-char last-ConsL seqI_Λ P)
      show [λ[Λ.Src M] • Λ.Src N] *∼* [λ[Λ.Src M] • Λ.Src N]
        by (meson MN cong-transitive Λ.Arr-Src Beta-decomp(1))
      show stdz-insert (Λ.subst N M) (u # U) *∼* Λ.subst N M # u # U
        using 4 by fastforce
    qed
    also have [λ[Λ.Src M] • Λ.Src N] @ Λ.subst N M # u # U =
              ([λ[Λ.Src M] • Λ.Src N] @ [Λ.subst N M]) @ u # U
      by simp
    also have ... *∼* [λ[M] • N] @ u # U
      by (meson Beta-decomp(1) MN cong-append cong-reflexive seqE seq)
```

```
      also have [λ[M] • N] @ u # U = (λ[M] • N) # u # U
        by simp
      finally show stdz-insert (λ[M] • N) (u # U) *∼* (λ[M] • N) # u # U
        by blast
    qed
  qed
 qed
qed
qed
```

Because of the way the function package processes the pattern matching in the definition of *stdz-insert*, it produces eight separate subgoals for the remainder of the proof, even though these subgoals are all simple consequences of a single, more general fact. We first prove this fact, then use it to discharge the eight subgoals.

```
have *: ⋀M N u U.
        ⟦¬ (Λ.is-Lam M ∧ Λ.is-Beta u);
         Λ.Ide (M ∘ N) ⟹ ?P (hd (u # U)) (tl (u # U));
         ⟦¬ Λ.Ide (M ∘ N);
          Λ.seq (M ∘ N) (hd (u # U));
          Λ.contains-head-reduction (M ∘ N);
          Λ.Ide (Λ.resid (M ∘ N) (Λ.head-redex (M ∘ N)))⟧
            ⟹ ?P (hd (u # U)) (tl (u # U));
         ⟦¬ Λ.Ide (M ∘ N);
          Λ.seq (M ∘ N) (hd (u # U));
          Λ.contains-head-reduction (M ∘ N);
          ¬ Λ.Ide (Λ.resid (M ∘ N) (Λ.head-redex (M ∘ N)))⟧
            ⟹ ?P (Λ.resid (M ∘ N) (Λ.head-redex (M ∘ N))) (u # U);
         ⟦¬ Λ.Ide (M ∘ N);
          Λ.seq (M ∘ N) (hd (u # U));
          ¬ Λ.contains-head-reduction (M ∘ N);
          Λ.contains-head-reduction (hd (u # U));
          Λ.Ide (Λ.resid (M ∘ N) (Λ.head-strategy (M ∘ N)))⟧
            ⟹ ?P (Λ.head-strategy (M ∘ N)) (tl (u # U));
         ⟦¬ Λ.Ide (M ∘ N);
          Λ.seq (M ∘ N) (hd (u # U));
          ¬ Λ.contains-head-reduction (M ∘ N);
          Λ.contains-head-reduction (hd (u # U));
          ¬ Λ.Ide (Λ.resid (M ∘ N) (Λ.head-strategy (M ∘ N)))⟧
            ⟹ ?P (Λ.resid (M ∘ N) (Λ.head-strategy (M ∘ N))) (tl (u # U));
         ⟦¬ Λ.Ide (M ∘ N);
          Λ.seq (M ∘ N) (hd (u # U));
          ¬ Λ.contains-head-reduction (M ∘ N);
          ¬ Λ.contains-head-reduction (hd (u # U))⟧
            ⟹ ?P M (filter notIde (map Λ.un-App1 (u # U)));
         ⟦¬ Λ.Ide (M ∘ N);
          Λ.seq (M ∘ N) (hd (u # U));
          ¬ Λ.contains-head-reduction (M ∘ N);
          ¬ Λ.contains-head-reduction (hd (u # U))⟧
            ⟹ ?P N (filter notIde (map Λ.un-App2 (u # U)))⟧
```

```
        ⟹ ?P (M ∘ N) (u # U)
proof −
  fix M N u U
  assume ind1: Λ.Ide (M ∘ N) ⟹ ?P (hd (u # U)) (tl (u # U))
  assume ind2: ⟦¬ Λ.Ide (M ∘ N);
                Λ.seq (M ∘ N) (hd (u # U));
                Λ.contains-head-reduction (M ∘ N);
                Λ.Ide (Λ.resid (M ∘ N) (Λ.head-redex (M ∘ N)))⟧
                  ⟹ ?P (hd (u # U)) (tl (u # U))
  assume ind3: ⟦¬ Λ.Ide (M ∘ N);
                Λ.seq (M ∘ N) (hd (u # U));
                Λ.contains-head-reduction (M ∘ N);
                ¬ Λ.Ide (Λ.resid (M ∘ N) (Λ.head-redex (M ∘ N)))⟧
                  ⟹ ?P (Λ.resid (M ∘ N) (Λ.head-redex (M ∘ N))) (u # U)
  assume ind4: ⟦¬ Λ.Ide (M ∘ N);
                Λ.seq (M ∘ N) (hd (u # U));
                ¬ Λ.contains-head-reduction (M ∘ N);
                Λ.contains-head-reduction (hd (u # U));
                Λ.Ide (Λ.resid (M ∘ N) (Λ.head-strategy (M ∘ N)))⟧
                  ⟹ ?P (Λ.head-strategy (M ∘ N)) (tl (u # U))
  assume ind5: ⟦¬ Λ.Ide (M ∘ N);
                Λ.seq (M ∘ N) (hd (u # U));
                ¬ Λ.contains-head-reduction (M ∘ N);
                Λ.contains-head-reduction (hd (u # U));
                ¬ Λ.Ide (Λ.resid (M ∘ N) (Λ.head-strategy (M ∘ N)))⟧
                  ⟹ ?P (Λ.resid (M ∘ N) (Λ.head-strategy (M ∘ N))) (tl (u # U))
  assume ind7: ⟦¬ Λ.Ide (M ∘ N);
                Λ.seq (M ∘ N) (hd (u # U));
                ¬ Λ.contains-head-reduction (M ∘ N);
                ¬ Λ.contains-head-reduction (hd (u # U))⟧
                  ⟹ ?P M (filter notIde (map Λ.un-App1 (u # U)))
  assume ind8: ⟦¬ Λ.Ide (M ∘ N);
                Λ.seq (M ∘ N) (hd (u # U));
                ¬ Λ.contains-head-reduction (M ∘ N);
                ¬ Λ.contains-head-reduction (hd (u # U))⟧
                  ⟹ ?P N (filter notIde (map Λ.un-App2 (u # U)))
  assume ∗: ¬ (Λ.is-Lam M ∧ Λ.is-Beta u)
  show ?P (M ∘ N) (u # U)
  proof (intro impI, elim conjE)
    assume seq: seq [M ∘ N] (u # U)
    assume Std: Std (u # U)
    have MN: Λ.Arr M ∧ Λ.Arr N
      using seq-char seq by force
    have u: Λ.Arr u
      using Std
      by (meson Std-imp-Arr Arr.simps(2) Con-Arr-self Con-implies-Arr(1)
          Con-initial-left Λ.arr-char list.simps(3))
    have U ≠ [] ⟹ Arr U
      using Std Std-imp-Arr Arr.simps(3)
```

```
  by (metis Arr.elims(3) list.discI)
have Λ.is-App u ∨ Λ.is-Beta u
  using ∗ seq MN u seq-char Λ.arr-char Srcs-simp_ΛP Λ.targets-char_Λ
  by (cases M; cases u) auto
have ∗∗: Λ.seq (M ⋅ N) u
  using Srcs-simp_ΛP seq-char seq Λ.seq-def u by force
show Std (stdz-insert (M ⋅ N) (u # U)) ∧
     (¬ Ide ((M ⋅ N) # u # U)
        ⟶ cong (stdz-insert (M ⋅ N) (u # U)) ((M ⋅ N) # u # U))
proof (cases Λ.Ide (M ⋅ N))
  assume 1: Λ.Ide (M ⋅ N)
  have MN: Λ.Arr M ∧ Λ.Arr N ∧ Λ.Ide M ∧ Λ.Ide N
    using MN 1 by simp
  have 2: stdz-insert (M ⋅ N) (u # U) = stdz-insert u U
    using MN 1
    by (simp add: Std seq stdz-insert-Ide-Std)
  show ?thesis
  proof (cases U = [])
    assume U: U = []
    have 2: stdz-insert (M ⋅ N) (u # U) = standard-development u
      using 1 2 U by simp
    show ?thesis
    proof (intro conjI)
      show Std (stdz-insert (M ⋅ N) (u # U))
        using 2 Std-standard-development by presburger
      show ¬ Ide ((M ⋅ N) # u # U) ⟶
               stdz-insert (M ⋅ N) (u # U) *∼* (M ⋅ N) # u # U
        by (metis 1 2 Ide.simps(2) U cong-cons-ideI(1) cong-standard-development
            ide-backward-stable ide-char Λ.ide-char prfx-transitive seq u)
    qed
    next
    assume U: U ≠ []
    have 2: stdz-insert (M ⋅ N) (u # U) = stdz-insert u U
      using 1 2 U by simp
    have 3: seq [u] U
      by (simp add: Std U arrI_P arr-append-imp-seq)
    have 4: Std (stdz-insert u U) ∧
            set (stdz-insert u U) ⊆ {a. Λ.elementary-reduction a} ∧
            (¬ Ide (u # U) ⟶ cong (stdz-insert u U) (u # U))
      using MN 3 Std ind1 Std-implies-set-subset-elementary-reduction
      by (metis 1 Std.simps(3) U list.sel(1) list.sel(3) standardize.cases)
    show ?thesis
    proof (intro conjI)
      show Std (stdz-insert (M ⋅ N) (u # U))
        by (metis 1 2 3 Std Std.simps(3) U ind1 list.exhaust-sel list.sel(1,3))
      show ¬ Ide ((M ⋅ N) # u # U) ⟶
               stdz-insert (M ⋅ N) (u # U) *∼* (M ⋅ N) # u # U
      proof
        assume 5: ¬ Ide ((M ⋅ N) # u # U)
```

```
      have stdz-insert (M ∘ N) (u # U) *∼* u # U
        using 1 2 4 5 seq-char seq by force
      also have u # U *∼* [M ∘ N] @ u # U
        using 1 Ide.simps(2) cong-append-ideI(1) ide-char seq by blast
      also have [M ∘ N] @ (u # U) = (M ∘ N) # u # U
        by simp
      finally show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
        by blast
    qed
  qed
qed
next
assume 1: ¬ Λ.Ide (M ∘ N)
show ?thesis
proof (cases Λ.contains-head-reduction (M ∘ N))
  assume 2: Λ.contains-head-reduction (M ∘ N)
  show ?thesis
  proof (cases Λ.Ide ((M ∘ N) \ Λ.head-redex (M ∘ N)))
    assume 3: Λ.Ide ((M ∘ N) \ Λ.head-redex (M ∘ N))
    have 4: ¬ Ide (u # U)
      by (metis Std Std-implies-set-subset-elementary-reduction in-mono
          Λ.elementary-reduction-not-ide list.set-intros(1) mem-Collect-eq
          set-Ide-subset-ide)
    have 5: stdz-insert (M ∘ N) (u # U) = Λ.head-redex (M ∘ N) # stdz-insert u U
      using MN 1 2 3 4 ** by auto
    show ?thesis
    proof (cases U = [])
      assume U: U = []
      have u: Λ.Arr u ∧ ¬ Λ.Ide u
        using 4 U u by force
      have 5: stdz-insert (M ∘ N) (u # U) =
                Λ.head-redex (M ∘ N) # standard-development u
        using 5 U by simp
      show ?thesis
      proof (intro conjI)
        show Std (stdz-insert (M ∘ N) (u # U))
        proof −
          have Λ.sseq (Λ.head-redex (M ∘ N)) (hd (standard-development u))
          proof −
            have Λ.seq (Λ.head-redex (M ∘ N)) (hd (standard-development u))
            proof
              show Λ.Arr (Λ.head-redex (M ∘ N))
                using MN Λ.Arr.simps(4) Λ.Arr-head-redex by presburger
              show Λ.Arr (hd (standard-development u))
                using Arr-imp-arr-hd Ide-iff-standard-development-empty
                      Std-standard-development u
                by force
            show Λ.Trg (Λ.head-redex (M ∘ N)) = Λ.Src (hd (standard-development u))
              proof −
```

```
      have Λ.Trg (Λ.head-redex (M ⋅ N)) =
           Λ.Trg ((M ⋅ N) \ Λ.head-redex (M ⋅ N))
        by (metis 3 MN Λ.Con-Arr-head-redex Λ.Src-resid
           Λ.Arr.simps(4) Λ.Ide-iff-Src-self Λ.Ide-iff-Trg-self
           Λ.Ide-implies-Arr)
      also have ... = Λ.Src u
        using MN
        by (metis Trg-last-Src-hd-eqI Trg-last-eqI head-redex-decomp
           Λ.Arr.simps(4) last-ConsL last-appendR list.sel(1)
           not-Cons-self2 seq)
      also have ... = Λ.Src (hd (standard-development u))
        using ** 2 3 u MN Src-hd-standard-development [of u] by metis
      finally show ?thesis by blast
    qed
  qed
  thus ?thesis
    by (metis 2 u MN Λ.Arr.simps(4) Ide-iff-standard-development-empty
       development.simps(2) development-standard-development
       Λ.head-redex-is-head-reduction list.exhaust-sel
       Λ.sseq-head-reductionI)
qed
thus ?thesis
  by (metis 5 Ide-iff-standard-development-empty Std.simps(3)
     Std-standard-development list.exhaust u)
qed
show ¬ Ide ((M ⋅ N) # u # U) ⟶
       stdz-insert (M ⋅ N) (u # U) *∼* (M ⋅ N) # u # U
proof
  have stdz-insert (M ⋅ N) (u # U) =
       [Λ.head-redex (M ⋅ N)] @ standard-development u
    using 5 by simp
  also have ... *∼* [Λ.head-redex (M ⋅ N)] @ [u]
    using u cong-standard-development [of u] cong-append
    by (metis 2 5 Ide-iff-standard-development-empty Std-imp-Arr
       ‹Std (stdz-insert (M ⋅ N) (u # U))›
       arr-append-imp-seq arr-char calculation cong-standard-development
       cong-transitive Λ.Arr-head-redex Λ.contains-head-reduction-iff
       list.distinct(1))
  also have [Λ.head-redex (M ⋅ N)] @ [u] *∼*
            ([Λ.head-redex (M ⋅ N)] @ [(M ⋅ N) \ Λ.head-redex (M ⋅ N)]) @ [u]
  proof −
    have [Λ.head-redex (M ⋅ N)] *∼*
         [Λ.head-redex (M ⋅ N)] @ [(M ⋅ N) \ Λ.head-redex (M ⋅ N)]
      by (metis (no-types, lifting) 1 3 MN Arr-iff-Con-self Ide.simps(2)
         Resid.simps(2) arr-append-imp-seq arr-char cong-append-ideI(4)
         cong-transitive head-redex-decomp ide-backward-stable ide-char
         Λ.Arr.simps(4) Λ.ide-char not-Cons-self2)
    thus ?thesis
      using MN U u seq
```

```
      by (meson cong-append head-redex-decomp Λ.Arr.simps(4) prfx-transitive)
    qed
    also have ([Λ.head-redex (M ∘ N)] @
                 [(M ∘ N) \ Λ.head-redex (M ∘ N)]) @ [u] *∼*
              [M ∘ N] @ [u]
      by (metis Λ.Arr.simps(4) MN U Resid-Arr-self cong-append ide-char
          seq-char head-redex-decomp seq)
    also have [M ∘ N] @ [u] = (M ∘ N) # u # U
      using U by simp
    finally show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
      by blast
  qed
qed
next
assume U: U ≠ []
have 6: Std (stdz-insert u U) ∧
        set (stdz-insert u U) ⊆ {a. Λ.elementary-reduction a} ∧
        cong (stdz-insert u U) (u # U)
proof −
  have seq [u] U
    by (simp add: Std U arrI_P arr-append-imp-seq)
  moreover have Std U
    using Std Std.elims(2) U by blast
  ultimately show ?thesis
    using ind2 ** 1 2 3 4 Std-implies-set-subset-elementary-reduction
    by force
qed
show ?thesis
proof (intro conjI)
  show Std (stdz-insert (M ∘ N) (u # U))
  proof −
    have Λ.sseq (Λ.head-redex (M ∘ N)) (hd (stdz-insert u U))
    proof −
      have Λ.seq (Λ.head-redex (M ∘ N)) (hd (stdz-insert u U))
      proof
        show Λ.Arr (Λ.head-redex (M ∘ N))
          using MN Λ.Arr-head-redex by force
        show Λ.Arr (hd (stdz-insert u U))
          using 6
          by (metis Arr-imp-arr-hd Con-implies-Arr(2) Ide.simps(1) ide-char
              Λ.arr-char)
        show Λ.Trg (Λ.head-redex (M ∘ N)) = Λ.Src (hd (stdz-insert u U))
        proof −
          have Λ.Trg (Λ.head-redex (M ∘ N)) =
               Λ.Trg ((M ∘ N) \ Λ.head-redex (M ∘ N))
            by (metis 3 Λ.Arr-not-Nil Λ.Ide-iff-Src-self
                Λ.Ide-iff-Trg-self Λ.Ide-implies-Arr Λ.Src-resid)
          also have ... = Λ.Trg (M ∘ N)
            by (metis 1 MN Trg-last-eqI Trg-last-standard-development
```

```
        cong-standard-development head-redex-decomp Λ.Arr.simps(4)
        last-snoc)
    also have ... = Λ.Src (hd (stdz-insert u U))
      by (metis ** 6 Src-hd-eqI Λ.seqE_Λ list.sel(1))
    finally show ?thesis by blast
  qed
 qed
 thus ?thesis
  by (metis 2 6 MN Λ.Arr.simps(4) Std.elims(1) Ide.simps(1)
      Resid.simps(2) ide-char Λ.head-redex-is-head-reduction
      list.sel(1) Λ.sseq-head-reductionI Λ.sseq-imp-elementary-reduction1)
qed
thus ?thesis
 by (metis 5 6 Std.simps(3) Arr.simps(1) Con-implies-Arr(1)
     con-char prfx-implies-con list.exhaust-sel)
qed
show ¬ Ide ((M ⋅ N) # u # U) ⟶
       stdz-insert (M ⋅ N) (u # U) *∼* (M ⋅ N) # u # U
proof
 have stdz-insert (M ⋅ N) (u # U) =
      [Λ.head-redex (M ⋅ N)] @ stdz-insert u U
   using 5 by simp
 also have 7: [Λ.head-redex (M ⋅ N)] @ stdz-insert u U *∼*
               [Λ.head-redex (M ⋅ N)] @ u # U
   using 6 cong-append [of [Λ.head-redex (M ⋅ N)] stdz-insert u U
                           [Λ.head-redex (M ⋅ N)] u # U]
   by (metis 2 5 Arr.simps(1) Resid.simps(2) Std-imp-Arr
       ‹Std (stdz-insert (M ⋅ N) (u # U))›
       arr-append-imp-seq arr-char calculation cong-standard-development
       cong-transitive ide-implies-arr Λ.Arr-head-redex
       Λ.contains-head-reduction-iff list.distinct(1))
 also have [Λ.head-redex (M ⋅ N)] @ u # U *∼*
           ([Λ.head-redex (M ⋅ N)] @
             [(M ⋅ N) \ Λ.head-redex (M ⋅ N)]) @ u # U
 proof −
   have [Λ.head-redex (M ⋅ N)] *∼*
        [Λ.head-redex (M ⋅ N)] @ [(M ⋅ N) \ Λ.head-redex (M ⋅ N)]
     by (metis 2 3 head-redex-decomp Λ.Arr-head-redex
         Λ.Con-Arr-head-redex Λ.Ide-iff-Src-self Λ.Ide-implies-Arr
         Λ.Src-resid Λ.contains-head-reduction-iff Λ.resid-Arr-self
         prfx-decomp prfx-transitive)
   moreover have seq [Λ.head-redex (M ⋅ N)] (u # U)
     by (metis 7 arr-append-imp-seq cong-implies-coterminal coterminalE
         list.distinct(1))
   ultimately show ?thesis
     using 3 ide-char cong-symmetric cong-append
     by (meson 6 prfx-transitive)
 qed
 also have ([Λ.head-redex (M ⋅ N)] @
```

```
                [(M ∘ N) \ Λ.head-redex (M ∘ N)]) @ u # U *∼*
              [M ∘ N] @ u # U
      by (meson 6 MN Λ.Arr.simps(4) cong-append prfx-transitive
          head-redex-decomp seq)
    also have [M ∘ N] @ (u # U) = (M ∘ N) # u # U
      by simp
    finally show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
      by blast
  qed
 qed
qed
next
assume 3: ¬ Λ.Ide ((M ∘ N) \ Λ.head-redex (M ∘ N))
have 4: stdz-insert (M ∘ N) (u # U) =
        Λ.head-redex (M ∘ N) #
          stdz-insert ((M ∘ N) \ Λ.head-redex (M ∘ N)) (u # U)
 using MN 1 2 3 ** by auto
have 5: Std (stdz-insert ((M ∘ N) \ Λ.head-redex (M ∘ N)) (u # U)) ∧
        set (stdz-insert ((M ∘ N) \ Λ.head-redex (M ∘ N)) (u # U))
          ⊆ {a. Λ.elementary-reduction a} ∧
        stdz-insert ((M ∘ N) \ Λ.head-redex (M ∘ N)) (u # U) *∼*
        (M ∘ N) \ Λ.head-redex (M ∘ N) # u # U
proof −
 have seq [(M ∘ N) \ Λ.head-redex (M ∘ N)] (u # U)
   by (metis (full-types) MN arr-append-imp-seq cong-implies-coterminal
       coterminalE head-redex-decomp Λ.Arr.simps(4) not-Cons-self2
       seq seq-def targets-append)
 thus ?thesis
   using ind3 1 2 3 ** Std Std-implies-set-subset-elementary-reduction
   by auto
qed
show ?thesis
proof (intro conjI)
 show Std (stdz-insert (M ∘ N) (u # U))
 proof −
   have Λ.sseq (Λ.head-redex (M ∘ N))
               (hd (stdz-insert ((M ∘ N) \ Λ.head-redex (M ∘ N)) (u # U)))
   proof −
     have Λ.seq (Λ.head-redex (M ∘ N))
                (hd (stdz-insert ((M ∘ N) \ Λ.head-redex (M ∘ N)) (u # U)))
       using MN 5 Λ.Arr-head-redex
       by (metis (no-types, lifting) Arr-imp-arr-hd Con-implies-Arr(2)
           Ide.simps(1) Src-hd-eqI ide-char Λ.Arr.simps(4) Λ.Arr-head-redex
           Λ.Con-Arr-head-redex Λ.Src-resid Λ.arr-char Λ.seq-char list.sel(1))
     moreover have Λ.elementary-reduction
                     (hd (stdz-insert ((M ∘ N) \ Λ.head-redex (M ∘ N))
                                      (u # U)))
       using 5
       by (metis Arr.simps(1) Con-implies-Arr(2) Ide.simps(1) hd-in-set
```

```
          ide-char mem-Collect-eq subset-code(1))
      ultimately show ?thesis
        using MN 2 Λ.head-redex-is-head-reduction Λ.sseq-head-reductionI
        by simp
    qed
    thus ?thesis
      by (metis 4 5 Std.simps(3) Arr.simps(1) Con-implies-Arr(2)
          Ide.simps(1) ide-char list.exhaust-sel)
  qed
  show ¬ Ide ((M ∘ N) # u # U) ⟶
         stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
  proof
    have stdz-insert (M ∘ N) (u # U) =
        [Λ.head-redex (M ∘ N)] @
          stdz-insert ((M ∘ N) \ Λ.head-redex (M ∘ N)) (u # U)
      using 4 by simp
    also have ... *∼* [Λ.head-redex (M ∘ N)] @
                      ((M ∘ N) \ Λ.head-redex (M ∘ N) # u # U)
    proof (intro cong-append)
      show seq [Λ.head-redex (M ∘ N)]
               (stdz-insert ((M ∘ N) \ Λ.head-redex (M ∘ N)) (u # U))
        by (metis 4 5 Ide.simps(1) Resid.simps(1) Std-imp-Arr
            ‹Std (stdz-insert (M ∘ N) (u # U))› arrI_P arr-append-imp-seq
            calculation ide-char list.discI)
      show [Λ.head-redex (M ∘ N)] *∼* [Λ.head-redex (M ∘ N)]
        using MN Λ.cong-reflexive ide-char Λ.Arr-head-redex by force
     show stdz-insert ((M ∘ N) \ Λ.head-redex (M ∘ N)) (u # U) *∼* (M ∘ N) \
          Λ.head-redex (M ∘ N) # u # U
        using 5 by fastforce
    qed
    also have ([Λ.head-redex (M ∘ N)] @
                ((M ∘ N) \ Λ.head-redex (M ∘ N) # u # U)) =
              ([Λ.head-redex (M ∘ N)] @
                [(M ∘ N) \ Λ.head-redex (M ∘ N)]) @ (u # U)
      by simp
    also have ([Λ.head-redex (M ∘ N)] @
                [(M ∘ N) \ Λ.head-redex (M ∘ N)]) @ u # U *∼*
              [M ∘ N] @ u # U
      by (meson ** cong-append cong-reflexive seqE head-redex-decomp
          seq Λ.seq-char)
    also have [M ∘ N] @ (u # U) = (M ∘ N) # u # U
      by simp
    finally show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
      by blast
  qed
 qed
qed
next
assume 2: ¬ Λ.contains-head-reduction (M ∘ N)
```

```
show ?thesis
proof (cases Λ.contains-head-reduction u)
  assume 3: Λ.contains-head-reduction u
  have B: [Λ.head-strategy (M ∘ N)] @ [(M ∘ N) \ Λ.head-strategy (M ∘ N)] *∼*
          [M ∘ N] @ [u]
  proof −
    have [M ∘ N] @ [u] *∼* [Λ.head-strategy (Λ.Src (M ∘ N)) ⊔ M ∘ N]
    proof −
      have Λ.is-internal-reduction (M ∘ N)
        using 2 ** Λ.is-internal-reduction-iff by blast
      moreover have Λ.is-head-reduction u
      proof −
        have Λ.elementary-reduction u
          by (metis Std lambda-calculus.sseq-imp-elementary-reduction1
              list.discI list.sel(1) reduction-paths.Std.elims(2))
        thus ?thesis
          using Λ.is-head-reduction-if 3 by force
      qed
      moreover have Λ.head-strategy (Λ.Src (M ∘ N)) \ (M ∘ N) = u
        using Λ.resid-head-strategy-Src(1) ** calculation(1−2) by fastforce
      moreover have [M ∘ N] *≲* [Λ.head-strategy (Λ.Src (M ∘ N)) ⊔ M ∘ N]
        using MN Λ.prfx-implies-con ide-char Λ.Arr-head-strategy
              Λ.Src-head-strategy Λ.prfx-Join
        by force
      ultimately show ?thesis
        using u Λ.Coinitial-iff-Con Λ.Arr-not-Nil Λ.resid-Join
              prfx-decomp [of M ∘ N Λ.head-strategy (Λ.Src (M ∘ N)) ⊔ M ∘ N]
        by simp
    qed
    also have [Λ.head-strategy (Λ.Src (M ∘ N)) ⊔ M ∘ N] *∼*
              [Λ.head-strategy (Λ.Src (M ∘ N))] @
                [(M ∘ N) \ Λ.head-strategy (Λ.Src (M ∘ N))]
    proof −
      have 3: Λ.composite-of
                (Λ.head-strategy (Λ.Src (M ∘ N)))
                ((M ∘ N) \ Λ.head-strategy (Λ.Src (M ∘ N)))
                (Λ.head-strategy (Λ.Src (M ∘ N)) ⊔ M ∘ N)
        using Λ.Arr-head-strategy MN Λ.Src-head-strategy Λ.join-of-Join
              Λ.join-of-def
        by force
      hence composite-of
              [Λ.head-strategy (Λ.Src (M ∘ N))]
              [(M ∘ N) \ Λ.head-strategy (Λ.Src (M ∘ N))]
              [Λ.head-strategy (Λ.Src (M ∘ N)) ⊔ M ∘ N]
        using composite-of-single-single
        by (metis (no-types, lifting) Λ.Con-sym Ide.simps(2) Resid.simps(3)
            composite-ofI Λ.composite-ofE Λ.con-char ide-char Λ.prfx-implies-con)
      hence [Λ.head-strategy (Λ.Src (M ∘ N))] @
              [(M ∘ N) \ Λ.head-strategy (Λ.Src (M ∘ N))] *∼*
```

```
        [Λ.head-strategy (Λ.Src (M ∘ N)) ⊔ M ∘ N]
    using Λ.resid-Join
    by (meson 3 composite-of-single-single composite-of-unq-upto-cong)
  thus ?thesis by blast
 qed
 also have [Λ.head-strategy (Λ.Src (M ∘ N))] @
             [(M ∘ N) \ Λ.head-strategy (Λ.Src (M ∘ N))] *∼*
           [Λ.head-strategy (M ∘ N)] @
             [(M ∘ N) \ Λ.head-strategy (M ∘ N)]
  by (metis (full-types) Λ.Arr.simps(4) MN prfx-transitive calculation
      Λ.head-strategy-Src)
 finally show ?thesis by blast
qed
show ?thesis
proof (cases Λ.Ide ((M ∘ N) \ Λ.head-strategy (M ∘ N)))
 assume 4: Λ.Ide ((M ∘ N) \ Λ.head-strategy (M ∘ N))
 have A: [Λ.head-strategy (M ∘ N)] *∼*
         [Λ.head-strategy (M ∘ N)] @ [(M ∘ N) \ Λ.head-strategy (M ∘ N)]
  by (meson 4 B Con-implies-Arr(1) Ide.simps(2) arr-append-imp-seq arr-char
      con-char cong-append-ideI(2) ide-char Λ.ide-char not-Cons-self2
      prfx-implies-con)
 have 5: ¬ Ide (u # U)
  by (meson 3 Ide-consE Λ.ide-backward-stable Λ.subs-head-redex
      Λ.subs-implies-prfx Λ.contains-head-reduction-iff
      Λ.elementary-reduction-head-redex Λ.elementary-reduction-not-ide)
 have 6: stdz-insert (M ∘ N) (u # U) =
         stdz-insert (Λ.head-strategy (M ∘ N)) U
  using 1 2 3 4 5 * ** ‹Λ.is-App u ∨ Λ.is-Beta u›
  apply (cases u)
    apply simp-all
   apply blast
  by (cases M) auto
 show ?thesis
 proof (cases U = [])
  assume U: U = []
  have u: ¬ Λ.Ide u
   using 5 U by simp
  have 6: stdz-insert (M ∘ N) (u # U) =
          standard-development (Λ.head-strategy (M ∘ N))
   using 6 U by simp
  show ?thesis
  proof (intro conjI)
   show Std (stdz-insert (M ∘ N) (u # U))
    using 6 Std-standard-development by presburger
   show ¬ Ide ((M ∘ N) # u # U) ⟶
            stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
   proof
    have stdz-insert (M ∘ N) (u # U) *∼* [Λ.head-strategy (M ∘ N)]
     using 4 6 cong-standard-development ** 1 2 3 Λ.Arr.simps(4)
```

```
        Λ.Arr-head-strategy MN Λ.ide-backward-stable Λ.ide-char
    by metis
  also have [Λ.head-strategy (M ∘ N)] *∼* [M ∘ N] @ [u]
    by (meson A B prfx-transitive)
  also have [M ∘ N] @ [u] = (M ∘ N) # u # U
    using U by auto
  finally show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
    by blast
 qed
qed
next
assume U: U ≠ []
have 7: seq [Λ.head-strategy (M ∘ N)] U
proof
 show Arr [Λ.head-strategy (M ∘ N)]
   by (meson A Con-implies-Arr(1) con-char prfx-implies-con)
 show Arr U
   using U ‹U ≠ [] ⟹ Arr U› by presburger
 show Λ.Trg (last [Λ.head-strategy (M ∘ N)]) = Λ.Src (hd U)
 by (metis A B Std Std-consE Trg-last-eqI U Λ.seqE_Λ Λ.sseq-imp-seq last-snoc)
qed
have 8: Std (stdz-insert (Λ.head-strategy (M ∘ N)) U) ∧
        set (stdz-insert (Λ.head-strategy (M ∘ N)) U)
          ⊆ {a. Λ.elementary-reduction a} ∧
        stdz-insert (Λ.head-strategy (M ∘ N)) U *∼*
        Λ.head-strategy (M ∘ N) # U
proof −
 have Std U
   by (metis Std Std.simps(3) U list.exhaust-sel)
 moreover have ¬ Ide (Λ.head-strategy (M ∘ N) # tl (u # U))
   using 1 4 Λ.ide-backward-stable by blast
 ultimately show ?thesis
   using ind4 ** 1 2 3 4 7 Std-implies-set-subset-elementary-reduction
   by force
qed
show ?thesis
proof (intro conjI)
 show Std (stdz-insert (M ∘ N) (u # U))
   using 6 8 by presburger
 show ¬ Ide ((M ∘ N) # u # U) ⟶
           stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
 proof
   have stdz-insert (M ∘ N) (u # U) =
        stdz-insert (Λ.head-strategy (M ∘ N)) U
     using 6 by simp
   also have ... *∼* [Λ.head-strategy (M ∘ N)] @ U
     using 8 by simp
   also have [Λ.head-strategy (M ∘ N)] @ U *∼* ([M ∘ N] @ [u]) @ U
     by (meson A B U 7 Resid-Arr-self cong-append ide-char
```

```
        prfx-transitive ‹U ≠ [] ⟹ Arr U›)
    also have ([M ∘ N] @ [u]) @ U = (M ∘ N) # u # U
      by simp
    finally show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
      by blast
  qed
 qed
qed
next
assume 4: ¬ Λ.Ide ((M ∘ N) \ Λ.head-strategy (M ∘ N))
show ?thesis
proof (cases U = [])
 assume U: U = []
 have 5: stdz-insert (M ∘ N) (u # U) =
         Λ.head-strategy (M ∘ N) #
           standard-development ((M ∘ N) \ Λ.head-strategy (M ∘ N))
   using 1 2 3 4 U ∗ ∗∗ ‹Λ.is-App u ∨ Λ.is-Beta u›
   apply (cases u)
     apply simp-all
    apply blast
   apply (cases M)
     apply simp-all
   by blast+
 show ?thesis
 proof (intro conjI)
   show Std (stdz-insert (M ∘ N) (u # U))
   proof −
     have Λ.sseq (Λ.head-strategy (M ∘ N))
                 (hd (standard-development
                       ((M ∘ N) \ Λ.head-strategy (M ∘ N))))
     proof −
       have Λ.seq (Λ.head-strategy (M ∘ N))
                  (hd (standard-development
                        ((M ∘ N) \ Λ.head-strategy (M ∘ N))))
         using MN ∗∗ 4 Λ.Arr-head-strategy Arr-imp-arr-hd
               Ide-iff-standard-development-empty Src-hd-standard-development
               Std-imp-Arr Std-standard-development Λ.Arr-resid
               Λ.Src-head-strategy Λ.Src-resid
         by force
       moreover have Λ.elementary-reduction
                       (hd (standard-development
                             ((M ∘ N) \ Λ.head-strategy (M ∘ N))))
         by (metis 4 Ide-iff-standard-development-empty MN Std-consE
             Std-standard-development hd-Cons-tl Λ.Arr.simps(4)
             Λ.Arr-resid Λ.Con-head-strategy
             Λ.sseq-imp-elementary-reduction1 Std.simps(2))
       ultimately show ?thesis
         using Λ.sseq-head-reductionI Std-standard-development
         by (metis ∗∗ 2 3 Std U Λ.internal-reduction-preserves-no-head-redex
```

```
        Λ.is-internal-reduction-iff Λ.Src-head-strategy
        Λ.elementary-reduction-not-ide Λ.head-strategy-Src
        Λ.head-strategy-is-elementary Λ.ide-char Λ.is-head-reduction-char
        Λ.is-head-reduction-if Λ.seqE_Λ Std.simps(2))
  qed
  thus ?thesis
    by (metis 4 5 MN Ide-iff-standard-development-empty
        Std-standard-development Λ.Arr.simps(4) Λ.Arr-resid
        Λ.Con-head-strategy list.exhaust-sel Std.simps(3))
qed
show ¬ Ide ((M ∘ N) # u # U) ⟶
       stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
proof
  have stdz-insert (M ∘ N) (u # U) =
       [Λ.head-strategy (M ∘ N)] @
         standard-development ((M ∘ N) \ Λ.head-strategy (M ∘ N))
    using 5 by simp
  also have ... *∼* [Λ.head-strategy (M ∘ N)] @
                    [(M ∘ N) \ Λ.head-strategy (M ∘ N)]
  proof (intro cong-append)
    show 6: seq [Λ.head-strategy (M ∘ N)]
                (standard-development
                  ((M ∘ N) \ Λ.head-strategy (M ∘ N)))
      using 4 Ide-iff-standard-development-empty MN
            ‹Std (stdz-insert (M ∘ N) (u # U))›
            arr-append-imp-seq arr-char calculation Λ.Arr-head-strategy
            Λ.Arr-resid lambda-calculus.Src-head-strategy
      by force
    show [Λ.head-strategy (M ∘ N)] *∼* [Λ.head-strategy (M ∘ N)]
      by (meson MN 6 cong-reflexive seqE)
    show standard-development ((M ∘ N) \ Λ.head-strategy (M ∘ N)) *∼*
         [(M ∘ N) \ Λ.head-strategy (M ∘ N)]
      using 4 MN cong-standard-development Λ.Arr.simps(4)
            Λ.Arr-resid Λ.Con-head-strategy
      by presburger
  qed
  also have [Λ.head-strategy (M ∘ N)] @
              [(M ∘ N) \ Λ.head-strategy (M ∘ N)] *∼*
            [M ∘ N] @ [u]
    using B by blast
  also have [M ∘ N] @ [u] = (M ∘ N) # u # U
    using U by simp
  finally show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
    by blast
qed
qed
next
assume U: U ≠ []
have 5: stdz-insert (M ∘ N) (u # U) =
```

```
        Λ.head-strategy (M ∘ N) #
          stdz-insert (Λ.resid (M ∘ N) (Λ.head-strategy (M ∘ N))) U
  using 1 2 3 4 U ∗ ∗∗ ‹Λ.is-App u ∨ Λ.is-Beta u›
  apply (cases u)
    apply simp-all
   apply blast
  apply (cases M)
     apply simp-all
  by blast+
have 6: Std (stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U) ∧
        set (stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U)
          ⊆ {a. Λ.elementary-reduction a} ∧
        stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U *∼*
        (M ∘ N) \ Λ.head-strategy (M ∘ N) # U
proof −
  have seq [(M ∘ N) \ Λ.head-strategy (M ∘ N)] U
  proof
    show Arr [(M ∘ N) \ Λ.head-strategy (M ∘ N)]
      by (simp add: MN Λ.Arr-resid Λ.Con-head-strategy)
    show Arr U
      using U ‹U ≠ [] ⟹ Arr U› by blast
    show Λ.Trg (last [(M ∘ N) \ Λ.head-strategy (M ∘ N)]) = Λ.Src (hd U)
      by (metis (mono-tags, lifting) B U Std Std-consE Trg-last-eqI
          Λ.seq-char Λ.sseq-imp-seq last-ConsL last-snoc)
  qed
  thus ?thesis
    using ind5 Std-implies-set-subset-elementary-reduction
    by (metis ∗∗ 1 2 3 4 Std Std.simps(3) Arr-iff-Con-self Ide.simps(3)
        Resid.simps(1) seq-char Λ.ide-char list.exhaust-sel list.sel(1,3))
qed
show ?thesis
proof (intro conjI)
  show Std (stdz-insert (M ∘ N) (u # U))
  proof −
    have Λ.sseq (Λ.head-strategy (M ∘ N))
                (hd (stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U))
    proof −
      have Λ.seq (Λ.head-strategy (M ∘ N))
                 (hd (stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U))
      proof
        show Λ.Arr (Λ.head-strategy (M ∘ N))
          using MN Λ.Arr-head-strategy by force
        show Λ.Arr (hd (stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U))
          using 6
          by (metis Ide.simps(1) Resid.simps(2) Std-consE hd-Cons-tl ide-char)
        show Λ.Trg (Λ.head-strategy (M ∘ N)) =
             Λ.Src (hd (stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U))
          using 6
          by (metis MN Src-hd-eqI Λ.Arr.simps(4) Λ.Con-head-strategy
```

```
        Λ.Src-resid list.sel(1))
    qed
    moreover have Λ.is-head-reduction (Λ.head-strategy (M ∘ N))
      using ** 1 2 3 Λ.Src-head-strategy Λ.head-strategy-is-elementary
          Λ.head-strategy-Src Λ.is-head-reduction-char Λ.seq-char
      by (metis Λ.Src-head-redex Λ.contains-head-reduction-iff
          Λ.head-redex-is-head-reduction
          Λ.internal-reduction-preserves-no-head-redex
          Λ.is-internal-reduction-iff)
    moreover have Λ.elementary-reduction
                  (hd (stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U))
      by (metis 6 Ide.simps(1) Resid.simps(2) ide-char hd-in-set
          in-mono mem-Collect-eq)
    ultimately show ?thesis
      using Λ.sseq-head-reductionI by blast
  qed
  thus ?thesis
    by (metis 5 6 Std.simps(3) Arr.simps(1) Con-implies-Arr(1)
        con-char prfx-implies-con list.exhaust-sel)
qed
show ¬ Ide ((M ∘ N) # u # U) ⟶
       stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
proof
  have stdz-insert (M ∘ N) (u # U) =
       [Λ.head-strategy (M ∘ N)] @
         stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U
    using 5 by simp
  also have 10: ... *∼* [Λ.head-strategy (M ∘ N)] @
                        ((M ∘ N) \ Λ.head-strategy (M ∘ N) # U)
  proof (intro cong-append)
    show 10: seq [Λ.head-strategy (M ∘ N)]
                 (stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U)
      by (metis 5 6 Ide.simps(1) Resid.simps(1) Std-imp-Arr
          ‹Std (stdz-insert (M ∘ N) (u # U))› arr-append-imp-seq
          arr-char calculation ide-char list.distinct(1))
    show [Λ.head-strategy (M ∘ N)] *∼* [Λ.head-strategy (M ∘ N)]
      using MN 10 cong-reflexive by blast
    show stdz-insert ((M ∘ N) \ Λ.head-strategy (M ∘ N)) U *∼*
         (M ∘ N) \ Λ.head-strategy (M ∘ N) # U
      using 6 by auto
  qed
  also have 11: [Λ.head-strategy (M ∘ N)] @
                  ((M ∘ N) \ Λ.head-strategy (M ∘ N) # U) =
                ([Λ.head-strategy (M ∘ N)] @
                  [(M ∘ N) \ Λ.head-strategy (M ∘ N)]) @ U
    by simp
  also have ... *∼* (([M ∘ N] @ [u]) @ U)
  proof −
    have seq ([Λ.head-strategy (M ∘ N)] @
```

```
                [(M ∘ N) \ Λ.head-strategy (M ∘ N)]) U
        by (metis U 10 11 append-is-Nil-conv arr-append-imp-seq
            cong-implies-coterminal coterminalE not-Cons-self2)
      thus ?thesis
        using B cong-append cong-reflexive by blast
    qed
    also have ([M ∘ N] @ [u]) @ U = (M ∘ N) # u # U
      by simp
    finally show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
      by blast
  qed
 qed
qed
next
assume 3: ¬ Λ.contains-head-reduction u
have u: Λ.Arr u ∧ Λ.is-App u ∧ ¬ Λ.contains-head-reduction u
 using 3 ‹Λ.is-App u ∨ Λ.is-Beta u› Λ.is-Beta-def u by force
have 5: ¬ Λ.Ide u
 by (metis Std Std.simps(2) Std.simps(3) Λ.elementary-reduction-not-ide
     Λ.ide-char neq-Nil-conv Λ.sseq-imp-elementary-reduction1)
show ?thesis
proof −
 have 4: stdz-insert (M ∘ N) (u # U) =
         map (λX. Λ.App X (Λ.Src N))
             (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) @
         map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
             (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))
   using u MN 1 2 3 5 ∗ ∗∗ ‹Λ.is-App u ∨ Λ.is-Beta u›
   apply (cases u)
     apply simp-all
   apply (cases U = [])
    apply simp-all
   by blast+
 have ∗∗∗: set U ⊆ Collect Λ.is-App
   using u 5 Std seq-App-Std-implies by blast
 have X: Std (filter notIde (map Λ.un-App1 (u # U)))
   by (metis ∗∗∗ Std Std-filter-map-un-App1 insert-subset list.simps(15)
       mem-Collect-eq u)
 have Y: Std (filter notIde (map Λ.un-App2 (u # U)))
   by (metis ∗∗∗ u Std Std-filter-map-un-App2 insert-subset list.simps(15)
       mem-Collect-eq)
 have A: ¬ Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide ⟹
           Std (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) ∧
           set (stdz-insert M (filter notIde (map Λ.un-App1 (u # U))))
             ⊆ {a. Λ.elementary-reduction a} ∧
           stdz-insert M (filter notIde (map Λ.un-App1 (u # U))) *∼*
           M # filter notIde (map Λ.un-App1 (u # U))
 proof −
```

```
  assume ∗: ¬ Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide
  have seq [M] (filter notIde (map Λ.un-App1 (u # U)))
  proof
    show Arr [M]
      using MN by simp
    show Arr (filter notIde (map Λ.un-App1 (u # U)))
      by (metis (mono-tags, lifting) ∗ Std-imp-Arr X empty-filter-conv
          list.set-map mem-Collect-eq subset-code(1))
   show Λ.Trg (last [M]) = Λ.Src (hd (filter notIde (map Λ.un-App1 (u # U))))
    proof −
      have Λ.Trg (last [M]) = Λ.Src (hd (map Λ.un-App1 (u # U)))
        using ∗∗ u by fastforce
      also have ... = Λ.Src (hd (filter notIde (map Λ.un-App1 (u # U))))
      proof −
        have Arr (map Λ.un-App1 (u # U))
          using u ∗∗∗
          by (metis Arr-map-un-App1 Std Std-imp-Arr insert-subset
              list.simps(15) mem-Collect-eq neq-Nil-conv)
        moreover have ¬ Ide (map Λ.un-App1 (u # U))
          by (metis ∗ Collect-cong Λ.ide-char list.set-map set-Ide-subset-ide)
        ultimately show ?thesis
          using Src-hd-eqI cong-filter-notIde by blast
      qed
      finally show ?thesis by blast
    qed
  qed
  moreover have ¬ Ide (M # filter notIde (map Λ.un-App1 (u # U)))
    using ∗
    by (metis (no-types, lifting) ∗∗∗ Arr-map-un-App1 Std Std-imp-Arr
        Arr.simps(1) Ide.elims(2) Resid-Arr-Ide-ind ide-char
        seq-char calculation(1) cong-filter-notIde filter-notIde-Ide
        insert-subset list.discI list.sel(3) list.simps(15) mem-Collect-eq u)
  ultimately show ?thesis
    by (metis X 1 2 3 ∗∗ ind7 Std-implies-set-subset-elementary-reduction
        list.sel(1))
qed
have B: ¬ Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide ⟹
          Std (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))) ∧
          set (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))
            ⊆ {a. Λ.elementary-reduction a} ∧
          stdz-insert N (filter notIde (map Λ.un-App2 (u # U))) *∼*
          N # filter notIde (map Λ.un-App2 (u # U))
proof −
  assume ∗∗: ¬ Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide
  have seq [N] (filter notIde (map Λ.un-App2 (u # U)))
  proof
    show Arr [N]
      using MN by simp
    show Arr (filter (λu. ¬ Λ.Ide u) (map Λ.un-App2 (u # U)))
```

```
      by (metis (mono-tags, lifting) ** Std-imp-Arr Y empty-filter-conv
          list.set-map mem-Collect-eq subset-code(1))
    show Λ.Trg (last [N]) = Λ.Src (hd (filter notIde (map Λ.un-App2 (u # U))))
     proof −
      have Λ.Trg (last [N]) = Λ.Src (hd (map Λ.un-App2 (u # U)))
        by (metis u seq Trg-last-Src-hd-eqI Λ.Src.simps(4)
            Λ.Trg.simps(3) Λ.is-App-def Λ.lambda.sel(4) last-ConsL
            list.discI list.map-sel(1) list.sel(1))
      also have ... = Λ.Src (hd (filter notIde (map Λ.un-App2 (u # U))))
      proof −
        have Arr (map Λ.un-App2 (u # U))
          using u ***
          by (metis Arr-map-un-App2 Std Std-imp-Arr list.distinct(1)
              mem-Collect-eq set-ConsD subset-code(1))
        moreover have ¬ Ide (map Λ.un-App2 (u # U))
          by (metis ** Collect-cong Λ.ide-char list.set-map set-Ide-subset-ide)
        ultimately show ?thesis
          using Src-hd-eqI cong-filter-notIde by blast
      qed
      finally show ?thesis by blast
    qed
  qed
  moreover have Λ.seq (M ∘ N) u
    by (metis u Srcs-simpΛP Arr.simps(2) Trgs.simps(2) seq-char
        list.sel(1) seq Λ.seqI(1) Λ.sources-charΛ)
  moreover have ¬ Ide (N # filter notIde (map Λ.un-App2 (u # U)))
    using u *
    by (metis (no-types, lifting) *** Arr-map-un-App2 Std Std-imp-Arr
        Arr.simps(1) Ide.elims(2) Resid-Arr-Ide-ind ide-char
        seq-char calculation(1) cong-filter-notIde filter-notIde-Ide
        insert-subset list.discI list.sel(3) list.simps(15) mem-Collect-eq)
  ultimately show ?thesis
    using * 1 2 3 Y ind8 Std-implies-set-subset-elementary-reduction
    by simp
qed
show ?thesis
proof (cases Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide;
       cases Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide)
  show ⟦Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide;
         Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide⟧
           ⟹ ?thesis
  proof −
    assume *: Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide
    assume **: Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide
    have False
      using u 5 * ** Ide-iff-standard-development-empty
      by (metis Λ.Ide.simps(4) image-subset-iff Λ.lambda.collapse(3)
          list.set-intros(1) mem-Collect-eq)
    thus ?thesis by blast
```

```
qed
show ⟦Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide;
      ¬ Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide⟧
       ⟹ ?thesis
proof −
  assume ∗: Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide
  assume ∗∗: ¬ Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide
  have 6: Λ.Trg (Λ.un-App1 (last (u # U))) = Λ.Trg M
  proof −
    have Λ.Trg M = Λ.Src (hd (map Λ.un-App1 (u # U)))
     by (metis u seq Trg-last-Src-hd-eqI hd-map Λ.Src.simps(4) Λ.Trg.simps(3)
         Λ.is-App-def Λ.lambda.sel(3) last-ConsL list.discI list.sel(1))
    also have ... = Λ.Trg (last (map Λ.un-App1 (u # U)))
    proof −
      have 6: Ide (map Λ.un-App1 (u # U))
        using ∗ ∗∗∗ u Std Std-imp-Arr Ide-char ide-char Arr-map-un-App1
        by (metis (mono-tags, lifting) Collect-cong insert-subset
            Λ.ide-char list.distinct(1) list.set-map list.simps(15)
            mem-Collect-eq)
      hence Src (map Λ.un-App1 (u # U)) = Trg (map Λ.un-App1 (u # U))
        using Ide-imp-Src-eq-Trg by blast
      thus ?thesis
        using 6 Ide-implies-Arr by force
    qed
    also have ... = Λ.Trg (Λ.un-App1 (last (u # U)))
     by (simp add: last-map)
    finally show ?thesis by simp
  qed
  have filter notIde (map Λ.un-App1 (u # U)) = []
    using ∗ by (simp add: subset-eq)
  hence 4: stdz-insert (M ∘ N) (u # U) =
           map (λX. X ∘ Λ.Src N) (standard-development M) @
           map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
               (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))
    using u 4 5 ∗ ∗∗ Ide-iff-standard-development-empty MN
    by simp
  show ?thesis
  proof (intro conjI)
    have Std (map (λX. X ∘ Λ.Src N) (standard-development M) @
              map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))))
    proof (intro Std-append)
      show Std (map (λX. X ∘ Λ.Src N) (standard-development M))
        using Std-map-App1 Std-standard-development MN Λ.Ide-Src
        by force
      show Std (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                    (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))))
        using ∗∗ B MN 6 Std-map-App2 Λ.Ide-Trg by presburger
      show map (λX. X ∘ Λ.Src N) (standard-development M) = [] ∨
```

```
      map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
        (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))) = [] ∨
      Λ.sseq (last (map (λX. X ∘ Λ.Src N) (standard-development M)))
          (hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                (stdz-insert N (filter notIde
                                  (map Λ.un-App2 (u # U))))))
proof (cases Λ.Ide M)
  show Λ.Ide M ⟹ ?thesis
    using Ide-iff-standard-development-empty MN by blast
  assume M: ¬ Λ.Ide M
  have Λ.sseq (last (map (λX. X ∘ Λ.Src N) (standard-development M)))
            (hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (stdz-insert N (filter notIde
                                    (map Λ.un-App2 (u # U))))))
  proof −
    have last (map (λX. X ∘ Λ.Src N) (standard-development M)) =
          Λ.App (last (standard-development M)) (Λ.Src N)
      using M
      by (simp add: Ide-iff-standard-development-empty MN last-map)
   moreover have hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                       (stdz-insert N (filter notIde
                                         (map Λ.un-App2 (u # U))))) =
                 Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))
                   (hd (stdz-insert N (filter notIde
                                         (map Λ.un-App2 (u # U)))))
      by (metis ⋆⋆ B Ide.simps(1) Resid.simps(2) hd-map ide-char)
    moreover
    have Λ.sseq (Λ.App (last (standard-development M)) (Λ.Src N))
                ...
    proof −
      have Λ.elementary-reduction (last (standard-development M))
        using M MN Std-standard-development
              Ide-iff-standard-development-empty last-in-set
              mem-Collect-eq set-standard-development subsetD
        by metis
      moreover have Λ.elementary-reduction
                      (hd (stdz-insert N
                            (filter notIde (map Λ.un-App2 (u # U)))))
        using ⋆⋆ B
        by (metis Arr.simps(1) Con-implies-Arr(2) Ide.simps(1)
            ide-char in-mono list.set-sel(1) mem-Collect-eq)
      moreover have Λ.Trg (last (standard-development M)) =
                      Λ.Trg (Λ.un-App1 (last (u # U)))
        using M MN 6 Trg-last-standard-development by presburger
      moreover have Λ.Src N =
                      Λ.Src (hd (stdz-insert N
                                  (filter notIde (map Λ.un-App2 (u # U)))))
        by (metis ⋆⋆ B Src-hd-eqI list.sel(1))
      ultimately show ?thesis
```

```
        by simp
      qed
      ultimately show ?thesis by simp
    qed
    thus ?thesis by blast
  qed
qed
thus Std (stdz-insert (M ∘ N) (u # U))
  using 4 by simp
show ¬ Ide ((M ∘ N) # u # U) ⟶
       stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
proof
  show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
  proof (cases Λ.Ide M)
    assume M: Λ.Ide M
    have stdz-insert (M ∘ N) (u # U) =
         map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
             (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))
      using 4 M MN Ide-iff-standard-development-empty by simp
    also have ... *∼* (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                           (N # filter notIde (map Λ.un-App2 (u # U))))
    proof −
      have Λ.Ide (Λ.Trg (Λ.un-App1 (last (u # U))))
        using M 6 Λ.Ide-Trg Λ.Ide-implies-Arr by fastforce
      thus ?thesis
        using ** *** B u cong-map-App1 by blast
    qed
    also have map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (N # filter notIde (map Λ.un-App2 (u # U))) =
              map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (filter notIde (N # map Λ.un-App2 (u # U)))
      using 1 M by force
    also have map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (filter notIde (N # map Λ.un-App2 (u # U))) *∼*
              map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (N # map Λ.un-App2 (u # U))
    proof −
      have Arr (N # map Λ.un-App2 (u # U))
      proof
        show Λ.arr N
          using MN by blast
        show Arr (map Λ.un-App2 (u # U))
          using *** u Std Arr-map-un-App2
          by (metis Std-imp-Arr insert-subset list.distinct(1)
              list.simps(15) mem-Collect-eq)
        show Λ.trg N = Src (map Λ.un-App2 (u # U))
          using u ‹Λ.seq (M ∘ N) u› Λ.seq-char Λ.is-App-def by auto
      qed
      moreover have ¬ Ide (N # map Λ.un-App2 (u # U))
```

```
    using 1 M by force
  moreover have Λ.Ide (Λ.Trg (Λ.un-App1 (last (u # U))))
    using M 6 Λ.Ide-Trg Λ.Ide-implies-Arr by presburger
  ultimately show ?thesis
    using cong-filter-notIde cong-map-App1 by blast
qed
also have map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
              (N # map Λ.un-App2 (u # U)) =
          map (Λ.App M) (N # map Λ.un-App2 (u # U))
  using M MN ‹Λ.Trg (Λ.un-App1 (last (u # U))) = Λ.Trg M›
        Λ.Ide-iff-Trg-self
  by force
also have ... = (M ∘ N) # map (Λ.App M) (map Λ.un-App2 (u # U))
  by simp
also have ... = (M ∘ N) # u # U
proof −
  have Arr (u # U)
    using Std Std-imp-Arr by blast
  moreover have set (u # U) ⊆ Collect Λ.is-App
    using *** u by simp
  moreover have Λ.un-App1 u = M
    by (metis ∗ u M seq Trg-last-Src-hd-eqI Λ.Ide-iff-Src-self
        Λ.Ide-iff-Trg-self Λ.Ide-implies-Arr Λ.Src.simps(4)
        Λ.Trg.simps(3) Λ.lambda.collapse(3) Λ.lambda.sel(3)
        last.simps list.distinct(1) list.sel(1) list.set-intros(1)
        list.set-map list.simps(9) mem-Collect-eq standardize.cases
        subset-iff)
  moreover have Λ.un-App1 ‘ set (u # U) ⊆ {M}
  proof −
    have Ide (map Λ.un-App1 (u # U))
      using ∗ *** Std Std-imp-Arr Arr-map-un-App1
      by (metis Collect-cong Ide-char calculation(1−2) Λ.ide-char
          list.set-map)
    thus ?thesis
      by (metis calculation(3) hd-map list.discI list.sel(1)
          list.set-map set-Ide-subset-single-hd)
  qed
  ultimately show ?thesis
    using M map-App-map-un-App2 by blast
qed
finally show ?thesis by blast
next
assume M: ¬ Λ.Ide M
have stdz-insert (M ∘ N) (u # U) =
     map (λX. X ∘ Λ.Src N) (standard-development M) @
     map (λX. Λ.Trg M ∘ X)
         (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))
  using 4 6 by simp
also have ... *∼* [M ∘ Λ.Src N] @ [Λ.Trg M ∘ N] @
```

```
                  map (λX. Λ.Trg M ∘ X)
                    (filter notIde (map Λ.un-App2 (u # U)))
proof (intro cong-append)
  show map (λX. X ∘ Λ.Src N) (standard-development M) *∼*
       [M ∘ Λ.Src N]
    using MN M cong-standard-development Λ.Ide-Src
          cong-map-App2 [of Λ.Src N standard-development M [M]]
    by simp
  show map (λX. Λ.Trg M ∘ X)
         (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))) *∼*
       [Λ.Trg M ∘ N] @
         map (λX. Λ.Trg M ∘ X)
           (filter notIde (map Λ.un-App2 (u # U)))
  proof −
    have map (λX. Λ.Trg M ∘ X)
           (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))) *∼*
         map (λX. Λ.Trg M ∘ X)
           (N # filter notIde (map Λ.un-App2 (u # U)))
      using ** B MN cong-map-App1 lambda-calculus.Ide-Trg
      by presburger
    also have map (λX. Λ.Trg M ∘ X)
                (N # filter notIde (map Λ.un-App2 (u # U))) =
              [Λ.Trg M ∘ N] @
                map (λX. Λ.Trg M ∘ X)
                  (filter notIde (map Λ.un-App2 (u # U)))
      by simp
    finally show ?thesis by blast
  qed
  show seq (map (λX. X ∘ Λ.Src N) (standard-development M))
           (map (λX. Λ.Trg M ∘ X)
              (stdz-insert N (filter notIde
                                (map Λ.un-App2 (u # U)))))
    using MN M ** B cong-standard-development [of M]
    by (metis Nil-is-append-conv Resid.simps(2) Std-imp-Arr
        ‹Std (stdz-insert (M ∘ N) (u # U))› arr-append-imp-seq
        arr-char calculation complete-development-Ide-iff
        complete-development-def list.map-disc-iff development.simps(1))
qed
also have [M ∘ Λ.Src N] @ [Λ.Trg M ∘ N] @
            map (λX. Λ.Trg M ∘ X)
              (filter notIde (map Λ.un-App2 (u # U))) =
          ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N]) @
            map (λX. Λ.Trg M ∘ X)
              (filter notIde (map Λ.un-App2 (u # U)))
  by simp
also have ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N]) @
            map (λX. Λ.Trg M ∘ X)
              (filter notIde (map Λ.un-App2 (u # U))) *∼*
          ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N]) @
```

```
            map (λX. Λ.Trg M ∘ X) (map Λ.un-App2 (u # U))
proof (intro cong-append)
  show seq ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])
           (map (λX. Λ.Trg M ∘ X)
                (filter notIde (map Λ.un-App2 (u # U))))
  proof
    show Arr ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])
      by (simp add: MN)
    show 9: Arr (map (λX. Λ.Trg M ∘ X)
                     (filter notIde (map Λ.un-App2 (u # U))))
    proof −
      have Arr (map Λ.un-App2 (u # U))
        using *** u Arr-map-un-App2
        by (metis Std Std-imp-Arr list.distinct(1) mem-Collect-eq
            set-ConsD subset-code(1))
      moreover have ¬ Ide (map Λ.un-App2 (u # U))
        using **
        by (metis Collect-cong Λ.ide-char list.set-map
            set-Ide-subset-ide)
      ultimately show ?thesis
        using cong-filter-notIde
        by (metis Arr-map-App2 Con-implies-Arr(2) Ide.simps(1)
            MN ide-char Λ.Ide-Trg)
    qed
    show Λ.Trg (last ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])) =
         Λ.Src (hd (map (λX. Λ.Trg M ∘ X)
                        (filter notIde (map Λ.un-App2 (u # U)))))
    proof −
      have Λ.Trg (last ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])) =
           Λ.Trg M ∘ Λ.Trg N
        using MN by auto
      also have ... = Λ.Src u
        using Trg-last-Src-hd-eqI seq by force
      also have ... = Λ.Src (Λ.Trg M ∘ Λ.un-App2 u)
        using MN ‹Λ.App (Λ.Trg M) (Λ.Trg N) = Λ.Src u› u by auto
      also have 8: ... = Λ.Trg M ∘ Λ.Src (Λ.un-App2 u)
        using MN by simp
      also have 7: ... = Λ.Trg M ∘
                         Λ.Src (hd (filter notIde
                                    (map Λ.un-App2 (u # U))))
        using u 5 list.simps(9) cong-filter-notIde
              ‹filter notIde (map Λ.un-App1 (u # U)) = []›
        by auto
      also have ... = Λ.Src (hd (map (λX. Λ.Trg M ∘ X)
                                     (filter notIde
                                       (map Λ.un-App2 (u # U)))))

        by (metis 7 8 9 Arr.simps(1) hd-map Λ.Src.simps(4)
            Λ.lambda.sel(4) list.simps(8))
```

```
      finally show Λ.Trg (last ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])) =
                   Λ.Src (hd (map (λX. Λ.Trg M ∘ X)
                                  (filter notIde
                                          (map Λ.un-App2 (u # U)))))
        by blast
    qed
  qed
  show seq [M ∘ Λ.Src N] [Λ.Trg M ∘ N]
    using MN by force
  show [M ∘ Λ.Src N] *∼* [M ∘ Λ.Src N]
    using MN
    by (meson head-redex-decomp Λ.Arr.simps(4) Λ.Arr-Src
        prfx-transitive)
  show [Λ.Trg M ∘ N] *∼* [Λ.Trg M ∘ N]
    using MN
    by (meson ‹seq [M ∘ Λ.Src N] [Λ.Trg M ∘ N]› cong-reflexive seqE)
  show map (λX. Λ.Trg M ∘ X)
           (filter notIde (map Λ.un-App2 (u # U))) *∼*
       map (λX. Λ.Trg M ∘ X) (map Λ.un-App2 (u # U))
  proof −
    have Arr (map Λ.un-App2 (u # U))
      using *** u Arr-map-un-App2
      by (metis Std Std-imp-Arr list.distinct(1) mem-Collect-eq
          set-ConsD subset-code(1))
    moreover have ¬ Ide (map Λ.un-App2 (u # U))
      using **
      by (metis Collect-cong Λ.ide-char list.set-map
          set-Ide-subset-ide)
    ultimately show ?thesis
      using M MN cong-filter-notIde cong-map-App1 Λ.Ide-Trg
      by presburger
  qed
qed
also have ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N]) @
            map (λX. Λ.Trg M ∘ X) (map Λ.un-App2 (u # U)) *∼*
          [M ∘ N] @ u # U
proof (intro cong-append)
  show seq ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])
           (map (λX. Λ.Trg M ∘ X) (map Λ.un-App2 (u # U)))
    by (metis Nil-is-append-conv Nil-is-map-conv arr-append-imp-seq
        calculation cong-implies-coterminal coterminalE
        list.distinct(1))
  show [M ∘ Λ.Src N] @ [Λ.Trg M ∘ N] *∼* [M ∘ N]
   using MN Λ.resid-Arr-self Λ.Arr-not-Nil Λ.Ide-Trg ide-char by simp
 show  map (λX. Λ.Trg M ∘ X) (map Λ.un-App2 (u # U)) *∼* u # U
  proof −
    have map (λX. Λ.Trg M ∘ X) (map Λ.un-App2 (u # U)) = u # U
    proof (intro map-App-map-un-App2)
      show Arr (u # U)
```

```
            using Std Std-imp-Arr by blast
          show set (u # U) ⊆ Collect Λ.is-App
            using *** u by auto
          show Λ.Ide (Λ.Trg M)
            using MN Λ.Ide-Trg by blast
          show Λ.un-App1 ‘ set (u # U) ⊆ {Λ.Trg M}
          proof −
            have Λ.un-App1 u = Λ.Trg M
              using * u seq seq-char
              apply (cases u)
                apply simp-all
              by (metis Trg-last-Src-hd-eqI Λ.Ide-iff-Src-self
                  Λ.Src-Src Λ.Src-Trg Λ.Src-eq-iff(2) Λ.Trg.simps(3)
                  last-ConsL list.sel(1) seq u)
            moreover have Ide (map Λ.un-App1 (u # U))
              using * Std Std-imp-Arr Arr-map-un-App1
              by (metis Collect-cong Ide-char
                  ‹Arr (u # U)› ‹set (u # U) ⊆ Collect Λ.is-App›
                  Λ.ide-char list.set-map)
            ultimately show ?thesis
              using set-Ide-subset-single-hd by force
          qed
        qed
        thus ?thesis
          by (simp add: Resid-Arr-self Std ide-char)
      qed
    qed
    also have [M ∘ N] @ u # U = (M ∘ N) # u # U
      by simp
    finally show ?thesis by blast
  qed
 qed
qed
show ⟦¬ Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide;
       Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide⟧
      ⟹ ?thesis
proof −
  assume *: ¬ Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide
  assume **: Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide
  have 10: filter notIde (map Λ.un-App2 (u # U)) = []
    using ** by (simp add: subset-eq)
  hence 4: stdz-insert (M ∘ N) (u # U) =
           map (λX. X ∘ Λ.Src N)
               (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) @
           map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
               (standard-development N)
    using u 4 5 * ** Ide-iff-standard-development-empty MN
    by simp
```

```
have 6: Λ.Ide (Λ.Trg (Λ.un-App1 (last (u # U))))
  using *** u Std Std-imp-Arr
  by (metis Arr-imp-arr-last in-mono Λ.Arr.simps(4) Λ.Ide-Trg Λ.arr-char
      Λ.lambda.collapse(3) last.simps last-in-set list.discI mem-Collect-eq)
show ?thesis
proof (intro conjI)
  show Std (stdz-insert (M ∘ N) (u # U))
  proof −
    have Std (map (λX. X ∘ Λ.Src N)
                  (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) @
              map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (standard-development N))
    proof (intro Std-append)
      show Std (map (λX. X ∘ Λ.Src N)
                    (stdz-insert M (filter notIde
                                           (map Λ.un-App1 (u # U)))))
        using * A MN Std-map-App1 Λ.Ide-Src by presburger
      show Std (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                    (standard-development N))
        using MN 6 Std-map-App2 Std-standard-development by simp
      show map (λX. X ∘ Λ.Src N)
               (stdz-insert M
                 (filter notIde (map Λ.un-App1 (u # U)))) = [] ∨
           map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
               (standard-development N) = [] ∨
           Λ.sseq (last (map (λX. Λ.App X (Λ.Src N))
                             (stdz-insert M
                               (filter notIde (map Λ.un-App1 (u # U))))))
                  (hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                           (standard-development N)))
      proof (cases Λ.Ide N)
        show Λ.Ide N ⟹ ?thesis
          using Ide-iff-standard-development-empty MN by blast
        assume N: ¬ Λ.Ide N
        have Λ.sseq (last (map (λX. X ∘ Λ.Src N)
                               (stdz-insert M
                                 (filter notIde (map Λ.un-App1 (u # U))))))
                    (hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                             (standard-development N)))
        proof −
          have hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                       (standard-development N)) =
                Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))
                      (hd (standard-development N))
            by (meson Ide-iff-standard-development-empty MN N list.map-sel(1))
          moreover have last (map (λX. X ∘ Λ.Src N)
                                  (stdz-insert M
                                    (filter notIde (map Λ.un-App1 (u # U))))) =
                        Λ.App (last (stdz-insert M
```

```
                                (filter notIde
                                    (map Λ.un-App1 (u # U)))))
                        (Λ.Src N)
    by (metis ∗ A Ide.simps(1) Resid.simps(1) ide-char last-map)
moreover have Λ.sseq ... (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))
                                (hd (standard-development N)))
  proof −
    have 7: Λ.elementary-reduction
              (last (stdz-insert M (filter notIde
                                    (map Λ.un-App1 (u # U)))))
      using ∗ A
      by (metis Ide.simps(1) Resid.simps(2) ide-char last-in-set
          mem-Collect-eq subset-iff)
    moreover
    have Λ.elementary-reduction (hd (standard-development N))
      using MN N hd-in-set set-standard-development
            Ide-iff-standard-development-empty
      by blast
    moreover have Λ.Src N = Λ.Src (hd (standard-development N))
      using MN N Src-hd-standard-development by auto
    moreover have Λ.Trg (last (stdz-insert M
                                (filter notIde
                                    (map Λ.un-App1 (u # U))))) =
                  Λ.Trg (Λ.un-App1 (last (u # U)))
    proof −
      have [Λ.Trg (last (stdz-insert M
                          (filter notIde
                              (map Λ.un-App1 (u # U)))))] =
           [Λ.Trg (Λ.un-App1 (last (u # U)))]
      proof −
        have Λ.Trg (last (stdz-insert M
                           (filter notIde
                               (map Λ.un-App1 (u # U))))) =
             Λ.Trg (last (map Λ.un-App1 (u # U)))
        proof −
          have Λ.Trg (last (stdz-insert M
                             (filter notIde (map Λ.un-App1 (u # U))))) =
               Λ.Trg (last (M # filter notIde (map Λ.un-App1 (u # U))))
            using ∗ A Trg-last-eqI by blast
          also have ... = Λ.Trg (last ([M] @ filter notIde
                                        (map Λ.un-App1 (u # U))))
            by simp
          also have ... = Λ.Trg (last (filter notIde
                                        (map Λ.un-App1 (u # U))))
          proof −
            have seq [M] (filter notIde (map Λ.un-App1 (u # U)))
            proof
              show Arr [M]
                using MN by simp
```

```
      show Arr (filter notIde (map Λ.un-App1 (u # U)))
        using ∗ Std-imp-Arr
        by (metis (no-types, lifting)
          X empty-filter-conv list.set-map mem-Collect-eq subsetI)
      show Λ.Trg (last [M]) =
           Λ.Src (hd (filter notIde (map Λ.un-App1 (u # U))))
      proof −
        have Λ.Trg (last [M]) = Λ.Trg M
          using MN by simp
        also have ... = Λ.Src (Λ.un-App1 u)
          by (metis Trg-last-Src-hd-eqI Λ.Src.simps(4)
              Λ.Trg.simps(3) Λ.lambda.collapse(3)
              Λ.lambda.inject(3) last-ConsL list.sel(1) seq u)
        also have ... = Λ.Src (hd (map Λ.un-App1 (u # U)))
          by auto
        also have ... = Λ.Src (hd (filter notIde
                                   (map Λ.un-App1 (u # U))))
          using u 5 10 by force
        finally show ?thesis by blast
      qed
    qed
    thus ?thesis
      by (metis Arr.simps(1) last-appendR seq-char)
  qed
  also have ... = Λ.Trg (last (map Λ.un-App1 (u # U)))
  proof −
    have filter (λu. ¬ Λ.Ide u) (map Λ.un-App1 (u # U)) *∼*
         map Λ.un-App1 (u # U)
    using ∗ ∗∗∗ u Std Std-imp-Arr Arr-map-un-App1 [of u # U]
          cong-filter-notIde
      by (metis (mono-tags, lifting) empty-filter-conv
          filter-notIde-Ide list.discI list.set-map
          mem-Collect-eq set-ConsD subset-code(1))
    thus ?thesis
      using cong-implies-coterminal Trg-last-eqI
      by presburger
  qed
  finally show ?thesis by blast
qed
thus ?thesis
  by (simp add: last-map)
qed
moreover
have Λ.Ide (Λ.Trg (last (stdz-insert M
                         (filter notIde
                                 (map Λ.un-App1 (u # U))))))
  using 7 Λ.Ide-Trg Λ.elementary-reduction-is-arr by blast
moreover have Λ.Ide (Λ.Trg (Λ.un-App1 (last (u # U))))
  using 6 by blast
```

```
        ultimately show ?thesis by simp
      qed
      ultimately show ?thesis
        using Λ.sseq.simps(4) by blast
    qed
    ultimately show ?thesis by argo
  qed
  thus ?thesis by blast
 qed
 thus ?thesis
  using 4 by simp
qed
show ¬ Ide ((M ∘ N) # u # U) ⟶
       stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
proof
 show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
 proof (cases Λ.Ide N)
  assume N: Λ.Ide N
  have stdz-insert (M ∘ N) (u # U) =
       map (λX. X ∘ N)
           (stdz-insert M (filter notIde
                           (map Λ.un-App1 (u # U))))
    using 4 N MN Ide-iff-standard-development-empty Λ.Ide-iff-Src-self
    by force
  also have ... *∼* map (λX. X ∘ N)
                       (M # filter notIde
                                  (map Λ.un-App1 (u # U)))
    using ∗ A MN N Λ.Ide-Src cong-map-App2 Λ.Ide-iff-Src-self
    by blast
  also have map (λX. X ∘ N)
                (M # filter notIde
                           (map Λ.un-App1 (u # U))) =
            [M ∘ N] @
              map (λX. Λ.App X N)
                  (filter notIde (map Λ.un-App1 (u # U)))
    by auto
  also have [M ∘ N] @
              map (λX. X ∘ N)
                  (filter notIde (map Λ.un-App1 (u # U))) *∼*
            [M ∘ N] @ map (λX. X ∘ N) (map Λ.un-App1 (u # U))
  proof (intro cong-append)
    show seq [M ∘ N]
             (map (λX. X ∘ N)
                  (filter notIde (map Λ.un-App1 (u # U))))
    proof
      have 20: Arr (map Λ.un-App1 (u # U))
        using ∗∗∗ u Std Arr-map-un-App1
        by (metis Std-imp-Arr insert-subset list.discI list.simps(15)
```

```
      mem-Collect-eq)
  show Arr [M ∘ N]
    using MN by auto
  show 21: Arr (map (λX. X ∘ N)
                    (filter notIde (map Λ.un-App1 (u # U))))
  proof −
    have Arr (filter notIde (map Λ.un-App1 (u # U)))
      using u 20 cong-filter-notIde
      by (metis (no-types, lifting) ∗ Std-imp-Arr
          ‹Std (filter notIde (map Λ.un-App1 (u # U)))›
          empty-filter-conv list.set-map mem-Collect-eq subsetI)
    thus ?thesis
      using MN N Arr-map-App1 Λ.Ide-Src by presburger
  qed
  show Λ.Trg (last [M ∘ N]) =
       Λ.Src (hd (map (λX. X ∘ N)
                      (filter notIde (map Λ.un-App1 (u # U)))))
  proof −
    have Λ.Trg (last [M ∘ N]) = Λ.Trg M ∘ N
      using MN N Λ.Ide-iff-Trg-self by simp
    also have ... = Λ.Src (Λ.un-App1 u) ∘ N
      using MN u seq seq-char
      by (metis Trg-last-Src-hd-eqI calculation Λ.Src-Src Λ.Src-Trg
          Λ.Src-eq-iff(2) Λ.is-App-def Λ.lambda.sel(3) list.sel(1))
    also have ... = Λ.Src (Λ.un-App1 u ∘ N)
      using MN N Λ.Ide-iff-Src-self by simp
    also have ... = Λ.Src (hd (map (λX. X ∘ N)
                                   (map Λ.un-App1 (u # U))))
      by simp
    also have ... = Λ.Src (hd (map (λX. X ∘ N)
                                   (filter notIde
                                           (map Λ.un-App1 (u # U)))))
    proof −
      have cong (map Λ.un-App1 (u # U))
                (filter notIde (map Λ.un-App1 (u # U)))
        using ∗ 20 21 cong-filter-notIde
        by (metis Arr.simps(1) filter-notIde-Ide map-is-Nil-conv)
      thus ?thesis
        by (metis (no-types, lifting) Ide.simps(1) Resid.simps(2)
            Src-hd-eqI hd-map ide-char Λ.Src.simps(4)
            list.distinct(1) list.simps(9))
    qed
    finally show ?thesis by blast
  qed
qed
show cong [M ∘ N] [M ∘ N]
  using MN
  by (meson head-redex-decomp Λ.Arr.simps(4) Λ.Arr-Src
      prfx-transitive)
```

```
  show map (λX. X ∘ N) (filter notIde (map Λ.un-App1 (u # U))) *∼*
       map (λX. X ∘ N) (map Λ.un-App1 (u # U))
  proof −
    have Arr (map Λ.un-App1 (u # U))
      using *** u Std Arr-map-un-App1
      by (metis Std-imp-Arr insert-subset list.discI list.simps(15)
          mem-Collect-eq)
    moreover have ¬ Ide (map Λ.un-App1 (u # U))
      using *
      by (metis Collect-cong Λ.ide-char list.set-map
          set-Ide-subset-ide)
    ultimately show ?thesis
      using *** u MN N cong-filter-notIde cong-map-App2
      by (meson Λ.Ide-Src)
  qed
qed
also have [M ∘ N] @ map (λX. X ∘ N) (map Λ.un-App1 (u # U)) *∼*
          [M ∘ N] @ u # U
proof −
  have map (λX. X ∘ N) (map Λ.un-App1 (u # U)) *∼* u # U
  proof −
    have map (λX. X ∘ N) (map Λ.un-App1 (u # U)) = u # U
    proof (intro map-App-map-un-App1)
      show Arr (u # U)
        using Std Std-imp-Arr by simp
      show set (u # U) ⊆ Collect Λ.is-App
        using *** u by auto
      show Λ.Ide N
        using N by simp
      show Λ.un-App2 ‘ set (u # U) ⊆ {N}
      proof −
        have Λ.Src (Λ.un-App2 u) = Λ.Trg N
          using ** seq u seq-char N
          apply (cases u)
            apply simp-all
          by (metis Trg-last-Src-hd-eqI Λ.Src.simps(4) Λ.Trg.simps(3)
              Λ.lambda.inject(3) last-ConsL list.sel(1) seq)
        moreover have Λ.Ide (Λ.un-App2 u) ∧ Λ.Ide N
          using ** N by simp
        moreover have Ide (map Λ.un-App2 (u # U))
          using ** Std Std-imp-Arr Arr-map-un-App2
          by (metis Collect-cong Ide-char
              ‹Arr (u # U)› ‹set (u # U) ⊆ Collect Λ.is-App›
              Λ.ide-char list.set-map)
        ultimately show ?thesis
          by (metis hd-map Λ.Ide-iff-Src-self Λ.Ide-iff-Trg-self
              Λ.Ide-implies-Arr list.discI list.sel(1)
              list.set-map set-Ide-subset-single-hd)
      qed
```

```
    qed
    thus ?thesis
      by (simp add: Resid-Arr-self Std ide-char)
  qed
  thus ?thesis
    using MN cong-append
    by (metis (no-types, lifting) 1 cong-standard-development
        cong-transitive Λ.Arr.simps(4) seq)
qed
also have [M ∘ N] @ (u # U) = (M ∘ N) # u # U
  by simp
finally show ?thesis by blast
next
assume N: ¬ Λ.Ide N
have stdz-insert (M ∘ N) (u # U) =
      map (λX. X ∘ Λ.Src N)
          (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) @
      map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
          (standard-development N)
  using 4 by simp
also have ... *∼* map (λX. X ∘ Λ.Src N)
                     (M # filter notIde (map Λ.un-App1 (u # U))) @
                     map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N]
proof (intro cong-append)
  show 23: map (λX. X ∘ Λ.Src N)
             (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) *∼*
           map (λX. X ∘ Λ.Src N)
               (M # filter notIde (map Λ.un-App1 (u # U)))
    using ∗ A MN Λ.Ide-Src cong-map-App2 by blast
  show 22: map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
               (standard-development N) *∼*
           map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N]
    using 6 ∗∗∗ u Std Std-imp-Arr MN N cong-standard-development
          cong-map-App1
    by presburger
  show seq (map (λX. X ∘ Λ.Src N)
                (stdz-insert M (filter notIde
                                 (map Λ.un-App1 (u # U)))))
           (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                (standard-development N))
  proof −
    have seq (map (λX. X ∘ Λ.Src N)
                  (M # filter notIde
                               (map Λ.un-App1 (u # U))))
             (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N])
    proof
      show 26: Arr (map (λX. X ∘ Λ.Src N)
                        (M # filter notIde
                                     (map Λ.un-App1 (u # U))))
```

```
  by (metis 23 Con-implies-Arr(2) Ide.simps(1) ide-char)
show Arr (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N])
  by (meson 22 arr-char con-implies-arr(2) prfx-implies-con)
show Λ.Trg (last (map (λX. X ∘ Λ.Src N)
                      (M # filter notIde
                                (map Λ.un-App1 (u # U))))) =
     Λ.Src (hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                    [N]))
proof −
  have Λ.Trg (last (map (λX. X ∘ Λ.Src N)
                        (M # map Λ.un-App1 (u # U))))
       ∼
       Λ.Trg (last (map (λX. X ∘ Λ.Src N)
                        (M # filter notIde
                                  (map Λ.un-App1 (u # U)))))
  proof −
    have targets (map (λX. X ∘ Λ.Src N)
                      (M # filter notIde
                                (map Λ.un-App1 (u # U)))) =
         targets (map (λX. X ∘ Λ.Src N)
                      (M # map Λ.un-App1 (u # U)))
    proof −
      have map (λX. X ∘ Λ.Src N)
               (M # filter notIde (map Λ.un-App1 (u # U))) *∼*
           map (λX. X ∘ Λ.Src N)
               (M # map Λ.un-App1 (u # U))
      proof −
        have map (λX. X ∘ Λ.Src N)
                 (M # map Λ.un-App1 (u # U)) =
             map (λX. X ∘ Λ.Src N)
                 ([M] @ map Λ.un-App1 (u # U))
          by simp
        also have cong ... (map (λX. X ∘ Λ.Src N)
                                ([M] @ filter notIde
                                       (map Λ.un-App1 (u # U))))
        proof −
          have [M] @ map Λ.un-App1 (u # U) *∼*
               [M] @ filter notIde
                           (map Λ.un-App1 (u # U))
          proof (intro cong-append)
            show cong [M] [M]
              using MN
              by (meson head-redex-decomp prfx-transitive)
            show seq [M] (map Λ.un-App1 (u # U))
            proof
              show Arr [M]
                using MN by simp
              show Arr (map Λ.un-App1 (u # U))
                using ⋆⋆⋆ u Std Arr-map-un-App1
```

```
            by (metis Std-imp-Arr insert-subset list.discI
                list.simps(15) mem-Collect-eq)
          show Λ.Trg (last [M]) =
                Λ.Src (hd (map Λ.un-App1 (u # U)))
            using MN u seq seq-char Srcs-simpΛP by auto
        qed
        show cong (map Λ.un-App1 (u # U))
                  (filter notIde
                          (map Λ.un-App1 (u # U)))
        proof −
          have Arr (map Λ.un-App1 (u # U))
            by (metis *** Arr-map-un-App1 Std Std-imp-Arr
                insert-subset list.discI list.simps(15)
                mem-Collect-eq u)
          moreover have ¬ Ide (map Λ.un-App1 (u # U))
            using * set-Ide-subset-ide by fastforce
          ultimately show ?thesis
            using cong-filter-notIde by blast
        qed
      qed
      thus map (λX. X ∘ Λ.Src N)
               ([M] @ map Λ.un-App1 (u # U)) *∼*
           map (λX. X ∘ Λ.Src N)
               ([M] @ filter notIde (map Λ.un-App1 (u # U)))
        using MN cong-map-App2 Λ.Ide-Src by presburger
    qed
    finally show ?thesis by simp
  qed
  thus ?thesis
    using cong-implies-coterminal by blast
qed
moreover have [Λ.Trg (last (map (λX. X ∘ Λ.Src N)
                                 (M # map Λ.un-App1 (u # U))))] ∈
              targets (map (λX. X ∘ Λ.Src N)
                           (M # map Λ.un-App1 (u # U)))
  by (metis (no-types, lifting) 26 calculation mem-Collect-eq
      single-Trg-last-in-targets targets-charΛP)
moreover have [Λ.Trg (last (map (λX. X ∘ Λ.Src N)
                                (M # filter notIde
                                     (map Λ.un-App1 (u # U)))))] ∈
              targets (map (λX. X ∘ Λ.Src N)
                           (M # filter notIde
                                (map Λ.un-App1 (u # U))))
  using 26 single-Trg-last-in-targets by blast
ultimately show ?thesis
  by (metis (no-types, lifting) 26 Ide.simps(1−2) Resid-rec(1)
      in-targets-iff ide-char)
qed
moreover have Λ.Ide (Λ.Trg (last (map (λX. X ∘ Λ.Src N)
```

```
                                        (M # map Λ.un-App1 (u # U)))))
        by (metis 6 MN Λ.Ide.simps(4) Λ.Ide-Src Λ.Trg.simps(3)
            Λ.Trg-Src last-ConsR last-map list.distinct(1)
            list.simps(9))
      moreover have Λ.Ide (Λ.Trg (last (map (λX. X ∘ Λ.Src N)
                                            (M # filter notIde
                                               (map Λ.un-App1 (u # U))))))
        using Λ.ide-backward-stable calculation(1−2) by fast
      ultimately show ?thesis
        by (metis (no-types, lifting) 6 MN hd-map
            Λ.Ide-iff-Src-self Λ.Ide-implies-Arr Λ.Src.simps(4)
            Λ.Trg.simps(3) Λ.Trg-Src Λ.cong-Ide-are-eq
            last.simps last-map list.distinct(1) list.map-disc-iff
            list.sel(1))
    qed
  qed
  thus ?thesis
    using 22 23 cong-respects-seqP by presburger
 qed
qed
also have map (λX. X ∘ Λ.Src N)
            (M # filter notIde (map Λ.un-App1 (u # U))) @
          map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N] =
         [M ∘ Λ.Src N] @
          map (λX. X ∘ Λ.Src N)
            (filter notIde (map Λ.un-App1 (u # U))) @
           [Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))) N]
 by simp
also have 1: [M ∘ Λ.Src N] @
             map (λX. X ∘ Λ.Src N)
               (filter notIde (map Λ.un-App1 (u # U))) @
              [Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))) N] *∼*
            [M ∘ Λ.Src N] @
             map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U)) @
              [Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))) N]
proof (intro cong-append)
 show [M ∘ Λ.Src N] *∼* [M ∘ Λ.Src N]
   using MN
   by (meson head-redex-decomp lambda-calculus.Arr.simps(4)
       lambda-calculus.Arr-Src prfx-transitive)
 show 21: map (λX. X ∘ Λ.Src N)
            (filter notIde (map Λ.un-App1 (u # U))) *∼*
          map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U))
 proof −
   have filter notIde (map Λ.un-App1 (u # U)) *∼*
        map Λ.un-App1 (u # U)
   proof −
     have ¬ Ide (map Λ.un-App1 (u # U))
       using ∗
```

```
      by (metis Collect-cong Λ.ide-char list.set-map
          set-Ide-subset-ide)
    thus ?thesis
      using ⋆⋆⋆ u Std Std-imp-Arr Arr-map-un-App1
          cong-filter-notIde
      by (metis ‹¬ Ide (map Λ.un-App1 (u # U))›
          list.distinct(1) mem-Collect-eq set-ConsD
          subset-code(1))
  qed
  thus ?thesis
    using MN cong-map-App2 [of Λ.Src N] Λ.Ide-Src by presburger
qed
show [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N] *∼*
     [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N]
  by (metis 6 Con-implies-Arr(1) MN Λ.Ide-implies-Arr arr-char
      cong-reflexive Λ.Ide-iff-Src-self neq-Nil-conv
      orthogonal-App-single-single(1))
show seq (map (λX. X ∘ Λ.Src N)
              (filter notIde (map Λ.un-App1 (u # U))))
         [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N]
proof
  show Arr (map (λX. X ∘ Λ.Src N)
                (filter notIde (map Λ.un-App1 (u # U))))
    by (metis 21 Con-implies-Arr(2) Ide.simps(1) ide-char)
  show Arr [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N]
    by (metis Con-implies-Arr(2) Ide.simps(1)
        ‹[Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N] *∼*
         [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N]›
        ide-char)
  show Λ.Trg (last (map (λX. X ∘ Λ.Src N)
                        (filter notIde
                                (map Λ.un-App1 (u # U))))) =
       Λ.Src (hd [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N])
    by (metis (no-types, lifting) 6 21 MN Trg-last-eqI
        Λ.Ide-iff-Src-self Λ.Ide-implies-Arr Λ.Src.simps(4)
        Λ.Trg.simps(3) Λ.Trg-Src last-map list.distinct(1)
        list.map-disc-iff list.sel(1))
qed
show seq [M ∘ Λ.Src N]
         (map (λX. X ∘ Λ.Src N)
              (filter notIde (map Λ.un-App1 (u # U))) @
          [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N])
proof
  show Arr [M ∘ Λ.Src N]
    using MN by simp
  show Arr (map (λX. X ∘ Λ.Src N)
                (filter notIde (map Λ.un-App1 (u # U))) @
            [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N])
    apply (intro Arr-appendI_P)
```

```
      apply (metis 21 Con-implies-Arr(2) Ide.simps(1) ide-char)
     apply (metis Con-implies-Arr(1) Ide.simps(1)
       ‹[Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N] *∼*
        [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N]› ide-char)
    by (metis (no-types, lifting) 21 Arr.simps(1)
        Arr-append-iff_P Con-implies-Arr(2) Ide.simps(1)
        append-is-Nil-conv calculation ide-char not-Cons-self2)
  show Λ.Trg (last [M ∘ Λ.Src N]) =
       Λ.Src (hd (map (λX. X ∘ Λ.Src N)
                      (filter notIde
                              (map Λ.un-App1 (u # U))) @
                        [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N]))
    by (metis (no-types, lifting) Con-implies-Arr(2) Ide.simps(1)
        Trg-last-Src-hd-eqI append-is-Nil-conv arr-append-imp-seq
        arr-char calculation ide-char not-Cons-self2)
  qed
qed
also have [M ∘ Λ.Src N] @
          map (λX. X ∘ Λ.Src N)(map Λ.un-App1 (u # U)) @
            [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N] *∼*
          [M ∘ Λ.Src N] @
            [Λ.Trg M ∘ N] @
              map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U))
proof (intro cong-append [of [Λ.App M (Λ.Src N)]])
  show seq [M ∘ Λ.Src N]
           (map (λX. X ∘ Λ.Src N)
                (map Λ.un-App1 (u # U)) @
                  [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N])
  proof
    show Arr [M ∘ Λ.Src N]
      using MN by simp
    show Arr (map (λX. X ∘ Λ.Src N)
                  (map Λ.un-App1 (u # U)) @
                    [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N])
      by (metis (no-types, lifting) 1 Con-append(2) Con-implies-Arr(2)
          Ide.simps(1) append-is-Nil-conv ide-char not-Cons-self2)
    show Λ.Trg (last [M ∘ Λ.Src N]) =
         Λ.Src (hd (map (λX. X ∘ Λ.Src N)
                        (map Λ.un-App1 (u # U)) @
                          [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N]))
    proof −
      have Λ.Trg M = Λ.Src (Λ.un-App1 u)
        using u seq
        by (metis Trg-last-Src-hd-eqI Λ.Src.simps(4) Λ.Trg.simps(3)
            Λ.lambda.collapse(3) Λ.lambda.inject(3) last-ConsL
            list.sel(1))
      thus ?thesis
        using MN by auto
    qed
```

```
  qed
  show [M ∘ Λ.Src N] *∼* [M ∘ Λ.Src N]
    using MN
    by (metis head-redex-decomp Λ.Arr.simps(4) Λ.Arr-Src
        prfx-transitive)
  show map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U)) @
         [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N] *∼*
       [Λ.Trg M ∘ N] @
         map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U))
  proof −
    have map (λX. X ∘ Λ.Src (hd [N])) (map Λ.un-App1 (u # U)) @
          map (Λ.App (Λ.Trg (last (map Λ.un-App1 (u # U))))) [N] *∼*
         map (Λ.App (Λ.Src (hd (map Λ.un-App1 (u # U))))) [N] @
           map  (λX. X ∘ Λ.Trg (last [N])) (map Λ.un-App1 (u # U))
    proof −
      have Arr (map Λ.un-App1 (u # U))
        using Std *** u Arr-map-un-App1
        by (metis Std-imp-Arr insert-subset list.discI list.simps(15)
            mem-Collect-eq)
      moreover have Arr [N]
        using MN by simp
      ultimately show ?thesis
        using orthogonal-App-cong by blast
    qed
    moreover
    have map (Λ.App (Λ.Src (hd (map Λ.un-App1 (u # U))))) [N] =
         [Λ.Trg M ∘ N]
      by (metis Trg-last-Src-hd-eqI lambda-calculus.Src.simps(4)
          Λ.Trg.simps(3) Λ.lambda.collapse(3) Λ.lambda.sel(3)
          last-ConsL list.sel(1) list.simps(8) list.simps(9) seq u)
    moreover have [Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N] =
                  map (Λ.App (Λ.Trg (last (map Λ.un-App1 (u # U))))) [N]
      by (simp add: last-map)
    ultimately show ?thesis
      using last-map by auto
  qed
qed
also have [M ∘ Λ.Src N] @
            [Λ.Trg M ∘ N] @
              map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U)) =
          ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N]) @
            map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U))
  by simp
also have ... *∼* [M ∘ N] @ (u # U)
proof (intro cong-append)
  show [M ∘ Λ.Src N] @ [Λ.Trg M ∘ N] *∼* [M ∘ N]
    using MN Λ.resid-Arr-self Λ.Arr-not-Nil Λ.Ide-Trg ide-char
    by auto
show 1: map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U)) *∼* u # U
```

```
  proof −
    have map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U)) = u # U
    proof (intro map-App-map-un-App1)
      show Arr (u # U)
        using Std Std-imp-Arr by simp
      show set (u # U) ⊆ Collect Λ.is-App
        using *** u by auto
      show Λ.Ide (Λ.Trg N)
        using MN Λ.Ide-Trg by simp
      show Λ.un-App2 ‘ set (u # U) ⊆ {Λ.Trg N}
      proof −
        have Λ.Src (Λ.un-App2 u) = Λ.Trg N
          using u seq seq-char
          apply (cases u)
             apply simp-all
          by (metis Trg-last-Src-hd-eqI Λ.Src.simps(4) Λ.Trg.simps(3)
              Λ.lambda.inject(3) last-ConsL list.sel(1) seq)
        moreover have Λ.Ide (Λ.un-App2 u)
          using ** by simp
        moreover have Ide (map Λ.un-App2 (u # U))
          using ** Std Std-imp-Arr Arr-map-un-App2
          by (metis Collect-cong Ide-char
              ‹Arr (u # U)› ‹set (u # U) ⊆ Collect Λ.is-App›
              Λ.ide-char list.set-map)
        ultimately show ?thesis
          by (metis Λ.Ide-iff-Src-self Λ.Ide-implies-Arr list.sel(1)
              list.set-map list.simps(9) set-Ide-subset-single-hd
              singleton-insert-inj-eq)
      qed
    qed
    thus ?thesis
      by (simp add: Resid-Arr-self Std ide-char)
  qed
  show seq ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])
           (map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U)))
  proof
    show Arr ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])
      using MN by simp
    show Arr (map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U)))
      using MN Std Std-imp-Arr Arr-map-un-App1 Arr-map-App1
      by (metis 1 Con-implies-Arr(1) Ide.simps(1) ide-char)
    show Λ.Trg (last ([M ∘ Λ.Src N] @ [Λ.Trg M ∘ N])) =
         Λ.Src (hd (map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U))))
      using MN Std Std-imp-Arr Arr-map-un-App1 Arr-map-App1
            seq seq-char u Srcs-simpΛP by auto
  qed
qed
also have [M ∘ N] @ (u # U) = (M ∘ N) # u # U
  by simp
```

```
      finally show ?thesis by blast
    qed
  qed
 qed
qed
show ⟦¬ Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide;
       ¬ Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide⟧
        ⟹ ?thesis
proof −
 assume ∗: ¬ Λ.un-App1 ‘ set (u # U) ⊆ Collect Λ.Ide
 assume ∗∗: ¬ Λ.un-App2 ‘ set (u # U) ⊆ Collect Λ.Ide
 show ?thesis
 proof (intro conjI)
  show Std (stdz-insert (M ∘ N) (u # U))
  proof −
   have Std (map (λX. X ∘ Λ.Src N)
                 (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) @
             map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                 (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))))
   proof (intro Std-append)
    show Std (map (λX. X ∘ Λ.Src N)
                  (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))))
     using ∗ A Λ.Ide-Src MN Std-map-App1 by presburger
    show Std (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))))
    proof −
     have Λ.Arr (Λ.un-App1 (last (u # U)))
      by (metis ∗∗∗ Λ.Arr.simps(4) Std Std-imp-Arr Arr.simps(2)
          Arr-append-iff_P append-butlast-last-id append-self-conv2
          Λ.arr-char Λ.lambda.collapse(3) last.simps last-in-set
          list.discI mem-Collect-eq subset-code(1) u)
     thus ?thesis
      using ∗∗ B Λ.Ide-Trg MN Std-map-App2 by presburger
    qed
    show map (λX. X ∘ Λ.Src N)
             (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) = [] ∨
         map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
             (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))) = [] ∨
         Λ.sseq (last (map (λX. X ∘ Λ.Src N)
                           (stdz-insert M (filter notIde (map Λ.un-App1 (u # U))))))
                (hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                         (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))))
    proof −
     have Λ.sseq (last (map (λX. X ∘ Λ.Src N)
                            (stdz-insert M (filter notIde (map Λ.un-App1 (u # U))))))
                 (hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                          (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))))
     proof −
      let ?M = Λ.un-App1 (last (map (λX. X ∘ Λ.Src N)
```

```
                    (stdz-insert M
                      (filter notIde
                        (map Λ.un-App1 (u # U))))))
let ?M' = Λ.Trg (Λ.un-App1 (last (u # U)))
let ?N = Λ.Src N
let ?N' = Λ.un-App2
            (hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (stdz-insert N
                    (filter notIde
                      (map Λ.un-App2 (u # U))))))
have M: ?M = last (stdz-insert M
                    (filter notIde (map Λ.un-App1 (u # U))))
  by (metis ∗ A Ide.simps(1) Resid.simps(1) ide-char
      Λ.lambda.sel(3) last-map)
have N': ?N' = hd (stdz-insert N
                    (filter notIde (map Λ.un-App2 (u # U))))
  by (metis ∗∗ B Ide.simps(1) Resid.simps(2) ide-char
      Λ.lambda.sel(4) hd-map)
have AppMN: last (map (λX. X ∘ Λ.Src N)
                  (stdz-insert M
                    (filter notIde (map Λ.un-App1 (u # U))))) =
          ?M ∘ ?N
  by (metis ∗ A Ide.simps(1) M Resid.simps(2) ide-char last-map)
moreover
have 4: hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                (stdz-insert N
                  (filter notIde (map Λ.un-App2 (u # U))))) =
        ?M' ∘ ?N'
  by (metis (no-types, lifting) ∗∗ B Resid.simps(2) con-char
      prfx-implies-con Λ.lambda.collapse(3) Λ.lambda.discI(3)
      Λ.lambda.inject(3) list.map-sel(1))
moreover have MM: Λ.elementary-reduction ?M
  by (metis ∗ A Arr.simps(1) Con-implies-Arr(2) Ide.simps(1)
      M ide-char in-mono last-in-set mem-Collect-eq)
moreover have NN': Λ.elementary-reduction ?N'
  using ∗∗ B N'
  by (metis Arr.simps(1) Con-implies-Arr(2) Ide.simps(1)
      ide-char in-mono list.set-sel(1) mem-Collect-eq)
moreover have Λ.Trg ?M = ?M'
proof −
  have 1: [Λ.Trg ?M] *∼* [?M']
  proof −
    have [Λ.Trg ?M] *∼*
         [Λ.Trg (last (M # filter notIde (map Λ.un-App1 (u # U))))]
    proof −
      have targets (stdz-insert M
                      (filter notIde (map Λ.un-App1 (u # U)))) =
           targets (M # filter notIde (map Λ.un-App1 (u # U)))
        using ∗ A cong-implies-coterminal by blast
```

```
    moreover
  have [Λ.Trg (last (M # filter notIde (map Λ.un-App1 (u # U))))]
         ∈ targets (M # filter notIde (map Λ.un-App1 (u # U)))
      by (metis (no-types, lifting) ∗ A Λ.Arr-Trg Λ.Ide-Trg
          Arr.simps(2) Arr-append-iff_P Arr-iff-Con-self
          Con-implies-Arr(2) Ide.simps(1) Ide.simps(2)
          Resid-Arr-Ide-ind ide-char append-butlast-last-id
          append-self-conv2 Λ.arr-char in-targets-iff Λ.ide-char
          list.discI)
    ultimately show ?thesis
      using ∗ A M in-targets-iff
      by (metis (no-types, lifting) Con-implies-Arr(1)
          con-char prfx-implies-con in-targets-iff)
  qed
  also have 2: [Λ.Trg (last (M # filter notIde
                                (map Λ.un-App1 (u # U))))] *∼*
             [Λ.Trg (last (filter notIde
                             (map Λ.un-App1 (u # U))))]
    by (metis (no-types, lifting) ∗ prfx-transitive
        calculation empty-filter-conv last-ConsR list.set-map
        mem-Collect-eq subsetI)
  also have [Λ.Trg (last (filter notIde
                            (map Λ.un-App1 (u # U))))] *∼*
          [Λ.Trg (last (map Λ.un-App1 (u # U)))]
  proof −
    have map Λ.un-App1 (u # U) *∼*
         filter notIde (map Λ.un-App1 (u # U))
      by (metis (mono-tags, lifting) ∗ *** Arr-map-un-App1
          Std Std-imp-Arr cong-filter-notIde empty-filter-conv
          filter-notIde-Ide insert-subset list.discI list.set-map
          list.simps(15) mem-Collect-eq subsetI u)
    thus ?thesis
      by (metis 2 Trg-last-eqI prfx-transitive)
  qed
  also have [Λ.Trg (last (map Λ.un-App1 (u # U)))] = [?M']
    by (simp add: last-map)
  finally show ?thesis by blast
qed
have 3: Λ.Trg ?M = Λ.Trg ?M \ ?M'
  by (metis (no-types, lifting) 1 ∗ A M Con-implies-Arr(2)
      Ide.simps(1) Resid-Arr-Ide-ind Resid-rec(1)
      ide-char target-is-ide in-targets-iff list.inject)
also have ... = ?M'
  by (metis (no-types, lifting) 1 4 Arr.simps(2) Con-implies-Arr(2)
      Ide.simps(1) Ide.simps(2) MM NN' Resid-Arr-Ide-ind
      Resid-rec(1) Src-hd-eqI calculation ide-char
      Λ.Ide-iff-Src-self Λ.Src-Trg Λ.arr-char
      Λ.elementary-reduction.simps(4)
      Λ.elementary-reduction-App-iff Λ.elementary-reduction-is-arr
```

```
            Λ.elementary-reduction-not-ide Λ.lambda.discI(3)
            Λ.lambda.sel(3) list.sel(1))
        finally show ?thesis by blast
      qed
      moreover have ?N = Λ.Src ?N'
      proof −
        have 1: [Λ.Src ?N'] *∼* [?N]
        proof −
          have sources (stdz-insert N
                          (filter notIde (map Λ.un-App2 (u # U)))) =
               sources [N]
            using ** B
            by (metis Con-implies-Arr(2) Ide.simps(1) coinitialE
                cong-implies-coinitial ide-char sources-cons)
          thus ?thesis
            by (metis (no-types, lifting) AppMN ** B Λ.Ide-Src
                MM MN N' NN' Λ.Trg-Src Arr.simps(1) Arr.simps(2)
                Con-implies-Arr(1) Ide.simps(2) con-char ideE ide-char
                sources-cons Λ.arr-char in-targets-iff
              Λ.elementary-reduction.simps(4) Λ.elementary-reduction-App-iff
                Λ.elementary-reduction-is-arr Λ.elementary-reduction-not-ide
                Λ.lambda.disc(14) Λ.lambda.sel(4) last-ConsL list.exhaust-sel
                targets-single-Src)
        qed
        have Λ.Src ?N' = Λ.Src ?N' \ ?N
          by (metis (no-types, lifting) 1 MN Λ.Coinitial-iff-Con
              Λ.Ide-Src Arr.simps(2) Ide.simps(1) Ide-implies-Arr
              Resid-rec(1) ide-char Λ.not-arr-null Λ.null-char
              Λ.resid-Arr-Ide)
        also have ... = ?N
          by (metis 1 MN NN' Src-hd-eqI calculation Λ.Src-Src Λ.arr-char
              Λ.elementary-reduction-is-arr list.sel(1))
        finally show ?thesis by simp
      qed
      ultimately show ?thesis
        using u Λ.sseq.simps(4)
        by (metis (mono-tags, lifting))
    qed
    thus ?thesis by blast
  qed
qed
thus ?thesis
  using 4 by presburger
qed
show ¬ Ide ((M ∘ N) # u # U) ⟶
       stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
proof
  have stdz-insert (M ∘ N) (u # U) =
       map (λX. X ∘ Λ.Src N)
```

```
        (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) @
      map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
        (stdz-insert N (filter notIde (map Λ.un-App2 (u # U))))
  using 4 by simp
also have ... *∼* map (λX. X ⋆ Λ.Src N)
                    (M # map Λ.un-App1 (u # U)) @
                    map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                      (N # map Λ.un-App2 (u # U))
proof (intro cong-append)
  have X: stdz-insert M (filter notIde (map Λ.un-App1 (u # U))) *∼*
          M # map Λ.un-App1 (u # U)
  proof −
    have stdz-insert M (filter notIde (map Λ.un-App1 (u # U))) *∼*
         [M] @ filter notIde (map Λ.un-App1 (u # U))
      using ∗ A by simp
    also have [M] @ filter notIde (map Λ.un-App1 (u # U)) *∼*
              [M] @ map Λ.un-App1 (u # U)
    proof −
      have filter notIde (map Λ.un-App1 (u # U)) *∼*
           map Λ.un-App1 (u # U)
        using ∗ cong-filter-notIde
        by (metis (mono-tags, lifting) ∗∗∗ Arr-map-un-App1 Std
           Std-imp-Arr empty-filter-conv filter-notIde-Ide insert-subset
           list.discI list.set-map list.simps(15) mem-Collect-eq subsetI u)
      moreover have seq [M] (filter notIde (map Λ.un-App1 (u # U)))
        by (metis ∗ A Arr.simps(1) Con-implies-Arr(1) append-Cons
           append-Nil arr-append-imp-seq arr-char calculation
           ide-implies-arr list.discI)
      ultimately show ?thesis
        using cong-append cong-reflexive by blast
    qed
    also have [M] @ map Λ.un-App1 (u # U) =
              M # map Λ.un-App1 (u # U)
      by simp
    finally show ?thesis by blast
  qed
  have Y: stdz-insert N (filter notIde (map Λ.un-App2 (u # U))) *∼*
          N # map Λ.un-App2 (u # U)
  proof −
    have 5: stdz-insert N (filter notIde (map Λ.un-App2 (u # U))) *∼*
            [N] @ filter notIde (map Λ.un-App2 (u # U))
      using ∗∗ B by simp
    also have [N] @ filter notIde (map Λ.un-App2 (u # U)) *∼*
              [N] @ map Λ.un-App2 (u # U)
    proof −
      have filter notIde (map Λ.un-App2 (u # U)) *∼*
           map Λ.un-App2 (u # U)
        using ∗∗ cong-filter-notIde
        by (metis (mono-tags, lifting) ∗∗∗ Arr-map-un-App2 Std
```

```
        Std-imp-Arr empty-filter-conv filter-notIde-Ide insert-subset
        list.discI list.set-map list.simps(15) mem-Collect-eq subsetI u)
    moreover have seq [N] (filter notIde (map Λ.un-App2 (u # U)))
      by (metis 5 Arr.simps(1) Con-implies-Arr(2) Ide.simps(1)
        arr-append-imp-seq arr-char calculation ide-char not-Cons-self2)
    ultimately show ?thesis
      using cong-append cong-reflexive by blast
  qed
  also have [N] @ map Λ.un-App2 (u # U) =
            N # map Λ.un-App2 (u # U)
    by simp
  finally show ?thesis by blast
 qed
 show seq (map (λX. X ∘ Λ.Src N)
               (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))))
          (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
               (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))))
  by (metis 4 * ** A B Ide.simps(1) Nil-is-append-conv Nil-is-map-conv
      Resid.simps(1) Std-imp-Arr ‹Std (stdz-insert (M ∘ N) (u # U))›
      arr-append-imp-seq arr-char ide-char)
 show map (λX. X ∘ Λ.Src N)
          (stdz-insert M (filter notIde (map Λ.un-App1 (u # U)))) *∼*
      map (λX. X ∘ Λ.Src N) (M # map Λ.un-App1 (u # U))
  using X cong-map-App2 MN lambda-calculus.Ide-Src by presburger
 show map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
          (stdz-insert N (filter notIde (map Λ.un-App2 (u # U)))) *∼*
      map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
          (N # map Λ.un-App2 (u # U))
 proof −
  have set U ⊆ Collect Λ.Arr ∩ Collect Λ.is-App
    using *** Std Std-implies-set-subset-elementary-reduction
        Λ.elementary-reduction-is-arr
    by blast
  hence Λ.Ide (Λ.Trg (Λ.un-App1 (last (u # U))))
    by (metis inf.boundedE Λ.Arr.simps(4) Λ.Ide-Trg
      Λ.lambda.collapse(3) last.simps last-in-set mem-Collect-eq
      subset-eq u)
  thus ?thesis
    using Y cong-map-App1 by blast
 qed
qed
also have map (λX. X ∘ Λ.Src N) (M # map Λ.un-App1 (u # U)) @
          map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
              (N # map Λ.un-App2 (u # U)) *∼*
        [M ∘ N] @ [u] @ U
proof −
 have (map (λX. X ∘ Λ.Src N) (M # map Λ.un-App1 (u # U)) @
       map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
           (N # map Λ.un-App2 (u # U))) =
```

```
      ([M ∘ Λ.Src N] @
        map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U))) @
      ([Λ.Trg (Λ.un-App1 (last (u # U))) ∘ N] @
        map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
          (map Λ.un-App2 (u # U)))
  by simp
also have ... = [M ∘ Λ.Src N] @
                  (map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U)) @
                 map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N]) @
                  map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                    (map Λ.un-App2 (u # U))
  by auto
also have ... *∼* [M ∘ Λ.Src N] @
                    (map (Λ.App (Λ.Src (Λ.un-App1 u))) [N] @
                    map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U))) @
                      map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                        (map Λ.un-App2 (u # U))
proof −

  have (map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U)) @
          map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N]) @
         map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
           (map Λ.un-App2 (u # U)) *∼*
       (map (Λ.App (Λ.Src (Λ.un-App1 u))) [N] @
          map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U))) @
          map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                (map Λ.un-App2 (u # U))
  proof −
    have 1: Arr (map Λ.un-App1 (u # U))
      using u ***
      by (metis Arr-map-un-App1 Std Std-imp-Arr list.discI
        mem-Collect-eq set-ConsD subset-code(1))
    have map (λX. Λ.App X (Λ.Src N)) (map Λ.un-App1 (u # U)) @
           map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N] *∼*
         map (Λ.App (Λ.Src (Λ.un-App1 u))) [N] @
           map (λX. Λ.App X (Λ.Trg N)) (map Λ.un-App1 (u # U))
    proof −
      have Arr [N]
        using MN by simp
      moreover have Λ.Trg (last (map Λ.un-App1 (u # U))) =
                    Λ.Trg (Λ.un-App1 (last (u # U)))
        by (simp add: last-map)
      ultimately show ?thesis
        using 1 orthogonal-App-cong [of map Λ.un-App1 (u # U) [N]]
        by simp
    qed
    moreover have seq (map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u #
U)) @
                       map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N])
```

```
                    (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                        (map Λ.un-App2 (u # U)))
  proof
    show Arr (map (λX. X ∘ Λ.Src N)
                  (map Λ.un-App1 (u # U)) @
              map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N])
      by (metis Con-implies-Arr(1) Ide.simps(1) calculation ide-char)
    show Arr (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (map Λ.un-App2 (u # U)))
      using u ***
      by (metis 1 Arr-imp-arr-last Arr-map-App2 Arr-map-un-App2
          Std Std-imp-Arr Λ.Ide-Trg Λ.arr-char last-map list.discI
          mem-Collect-eq set-ConsD subset-code(1))
    show Λ.Trg (last (map (λX. X ∘ Λ.Src N)
                          (map Λ.un-App1 (u # U)) @
                      map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                          [N])) =
         Λ.Src (hd (map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                        (map Λ.un-App2 (u # U))))
    proof −
      have 1: Λ.Arr (Λ.un-App1 u)
        using u Λ.is-App-def by force
      have 2: U ≠ [] ⟹ Λ.Arr (Λ.un-App1 (last U))
        by (metis *** Arr-imp-arr-last Arr-map-un-App1
            ‹U ≠ [] ⟹ Arr U› Λ.arr-char last-map)
      have 3: Λ.Trg N = Λ.Src (Λ.un-App2 u)
        by (metis Trg-last-Src-hd-eqI Λ.Src.simps(4) Λ.Trg.simps(3)
            Λ.lambda.collapse(3) Λ.lambda.inject(3) last-ConsL
            list.sel(1) seq u)
      show ?thesis
        using u *** seq 1 2 3
        by (cases U = []) auto
    qed
  qed
  moreover have map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                    (map Λ.un-App2 (u # U)) *∼*
                map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                    (map Λ.un-App2 (u # U))
    using calculation(2) cong-reflexive by blast
  ultimately show ?thesis
    using cong-append by blast
qed
moreover have seq [M ∘ Λ.Src N]
                  ((map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U)) @
                  map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N]) @
                    map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                        (map Λ.un-App2 (u # U)))
proof
  show Arr [M ∘ Λ.Src N]
```

```
      using MN by simp
    show Arr ((map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U)) @
                map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N]) @
                map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                (map Λ.un-App2 (u # U)))
      using MN u seq
      by (metis Con-implies-Arr(1) Ide.simps(1) calculation ide-char)
    show Λ.Trg (last [M ∘ Λ.Src N]) =
       Λ.Src (hd ((map (λX. X ∘ Λ.Src N) (map Λ.un-App1 (u # U)) @
                  map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U))))) [N]) @
                    map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                      (map Λ.un-App2 (u # U))))
      using MN u seq seq-char Srcs-simpΛP
      by (cases u) auto
  qed
  ultimately show ?thesis
    using cong-append
    by (meson Resid-Arr-self ide-char seq-char)
qed
also have [M ∘ Λ.Src N] @
            (map (Λ.App (Λ.Src (Λ.un-App1 u))) [N] @
            map (λX. Λ.App X (Λ.Trg N)) (map Λ.un-App1 (u # U))) @
              map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                (map Λ.un-App2 (u # U)) =
           ([M ∘ Λ.Src N] @ [Λ.Src (Λ.un-App1 u) ∘ N]) @
            (map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U))) @
              map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                (map Λ.un-App2 (u # U))
  by simp
also have ... *∼* ([M ∘ N] @ [u] @ U)
proof −
  have [M ∘ Λ.Src N] @ [Λ.Src (Λ.un-App1 u) ∘ N] *∼* [M ∘ N]
  proof −
    have Λ.Src (Λ.un-App1 u) = Λ.Trg M
      by (metis Trg-last-Src-hd-eqI Λ.Src.simps(4) Λ.Trg.simps(3)
          Λ.lambda.collapse(3) Λ.lambda.inject(3) last.simps
          list.sel(1) seq u)
    thus ?thesis
      using MN u seq seq-char Λ.Arr-not-Nil Λ.resid-Arr-self ide-char
            Λ.Ide-Trg
      by simp
  qed
 moreover have map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U)) @
                map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                  (map Λ.un-App2 (u # U)) *∼*
               [u] @ U
  proof −
    have Arr ([u] @ U)
      by (simp add: Std)
```

```
      moreover have set ([u] @ U) ⊆ Collect Λ.is-App
        using *** u by auto
      moreover have Λ.Src (Λ.un-App2 (hd ([u] @ U))) = Λ.Trg N
      proof −
        have Λ.Ide (Λ.Trg N)
          using MN lambda-calculus.Ide-Trg by presburger
        moreover have Λ.Ide (Λ.Src (Λ.un-App2 (hd ([u] @ U))))
          by (metis Std Std-implies-set-subset-elementary-reduction
              Λ.Ide-Src Λ.arr-iff-has-source Λ.ide-implies-arr
              ‹set ([u] @ U) ⊆ Collect Λ.is-App› append-Cons
              Λ.elementary-reduction-App-iff Λ.elementary-reduction-is-arr
              Λ.sources-char_Λ list.sel(1) list.set-intros(1)
              mem-Collect-eq subset-code(1))
        moreover have Λ.Src (Λ.Trg N) =
                      Λ.Src (Λ.Src (Λ.un-App2 (hd ([u] @ U))))
        proof −
          have Λ.Src (Λ.Trg N) = Λ.Trg N
            using MN by simp
          also have ... = Λ.Src (Λ.un-App2 u)
            using u seq seq-char Srcs-simp_ΛP
            by (cases u) auto
          also have ... = Λ.Src (Λ.Src (Λ.un-App2 (hd ([u] @ U))))
            by (metis Λ.Ide-iff-Src-self Λ.Ide-implies-Arr
                ‹Λ.Ide (Λ.Src (Λ.un-App2 (hd ([u] @ U))))›
                append-Cons list.sel(1))
          finally show ?thesis by blast
        qed
        ultimately show ?thesis
          by (metis Λ.Ide-iff-Src-self Λ.Ide-implies-Arr)
      qed
      ultimately show ?thesis
        using map-App-decomp
        by (metis append-Cons append-Nil)
    qed
    moreover have seq ([M ∘ Λ.Src N] @ [Λ.Src (Λ.un-App1 u) ∘ N])
                    (map (λX. X ∘ Λ.Trg N) (map Λ.un-App1 (u # U)) @
                      map (Λ.App (Λ.Trg (Λ.un-App1 (last (u # U)))))
                          (map Λ.un-App2 (u # U)))
      using calculation(1−2) cong-respects-seq_P seq by auto
    ultimately show ?thesis
      using cong-append by presburger
  qed
  finally show ?thesis by blast
qed
also have [M ∘ N] @ [u] @ U = (M ∘ N) # u # U
  by simp
finally show stdz-insert (M ∘ N) (u # U) *∼* (M ∘ N) # u # U
  by blast
qed
```

qed
qed
qed
qed
qed
qed
qed
qed
qed

The eight remaining subgoals are now trivial consequences of fact $*$. Unfortunately, I haven't found a way to discharge them without having to state each one of them explicitly.

show $\bigwedge N\ u\ U.\ [\![\Lambda.Ide\ (\sharp \circ N) \Longrightarrow ?P\ (hd\ (u\ \#\ U))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (\sharp \circ N);\ \Lambda.seq\ (\sharp \circ N)\ (hd\ (u\ \#\ U));$
$\Lambda.contains\text{-}head\text{-}reduction\ (\sharp \circ N);$
$\Lambda.Ide\ ((\sharp \circ N) \setminus \Lambda.head\text{-}redex\ (\sharp \circ N))]\!]$
$\Longrightarrow ?P\ (hd\ (u\ \#\ U))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (\sharp \circ N);\ \Lambda.seq\ (\sharp \circ N)\ (hd\ (u\ \#\ U));$
$\Lambda.contains\text{-}head\text{-}reduction\ (\sharp \circ N);$
$\neg\ \Lambda.Ide\ ((\sharp \circ N) \setminus \Lambda.head\text{-}redex\ (\sharp \circ N))]\!]$
$\Longrightarrow ?P\ ((\sharp \circ N) \setminus \Lambda.head\text{-}redex\ (\sharp \circ N))\ (u\ \#\ U);$
$[\![\neg\ \Lambda.Ide\ (\sharp \circ N);\ \Lambda.seq\ (\sharp \circ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (\sharp \circ N);$
$\Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U));$
$\Lambda.Ide\ ((\sharp \circ N) \setminus \Lambda.head\text{-}strategy\ (\sharp \circ N))]\!]$
$\Longrightarrow ?P\ (\Lambda.head\text{-}strategy\ (\sharp \circ N))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (\sharp \circ N);\ \Lambda.seq\ (\sharp \circ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (\sharp \circ N);$
$\Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.Ide\ ((\sharp \circ N) \setminus \Lambda.head\text{-}strategy\ (\sharp \circ N))]\!]$
$\Longrightarrow ?P\ (\Lambda.resid\ (\sharp \circ N)\ (\Lambda.head\text{-}strategy\ (\sharp \circ N)))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (\sharp \circ N);\ \Lambda.seq\ (\sharp \circ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (\sharp \circ N);$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U))]\!]$
$\Longrightarrow ?P\ \sharp\ (filter\ notIde\ (map\ \Lambda.un\text{-}App1\ (u\ \#\ U)));$
$[\![\neg\ \Lambda.Ide\ (\sharp \circ N);\ \Lambda.seq\ (\sharp \circ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (\sharp \circ N);$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U))]\!]$
$\Longrightarrow ?P\ N\ (filter\ notIde\ (map\ \Lambda.un\text{-}App2\ (u\ \#\ U)))]\!]$
$\Longrightarrow ?P\ (\sharp \circ N)\ (u\ \#\ U)$
using $*$ $\Lambda.lambda.disc(6)$ **by** *presburger*
show $\bigwedge x\ N\ u\ U.\ [\![\Lambda.Ide\ (\langle\!\langle x \rangle\!\rangle \circ N) \Longrightarrow ?P\ (hd\ (u\ \#\ U))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (\langle\!\langle x \rangle\!\rangle \circ N);\ \Lambda.seq\ (\langle\!\langle x \rangle\!\rangle \circ N)\ (hd\ (u\ \#\ U));$
$\Lambda.contains\text{-}head\text{-}reduction\ (\langle\!\langle x \rangle\!\rangle \circ N);$
$\Lambda.Ide\ ((\langle\!\langle x \rangle\!\rangle \circ N) \setminus \Lambda.head\text{-}redex\ (\langle\!\langle x \rangle\!\rangle \circ N))]\!]$
$\Longrightarrow ?P\ (hd\ (u\ \#\ U))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (\langle\!\langle x \rangle\!\rangle \circ N);\ \Lambda.seq\ (\langle\!\langle x \rangle\!\rangle \circ N)\ (hd\ (u\ \#\ U));$
$\Lambda.contains\text{-}head\text{-}reduction\ (\langle\!\langle x \rangle\!\rangle \circ N);$

$\neg\ \Lambda.Ide\ ((\ «x»\ \circ\ N)\ \backslash\ \Lambda.head\text{-}redex\ (\ «x»\ \circ\ N))\rrbracket$
$\Longrightarrow\ ?P\ ((\ «x»\ \circ\ N)\ \backslash\ \Lambda.head\text{-}redex\ (\ «x»\ \circ\ N))\ (u\ \#\ U);$
$[\![\neg\ \Lambda.Ide\ (\ «x»\ \circ\ N);\ \Lambda.seq\ (\ «x»\ \circ\ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (\ «x»\ \circ\ N);$
$\Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U));$
$\Lambda.Ide\ ((\ «x»\ \circ\ N)\ \backslash\ \Lambda.head\text{-}strategy\ (\ «x»\ \circ\ N))\rrbracket$
$\Longrightarrow\ ?P\ (\Lambda.head\text{-}strategy\ (\ «x»\ \circ\ N))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (\ «x»\ \circ\ N);\ \Lambda.seq\ (\ «x»\ \circ\ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (\ «x»\ \circ\ N);$
$\Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.Ide\ ((\ «x»\ \circ\ N)\ \backslash\ \Lambda.head\text{-}strategy\ (\ «x»\ \circ\ N))\rrbracket$
$\Longrightarrow\ ?P\ ((\ «x»\ \circ\ N)\ \backslash\ \Lambda.head\text{-}strategy\ (\ «x»\ \circ\ N))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (\ «x»\ \circ\ N);\ \Lambda.seq\ (\ «x»\ \circ\ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (\ «x»\ \circ\ N);$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ «x»\ (filter\ notIde\ (map\ \Lambda.un\text{-}App1\ (u\ \#\ U)));$
$[\![\neg\ \Lambda.Ide\ (\ «x»\ \circ\ N);\ \Lambda.seq\ (\ «x»\ \circ\ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (\ «x»\ \circ\ N);$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ N\ (filter\ notIde\ (map\ \Lambda.un\text{-}App2\ (u\ \#\ U)))\rrbracket$
$\Longrightarrow\ ?P\ (\ «x»\ \circ\ N)\ (u\ \#\ U)$

using $*\ \Lambda.lambda.disc(7)$ **by** *presburger*

show $\bigwedge M1\ M2\ N\ u\ U.\ [\![\Lambda.Ide\ (M1\ \circ\ M2\ \circ\ N)\ \Longrightarrow\ ?P\ (hd\ (u\ \#\ U))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (M1\ \circ\ M2\ \circ\ N);\ \Lambda.seq\ (M1\ \circ\ M2\ \circ\ N)\ (hd\ (u\ \#\ U));$
$\Lambda.contains\text{-}head\text{-}reduction\ (M1\ \circ\ M2\ \circ\ N);$
$\Lambda.Ide\ ((M1\ \circ\ M2\ \circ\ N)\ \backslash\ \Lambda.head\text{-}redex\ (M1\ \circ\ M2\ \circ\ N))\rrbracket$
$\Longrightarrow\ ?P\ (hd\ (u\ \#\ U))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (M1\ \circ\ M2\ \circ\ N);\ \Lambda.seq\ (M1\ \circ\ M2\ \circ\ N)\ (hd\ (u\ \#\ U));$
$\Lambda.contains\text{-}head\text{-}reduction\ (M1\ \circ\ M2\ \circ\ N);$
$\neg\ \Lambda.Ide\ ((M1\ \circ\ M2\ \circ\ N)\ \backslash\ \Lambda.head\text{-}redex\ (M1\ \circ\ M2\ \circ\ N))\rrbracket$
$\Longrightarrow\ ?P\ ((M1\ \circ\ M2\ \circ\ N)\ \backslash\ \Lambda.head\text{-}redex\ (M1\ \circ\ M2\ \circ\ N))\ (u\ \#\ U);$
$[\![\neg\ \Lambda.Ide\ (M1\ \circ\ M2\ \circ\ N);\ \Lambda.seq\ (M1\ \circ\ M2\ \circ\ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M1\ \circ\ M2\ \circ\ N);$
$\Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U));$
$\Lambda.Ide\ ((M1\ \circ\ M2\ \circ\ N)\ \backslash\ \Lambda.head\text{-}strategy\ (M1\ \circ\ M2\ \circ\ N))\rrbracket$
$\Longrightarrow\ ?P\ (\Lambda.head\text{-}strategy\ (M1\ \circ\ M2\ \circ\ N))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (M1\ \circ\ M2\ \circ\ N);\ \Lambda.seq\ (M1\ \circ\ M2\ \circ\ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M1\ \circ\ M2\ \circ\ N);$
$\Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.Ide\ ((M1\ \circ\ M2\ \circ\ N)\ \backslash\ \Lambda.head\text{-}strategy\ (M1\ \circ\ M2\ \circ\ N))\rrbracket$
$\Longrightarrow\ ?P\ ((M1\ \circ\ M2\ \circ\ N)\ \backslash\ \Lambda.head\text{-}strategy\ (M1\ \circ\ M2\ \circ\ N))\ (tl\ (u\ \#\ U));$
$[\![\neg\ \Lambda.Ide\ (M1\ \circ\ M2\ \circ\ N);\ \Lambda.seq\ (M1\ \circ\ M2\ \circ\ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M1\ \circ\ M2\ \circ\ N);$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ (M1\ \circ\ M2)\ (filter\ notIde\ (map\ \Lambda.un\text{-}App1\ (u\ \#\ U)));$
$[\![\neg\ \Lambda.Ide\ (M1\ \circ\ M2\ \circ\ N);\ \Lambda.seq\ (M1\ \circ\ M2\ \circ\ N)\ (hd\ (u\ \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M1\ \circ\ M2\ \circ\ N);$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (hd\ (u\ \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ N\ (filter\ notIde\ (map\ \Lambda.un\text{-}App2\ (u\ \#\ U)))\rrbracket$

$\Longrightarrow$ *?P* (*M1* $\circ$ *M2* $\circ$ *N*) (*u* # *U*)
using $*$ Λ.*lambda.disc*(*9*) **by** *presburger*
show $\bigwedge$*M1 M2 N u U*. ⟦Λ.*Ide* (**λ**[*M1*] • *M2* $\circ$ *N*) $\Longrightarrow$ *?P* (*hd* (*u* # *U*)) (*tl* (*u* # *U*));
⟦¬ Λ.*Ide* (**λ**[*M1*] • *M2* $\circ$ *N*); Λ.*seq* (**λ**[*M1*] • *M2* $\circ$ *N*) (*hd* (*u* # *U*));
Λ.*contains-head-reduction* (**λ**[*M1*] • *M2* $\circ$ *N*);
Λ.*Ide* ((**λ**[*M1*] • *M2* $\circ$ *N*) \ (Λ.*head-redex* (**λ**[*M1*] • *M2* $\circ$ *N*)))⟧
$\Longrightarrow$ *?P* (*hd* (*u* # *U*)) (*tl* (*u* # *U*));
⟦¬ Λ.*Ide* (**λ**[*M1*] • *M2* $\circ$ *N*); Λ.*seq* (**λ**[*M1*] • *M2* $\circ$ *N*) (*hd* (*u* # *U*));
Λ.*contains-head-reduction* (**λ**[*M1*] • *M2* $\circ$ *N*);
¬ Λ.*Ide* ((**λ**[*M1*] • *M2* $\circ$ *N*) \ (Λ.*head-redex* (**λ**[*M1*] • *M2* $\circ$ *N*)))⟧
$\Longrightarrow$ *?P* (Λ.*resid* (**λ**[*M1*] • *M2* $\circ$ *N*) (Λ.*head-redex* (**λ**[*M1*] • *M2* $\circ$ *N*)))
(*u* # *U*);
⟦¬ Λ.*Ide* (**λ**[*M1*] • *M2* $\circ$ *N*); Λ.*seq* (**λ**[*M1*] • *M2* $\circ$ *N*) (*hd* (*u* # *U*));
¬ Λ.*contains-head-reduction* (**λ**[*M1*] • *M2* $\circ$ *N*);
Λ.*contains-head-reduction* (*hd* (*u* # *U*));
Λ.*Ide* ((**λ**[*M1*] • *M2* $\circ$ *N*) \ Λ.*head-strategy* (**λ**[*M1*] • *M2* $\circ$ *N*))⟧
$\Longrightarrow$ *?P* (Λ.*head-strategy* (**λ**[*M1*] • *M2* $\circ$ *N*)) (*tl* (*u* # *U*));
⟦¬ Λ.*Ide* (**λ**[*M1*] • *M2* $\circ$ *N*); Λ.*seq* (**λ**[*M1*] • *M2* $\circ$ *N*) (*hd* (*u* # *U*));
¬ Λ.*contains-head-reduction* (**λ**[*M1*] • *M2* $\circ$ *N*);
Λ.*contains-head-reduction* (*hd* (*u* # *U*));
¬ Λ.*Ide* ((**λ**[*M1*] • *M2* $\circ$ *N*) \ Λ.*head-strategy* (**λ**[*M1*] • *M2* $\circ$ *N*))⟧
$\Longrightarrow$ *?P* ((**λ**[*M1*] • *M2* $\circ$ *N*) \ Λ.*head-strategy* (**λ**[*M1*] • *M2* $\circ$ *N*))
(*tl* (*u* # *U*));
⟦¬ Λ.*Ide* (**λ**[*M1*] • *M2* $\circ$ *N*); Λ.*seq* (**λ**[*M1*] • *M2* $\circ$ *N*) (*hd* (*u* # *U*));
¬ Λ.*contains-head-reduction* (**λ**[*M1*] • *M2* $\circ$ *N*);
¬ Λ.*contains-head-reduction* (*hd* (*u* # *U*))⟧
$\Longrightarrow$ *?P* (**λ**[*M1*] • *M2*) (*filter notIde* (*map* Λ.*un-App1* (*u* # *U*)));
⟦¬ Λ.*Ide* (**λ**[*M1*] • *M2* $\circ$ *N*); Λ.*seq* (**λ**[*M1*] • *M2* $\circ$ *N*) (*hd* (*u* # *U*));
¬ Λ.*contains-head-reduction* (**λ**[*M1*] • *M2* $\circ$ *N*);
¬ Λ.*contains-head-reduction* (*hd* (*u* # *U*))⟧
$\Longrightarrow$ *?P N* (*filter notIde* (*map* Λ.*un-App2* (*u* # *U*)))⟧
$\Longrightarrow$ *?P* (**λ**[*M1*] • *M2* $\circ$ *N*) (*u* # *U*)
using $*$ Λ.*lambda.disc*(*10*) **by** *presburger*
show $\bigwedge$*M N U*. ⟦Λ.*Ide* (*M* $\circ$ *N*) $\Longrightarrow$ *?P* (*hd* (♯ # *U*)) (*tl* (♯ # *U*));
⟦¬ Λ.*Ide* (*M* $\circ$ *N*); Λ.*seq* (*M* $\circ$ *N*) (*hd* (♯ # *U*));
Λ.*contains-head-reduction* (*M* $\circ$ *N*);
Λ.*Ide* ((*M* $\circ$ *N*) \ Λ.*head-redex* (*M* $\circ$ *N*))⟧
$\Longrightarrow$ *?P* (*hd* (♯ # *U*)) (*tl* (♯ # *U*));
⟦¬ Λ.*Ide* (*M* $\circ$ *N*); Λ.*seq* (*M* $\circ$ *N*) (*hd* (♯ # *U*));
Λ.*contains-head-reduction* (*M* $\circ$ *N*);
¬ Λ.*Ide* ((*M* $\circ$ *N*) \ Λ.*head-redex* (*M* $\circ$ *N*))⟧
$\Longrightarrow$ *?P* ((*M* $\circ$ *N*) \ Λ.*head-redex* (*M* $\circ$ *N*)) (♯ # *U*);
⟦¬ Λ.*Ide* (*M* $\circ$ *N*); Λ.*seq* (*M* $\circ$ *N*) (*hd* (♯ # *U*));
¬ Λ.*contains-head-reduction* (*M* $\circ$ *N*);
Λ.*contains-head-reduction* (*hd* (♯ # *U*));
Λ.*Ide* (Λ.*resid* (*M* $\circ$ *N*) (Λ.*head-strategy* (*M* $\circ$ *N*)))⟧
$\Longrightarrow$ *?P* (Λ.*head-strategy* (*M* $\circ$ *N*)) (*tl* (♯ # *U*));
⟦¬ Λ.*Ide* (*M* $\circ$ *N*); Λ.*seq* (*M* $\circ$ *N*) (*hd* (♯ # *U*));
¬ Λ.*contains-head-reduction* (*M* $\circ$ *N*);

$\Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (\mathit{hd}\ (\sharp\ \#\ U));$
$\neg\ \Lambda.\mathit{Ide}\ ((M \circ N) \setminus \Lambda.\mathit{head\text{-}strategy}\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ ((M \circ N) \setminus \Lambda.\mathit{head\text{-}strategy}\ (M \circ N))\ (\mathit{tl}\ (\sharp\ \#\ U));$
$\llbracket\neg\ \Lambda.\mathit{Ide}\ (M \circ N);\ \Lambda.\mathit{seq}\ (M \circ N)\ (\mathit{hd}\ (\sharp\ \#\ U));$
$\neg\ \Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (M \circ N);$
$\neg\ \Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (\mathit{hd}\ (\sharp\ \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ M\ (\mathit{filter}\ \mathit{notIde}\ (\mathit{map}\ \Lambda.\mathit{un\text{-}App1}\ (\sharp\ \#\ U)));$
$\llbracket\neg\ \Lambda.\mathit{Ide}\ (M \circ N);\ \Lambda.\mathit{seq}\ (M \circ N)\ (\mathit{hd}\ (\sharp\ \#\ U));$
$\neg\ \Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (M \circ N);$
$\neg\ \Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (\mathit{hd}\ (\sharp\ \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ N\ (\mathit{filter}\ \mathit{notIde}\ (\mathit{map}\ \Lambda.\mathit{un\text{-}App2}\ (\sharp\ \#\ U)))\rrbracket$
$\Longrightarrow\ ?P\ (M \circ N)\ (\sharp\ \#\ U)$

using $*$ $\Lambda.\mathit{lambda.disc}(16)$ **by** *presburger*

show $\bigwedge M\ N\ x\ U.\ \llbracket\Lambda.\mathit{Ide}\ (M \circ N) \Longrightarrow ?P\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U))\ (\mathit{tl}\ (\mathit{«x»}\ \#\ U));$
$\llbracket\neg\ \Lambda.\mathit{Ide}\ (M \circ N);\ \Lambda.\mathit{seq}\ (M \circ N)\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U));$
$\Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (M \circ N);$
$\Lambda.\mathit{Ide}\ ((M \circ N) \setminus \Lambda.\mathit{head\text{-}redex}\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U))\ (\mathit{tl}\ (\mathit{«x»}\ \#\ U));$
$\llbracket\neg\ \Lambda.\mathit{Ide}\ (M \circ N);\ \Lambda.\mathit{seq}\ (M \circ N)\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U));$
$\Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (M \circ N);$
$\neg\ \Lambda.\mathit{Ide}\ ((M \circ N) \setminus \Lambda.\mathit{head\text{-}redex}\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ ((M \circ N) \setminus \Lambda.\mathit{head\text{-}redex}\ (M \circ N))\ (\mathit{«x»}\ \#\ U);$
$\llbracket\neg\ \Lambda.\mathit{Ide}\ (M \circ N);\ \Lambda.\mathit{seq}\ (M \circ N)\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U));$
$\neg\ \Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (M \circ N);$
$\Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U));$
$\Lambda.\mathit{Ide}\ ((M \circ N) \setminus \Lambda.\mathit{head\text{-}strategy}\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ (\Lambda.\mathit{head\text{-}strategy}\ (M \circ N))\ (\mathit{tl}\ (\mathit{«x»}\ \#\ U));$
$\llbracket\neg\ \Lambda.\mathit{Ide}\ (M \circ N);\ \Lambda.\mathit{seq}\ (M \circ N)\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U));$
$\neg\ \Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (M \circ N);$
$\Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U));$
$\neg\ \Lambda.\mathit{Ide}\ ((M \circ N) \setminus \Lambda.\mathit{head\text{-}strategy}\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ ((M \circ N) \setminus \Lambda.\mathit{head\text{-}strategy}\ (M \circ N))\ (\mathit{tl}\ (\mathit{«x»}\ \#\ U));$
$\llbracket\neg\ \Lambda.\mathit{Ide}\ (M \circ N);\ \Lambda.\mathit{seq}\ (M \circ N)\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U));$
$\neg\ \Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (M \circ N);$
$\neg\ \Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ M\ (\mathit{filter}\ \mathit{notIde}\ (\mathit{map}\ \Lambda.\mathit{un\text{-}App1}\ (\mathit{«x»}\ \#\ U)));$
$\llbracket\neg\ \Lambda.\mathit{Ide}\ (M \circ N);\ \Lambda.\mathit{seq}\ (M \circ N)\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U));$
$\neg\ \Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (M \circ N);$
$\neg\ \Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (\mathit{hd}\ (\mathit{«x»}\ \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ N\ (\mathit{filter}\ \mathit{notIde}\ (\mathit{map}\ \Lambda.\mathit{un\text{-}App2}\ (\mathit{«x»}\ \#\ U)))\rrbracket$
$\Longrightarrow\ ?P\ (M \circ N)\ (\mathit{«x»}\ \#\ U)$

using $*$ $\Lambda.\mathit{lambda.disc}(17)$ **by** *presburger*

show $\bigwedge M\ N\ P\ U.\ \llbracket\Lambda.\mathit{Ide}\ (M \circ N) \Longrightarrow ?P\ (\mathit{hd}\ (\boldsymbol{\lambda}[P]\ \#\ U))\ (\mathit{tl}\ (\boldsymbol{\lambda}[P]\ \#\ U));$
$\llbracket\neg\ \Lambda.\mathit{Ide}\ (M \circ N);\ \Lambda.\mathit{seq}\ (M \circ N)\ (\mathit{hd}\ (\boldsymbol{\lambda}[P]\ \#\ U));$
$\Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (M \circ N);$
$\Lambda.\mathit{Ide}\ ((M \circ N) \setminus \Lambda.\mathit{head\text{-}redex}\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ (\mathit{hd}\ (\boldsymbol{\lambda}[P]\ \#\ U))\ (\mathit{tl}\ (\boldsymbol{\lambda}[P]\ \#\ U));$
$\llbracket\neg\ \Lambda.\mathit{Ide}\ (M \circ N);\ \Lambda.\mathit{seq}\ (M \circ N)\ (\mathit{hd}\ (\boldsymbol{\lambda}[P]\ \#\ U));$
$\Lambda.\mathit{contains\text{-}head\text{-}reduction}\ (M \circ N);$

$\neg\ \Lambda.Ide\ ((M \circ N) \setminus \Lambda.head\text{-}redex\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ ((M \circ N) \setminus \Lambda.head\text{-}redex\ (M \circ N))\ (\boldsymbol{\lambda}[P] \#\ U);$
$[\![\neg\ \Lambda.Ide\ (M \circ N);\ \Lambda.seq\ (M \circ N)\ (hd\ (\boldsymbol{\lambda}[P] \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M \circ N);$
$\Lambda.contains\text{-}head\text{-}reduction\ (hd\ (\boldsymbol{\lambda}[P] \#\ U));$
$\Lambda.Ide\ ((M \circ N) \setminus \Lambda.head\text{-}strategy\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ (\Lambda.head\text{-}strategy\ (M \circ N))\ (tl\ (\boldsymbol{\lambda}[P] \#\ U));$
$[\![\neg\ \Lambda.Ide\ (M \circ N);\ \Lambda.seq\ (M \circ N)\ (hd\ (\boldsymbol{\lambda}[P] \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M \circ N);$
$\Lambda.contains\text{-}head\text{-}reduction\ (hd\ (\boldsymbol{\lambda}[P] \#\ U));$
$\neg\ \Lambda.Ide\ ((M \circ N) \setminus \Lambda.head\text{-}strategy\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ (\Lambda.resid\ (M \circ N)\ (\Lambda.head\text{-}strategy\ (M \circ N)))\ (tl\ (\boldsymbol{\lambda}[P] \#\ U));$
$[\![\neg\ \Lambda.Ide\ (M \circ N);\ \Lambda.seq\ (M \circ N)\ (hd\ (\boldsymbol{\lambda}[P] \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M \circ N);$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (hd\ (\boldsymbol{\lambda}[P] \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ M\ (filter\ notIde\ (map\ \Lambda.un\text{-}App1\ (\boldsymbol{\lambda}[P] \#\ U)));$
$[\![\neg\ \Lambda.Ide\ (M \circ N);\ \Lambda.seq\ (M \circ N)\ (hd\ (\boldsymbol{\lambda}[P] \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M \circ N);$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (hd\ (\boldsymbol{\lambda}[P] \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ N\ (filter\ notIde\ (map\ \Lambda.un\text{-}App2\ (\boldsymbol{\lambda}[P] \#\ U)))\rrbracket$
$\Longrightarrow\ ?P\ (M \circ N)\ (\boldsymbol{\lambda}[P] \#\ U)$

using $*$ $\Lambda.lambda.disc(18)$ **by** *presburger*

show $\bigwedge M\ N\ P1\ P2\ U.\ [\![\Lambda.Ide\ (M \circ N)$
$\Longrightarrow\ ?P\ (hd\ ((P1 \circ P2) \#\ U))\ (tl\ ((P1 \circ P2) \#\ U));$
$[\![\neg\ \Lambda.Ide\ (M \circ N);\ \Lambda.seq\ (M \circ N)\ (hd\ ((P1 \circ P2) \#\ U));$
$\Lambda.contains\text{-}head\text{-}reduction\ (M \circ N);$
$\Lambda.Ide\ ((M \circ N) \setminus \Lambda.head\text{-}redex\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ (hd\ ((P1 \circ P2) \#\ U))\ (tl((P1 \circ P2) \#\ U));$
$[\![\neg\ \Lambda.Ide\ (M \circ N);\ \Lambda.seq\ (M \circ N)\ (hd\ ((P1 \circ P2) \#\ U));$
$\Lambda.contains\text{-}head\text{-}reduction\ (M \circ N);$
$\neg\ \Lambda.Ide\ ((M \circ N) \setminus \Lambda.head\text{-}redex\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ ((M \circ N) \setminus \Lambda.head\text{-}redex\ (M \circ N))\ ((P1 \circ P2) \#\ U);$
$[\![\neg\ \Lambda.Ide\ (M \circ N);\ \Lambda.seq\ (M \circ N)\ (hd\ ((P1 \circ P2) \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M \circ N);$
$\Lambda.contains\text{-}head\text{-}reduction\ (hd\ ((P1 \circ P2) \#\ U));$
$\Lambda.Ide\ ((M \circ N) \setminus \Lambda.head\text{-}strategy\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ (\Lambda.head\text{-}strategy\ (M \circ N))\ (tl\ ((P1 \circ P2) \#\ U));$
$[\![\neg\ \Lambda.Ide\ (M \circ N);\ \Lambda.seq\ (M \circ N)\ (hd\ ((P1 \circ P2) \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M \circ N);$
$\Lambda.contains\text{-}head\text{-}reduction\ (hd\ ((P1 \circ P2) \#\ U));$
$\neg\ \Lambda.Ide\ ((M \circ N) \setminus \Lambda.head\text{-}strategy\ (M \circ N))\rrbracket$
$\Longrightarrow\ ?P\ ((M \circ N) \setminus \Lambda.head\text{-}strategy\ (M \circ N))\ (tl\ ((P1 \circ P2) \#\ U));$
$[\![\neg\ \Lambda.Ide\ (M \circ N);\ \Lambda.seq\ (M \circ N)\ (hd\ ((P1 \circ P2) \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M \circ N);$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (hd\ ((P1 \circ P2) \#\ U))\rrbracket$
$\Longrightarrow\ ?P\ M\ (filter\ notIde\ (map\ \Lambda.un\text{-}App1\ ((P1 \circ P2) \#\ U)));$
$[\![\neg\ \Lambda.Ide\ (M \circ N);\ \Lambda.seq\ (M \circ N)\ (hd\ ((P1 \circ P2) \#\ U));$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (M \circ N);$
$\neg\ \Lambda.contains\text{-}head\text{-}reduction\ (hd\ ((P1 \circ P2) \#\ U))\rrbracket$

```
                    ⟹ ?P N (filter notIde (map Λ.un-App2 ((P1 ∘ P2) # U)))⟧
            ⟹ ?P (M ∘ N) ((P1 ∘ P2) # U)
    using ∗ Λ.lambda.disc(19) by presburger
qed
```

The Standardization Theorem

Using the function *standardize*, we can now prove the Standardization Theorem. There is still a little bit more work to do, because we have to deal with various cases in which the reduction path to be standardized is empty or consists entirely of identities.

```
theorem standardization-theorem:
shows Arr T ⟹ Std (standardize T) ∧ (Ide T ⟶ standardize T = []) ∧
              (¬ Ide T ⟶ cong (standardize T) T)
proof (induct T)
  show Arr [] ⟹ Std (standardize []) ∧ (Ide [] ⟶ standardize [] = []) ∧
                (¬ Ide [] ⟶ cong (standardize []) [])
    by simp
  fix t T
  assume ind: Arr T ⟹ Std (standardize T) ∧ (Ide T ⟶ standardize T = []) ∧
                       (¬ Ide T ⟶ cong (standardize T) T)
  assume tT: Arr (t # T)
  have t: Λ.Arr t
    using tT Arr-imp-arr-hd by force
  show Std (standardize (t # T)) ∧ (Ide (t # T) ⟶ standardize (t # T) = []) ∧
       (¬ Ide (t # T) ⟶ cong (standardize (t # T)) (t # T))
  proof (cases T = [])
    show T = [] ⟹ ?thesis
      using t tT Ide-iff-standard-development-empty Std-standard-development
            cong-standard-development
      by simp
    assume 0: T ≠ []
    hence T: Arr T
      using tT
      by (metis Arr-imp-Arr-tl list.sel(3))
    show ?thesis
    proof (intro conjI)
      show Std (standardize (t # T))
      proof −
        have 1: ¬ Ide T ⟹ seq [t] (standardize T)
          using t T ind 0 ide-char Con-implies-Arr(1)
          apply (intro seqI_{Λ P})
            apply simp
           apply (metis Con-implies-Arr(1) Ide.simps(1) ide-char)
          by (metis Src-hd-eqI Trg-last-Src-hd-eqI ‹T ≠ []› append-Cons arrI_P
                    arr-append-imp-seq list.distinct(1) self-append-conv2 tT)
        show ?thesis
          using T 1 ind Std-standard-development stdz-insert-correctness by auto
      qed
      show Ide (t # T) ⟶ standardize (t # T) = []
```

```
  using Ide-consE Ide-iff-standard-development-empty Ide-implies-Arr ind
        Λ.Ide-implies-Arr Λ.ide-char
  by (metis list.sel(1,3) standardize.simps(1−2) stdz-insert.simps(1))
show ¬ Ide (t # T) ⟶ standardize (t # T) *∼* t # T
proof
  assume 1: ¬ Ide (t # T)
  show standardize (t # T) *∼* t # T
  proof (cases Λ.Ide t)
    assume t: Λ.Ide t
    have 2: ¬ Ide T
      using 1 t tT by fastforce
    have standardize (t # T) = stdz-insert t (standardize T)
      by simp
    also have ... *∼* t # T
    proof −
      have 3: Std (standardize T) ∧ standardize T *∼* T
        using T 2 ind by blast
      have stdz-insert t (standardize T) =
           stdz-insert (hd (standardize T)) (tl (standardize T))
      proof −
        have seq [t] (standardize T)
          using 0 2 tT ind
          by (metis Arr.elims(2) Con-imp-eq-Srcs Con-implies-Arr(1) Ide.simps(1−2)
              Ide-implies-Arr Trgs.simps(2) ide-char Λ.ide-char list.inject
              seq-char seq-implies-Trgs-eq-Srcs t)
        thus ?thesis
          using t 3 stdz-insert-Ide-Std by blast
      qed
      also have ... *∼* hd (standardize T) # tl (standardize T)
      proof −
        have ¬ Ide (standardize T)
          using 2 3 ide-backward-stable ide-char by blast
        moreover have tl (standardize T) ≠ [] ⟹
                      seq [hd (standardize T)] (tl (standardize T)) ∧
                      Std (tl (standardize T))
          by (metis 3 Std-consE Std-imp-Arr append.left-neutral append-Cons
              arr-append-imp-seq arr-char hd-Cons-tl list.discI tl-Nil)
        ultimately show ?thesis
        by (metis 2 Ide.simps(2) Resid.simps(1) Std-consE T cong-standard-development
              ide-char ind Λ.ide-char list.exhaust-sel stdz-insert.simps(1)
              stdz-insert-correctness)
      qed
      also have hd (standardize T) # tl (standardize T) = standardize T
        by (metis 3 Arr.simps(1) Con-implies-Arr(2) Ide.simps(1) ide-char
            list.exhaust-sel)
      also have standardize T *∼* T
        using 3 by simp
      also have T *∼* t # T
        using 0 t tT arr-append-imp-seq arr-char cong-cons-ideI(2) by simp
```

```
        finally show ?thesis by blast
      qed
      thus ?thesis by auto
      next
      assume t: ¬ Λ.Ide t
      show ?thesis
      proof (cases Ide T)
        assume T: Ide T
        have standardize (t # T) = standard-development t
          using t T Ide-implies-Arr ind by simp
        also have ... *∼* [t]
          using t T tT cong-standard-development [of t] by blast
        also have [t] *∼* [t] @ T
          using t T tT cong-append-ideI(4) [of [t] T]
          by (simp add: 0 arrI_P arr-append-imp-seq ide-char)
        finally show ?thesis by auto
        next
        assume T: ¬ Ide T
        have 1: Std (standardize T) ∧ standardize T *∼* T
          using T ‹Arr T› ind by blast
        have 2: seq [t] (standardize T)
          by (metis 0 Arr.simps(2) Arr.simps(3) Con-imp-eq-Srcs Con-implies-Arr(2)
              Ide.elims(3) Ide.simps(1) T Trgs.simps(2) ide-char ind
              seq-char seq-implies-Trgs-eq-Srcs tT)
        have stdz-insert t (standardize T) *∼* t # standardize T
          using t 1 2 stdz-insert-correctness [of t standardize T] by blast
        also have t # standardize T *∼* t # T
          using 1 2
          by (meson Arr.simps(2) Λ.prfx-reflexive cong-cons seq-char)
        finally show ?thesis by auto
      qed
    qed
  qed
 qed
qed
```

The Leftmost Reduction Theorem

In this section we prove the Leftmost Reduction Theorem, which states that leftmost reduction is a normalizing strategy.

We first show that if a standard reduction path reaches a normal form, then the path must be the one produced by following the leftmost reduction strategy. This is because, in a standard reduction path, once a leftmost redex is skipped, all subsequent reductions occur "to the right of it", hence they are all non-leftmost reductions that do not contract the skipped redex, which remains in the leftmost position.

The Leftmost Reduction Theorem then follows from the Standardization Theorem. If a term is normalizable, there is a reduction path from that term to a normal form.

By the Standardization Theorem we may as well assume that path is standard. But a standard reduction path to a normal form is the path generated by following the leftmost reduction strategy, hence leftmost reduction reaches a normal form after a finite number of steps.

lemma *sseq-reflects-leftmost-reduction*:
assumes Λ.*sseq t u* **and** Λ.*is-leftmost-reduction u*
shows Λ.*is-leftmost-reduction t*
proof −
have ∗: ⋀*u*. *u* = Λ.*leftmost-strategy* (Λ.*Src t*) \ *t* ⟹ ¬ Λ.*sseq t u* **for** *t*
proof (*induct t*)
show ⋀*u*. ¬ Λ.*sseq* ♯ *u*
using Λ.*sseq-imp-seq* **by** *blast*
show ⋀*x u*. ¬ Λ.*sseq «x» u*
using Λ.*elementary-reduction.simps*(*2*) Λ.*sseq-imp-elementary-reduction1* **by** *blast*
show ⋀*t u*. ⟦⋀*u*. *u* = Λ.*leftmost-strategy* (Λ.*Src t*) \ *t* ⟹ ¬ Λ.*sseq t u*;
u = Λ.*leftmost-strategy* (Λ.*Src* **λ**[*t*]) \ **λ**[*t*]⟧
⟹ ¬ Λ.*sseq* **λ**[*t*] *u*
by *auto*
show ⋀*t1 t2 u*. ⟦⋀*u*. *u* = Λ.*leftmost-strategy* (Λ.*Src t1*) \ *t1* ⟹ ¬ Λ.*sseq t1 u*;
⋀*u*. *u* = Λ.*leftmost-strategy* (Λ.*Src t2*) \ *t2* ⟹ ¬ Λ.*sseq t2 u*;
u = Λ.*leftmost-strategy* (Λ.*Src* (**λ**[*t1*] • *t2*)) \ (**λ**[*t1*] • *t2*)⟧
⟹ ¬ Λ.*sseq* (**λ**[*t1*] • *t2*) *u*
apply *simp*
by (*metis* Λ.*sseq-imp-elementary-reduction2* Λ.*Coinitial-iff-Con* Λ.*Ide-Src*
Λ.*Ide-Subst* Λ.*elementary-reduction-not-ide* Λ.*ide-char* Λ.*resid-Ide-Arr*)
show ⋀*t1 t2*. ⟦⋀*u*. *u* = Λ.*leftmost-strategy* (Λ.*Src t1*) \ *t1* ⟹ ¬ Λ.*sseq t1 u*;
⋀*u*. *u* = Λ.*leftmost-strategy* (Λ.*Src t2*) \ *t2* ⟹ ¬ Λ.*sseq t2 u*;
u = Λ.*leftmost-strategy* (Λ.*Src* (Λ.*App t1 t2*)) \ Λ.*App t1 t2*⟧
⟹ ¬ Λ.*sseq* (Λ.*App t1 t2*) *u* **for** *u*
apply (*cases u*)
apply *simp-all*
apply (*metis* Λ.*elementary-reduction.simps*(*2*) Λ.*sseq-imp-elementary-reduction2*)
apply (*metis* Λ.*Src.simps*(*3*) Λ.*Src-resid* Λ.*Trg.simps*(*3*) Λ.*lambda.distinct*(*15*)
Λ.*lambda.distinct*(*3*))
proof −
show ⋀*t1 t2 u1 u2*.
⟦¬ Λ.*sseq t1* (Λ.*leftmost-strategy* (Λ.*Src t1*) \ *t1*);
¬ Λ.*sseq t2* (Λ.*leftmost-strategy* (Λ.*Src t2*) \ *t2*);
λ[*u1*] • *u2* = Λ.*leftmost-strategy* (Λ.*App* (Λ.*Src t1*) (Λ.*Src t2*)) \ Λ.*App t1 t2*;
u = Λ.*leftmost-strategy* (Λ.*App* (Λ.*Src t1*) (Λ.*Src t2*)) \ Λ.*App t1 t2*⟧
⟹ ¬ Λ.*sseq* (Λ.*App t1 t2*)
(Λ.*leftmost-strategy* (Λ.*App* (Λ.*Src t1*) (Λ.*Src t2*)) \ Λ.*App t1 t2*)
by (*metis* Λ.*sseq-imp-elementary-reduction1* Λ.*Arr.simps*(*5*) Λ.*Arr-resid*
Λ.*Coinitial-iff-Con* Λ.*Ide.simps*(*5*) Λ.*Ide-iff-Src-self* Λ.*Src.simps*(*4*)
Λ.*Src-resid* Λ.*contains-head-reduction.simps*(*8*) Λ.*is-head-reduction-if*
Λ.*lambda.discI*(*3*) Λ.*lambda.distinct*(*7*)
Λ.*leftmost-strategy-selects-head-reduction* Λ.*resid-Arr-self*
Λ.*sseq-preserves-App-and-no-head-reduction*)
show ⋀*u1 u2*.

```
          ⟦¬ Λ.sseq t1 (Λ.leftmost-strategy (Λ.Src t1) \ t1);
           ¬ Λ.sseq t2 (Λ.leftmost-strategy (Λ.Src t2) \ t2);
           Λ.App u1 u2 = Λ.leftmost-strategy (Λ.App (Λ.Src t1) (Λ.Src t2)) \ Λ.App t1 t2;
           u = Λ.leftmost-strategy (Λ.App (Λ.Src t1) (Λ.Src t2)) \ Λ.App t1 t2⟧
            ⟹ ¬ Λ.sseq (Λ.App t1 t2)
                       (Λ.leftmost-strategy (Λ.App (Λ.Src t1) (Λ.Src t2)) \ Λ.App t1 t2)
    for t1 t2
     apply (cases ¬ Λ.Arr t1)
      apply simp-all
      apply (meson Λ.Arr.simps(4) Λ.seq-char Λ.sseq-imp-seq)
     apply (cases ¬ Λ.Arr t2)
      apply simp-all
      apply (meson Λ.Arr.simps(4) Λ.seq-char Λ.sseq-imp-seq)
     using Λ.Arr-not-Nil
     apply (cases t1)
        apply simp-all
     using Λ.NF-iff-has-no-redex Λ.has-redex-iff-not-Ide-leftmost-strategy
           Λ.Ide-iff-Src-self Λ.Ide-iff-Trg-self
           Λ.NF-def Λ.elementary-reduction-not-ide Λ.eq-Ide-are-cong
           Λ.leftmost-strategy-is-reduction-strategy Λ.reduction-strategy-def
           Λ.resid-Arr-Src
      apply simp
      apply (metis Λ.Arr.simps(4) Λ.Ide.simps(4) Λ.Ide-Trg Λ.Src.simps(4)
                   Λ.sseq-imp-elementary-reduction2)
    by (metis Λ.Ide-Trg Λ.elementary-reduction-not-ide Λ.ide-char)
  qed
qed
have t ≠ Λ.leftmost-strategy (Λ.Src t) ⟹ False
proof −
  assume 1: t ≠ Λ.leftmost-strategy (Λ.Src t)
  have 2: ¬ Λ.Ide (Λ.leftmost-strategy (Λ.Src t))
    by (meson assms(1) Λ.NF-def Λ.NF-iff-has-no-redex Λ.arr-char
        Λ.elementary-reduction-is-arr Λ.elementary-reduction-not-ide
        Λ.has-redex-iff-not-Ide-leftmost-strategy Λ.ide-char
        Λ.sseq-imp-elementary-reduction1)
  have Λ.is-leftmost-reduction (Λ.leftmost-strategy (Λ.Src t) \ t)
  proof −
    have Λ.is-leftmost-reduction (Λ.leftmost-strategy (Λ.Src t))
      by (metis assms(1) 2 Λ.Ide-Src Λ.Ide-iff-Src-self Λ.arr-char
          Λ.elementary-reduction-is-arr Λ.elementary-reduction-leftmost-strategy
          Λ.is-leftmost-reduction-def Λ.leftmost-strategy-is-reduction-strategy
          Λ.reduction-strategy-def Λ.sseq-imp-elementary-reduction1)
    moreover have 3: Λ.elementary-reduction t
      using assms Λ.sseq-imp-elementary-reduction1 by simp
    moreover have ¬ Λ.is-leftmost-reduction t
      using 1 Λ.is-leftmost-reduction-def by auto
    moreover have Λ.coinitial (Λ.leftmost-strategy (Λ.Src t)) t
      using 3 Λ.leftmost-strategy-is-reduction-strategy Λ.reduction-strategy-def
            Λ.Ide-Src Λ.elementary-reduction-is-arr
```

```
      by force
    ultimately show ?thesis
      using 1 Λ.leftmost-reduction-preservation by blast
  qed
  moreover have Λ.coinitial (Λ.leftmost-strategy (Λ.Src t) \ t) u
    using assms(1) calculation Λ.Arr-not-Nil Λ.Src-resid Λ.elementary-reduction-is-arr
          Λ.is-leftmost-reduction-def Λ.seq-char Λ.sseq-imp-seq
    by force
  moreover have ⋀v. ⟦Λ.is-leftmost-reduction v; Λ.coinitial v u⟧ ⟹ v = u
    by (metis Λ.arr-iff-has-source Λ.arr-resid-iff-con Λ.confluence assms(2)
        Λ.Arr-not-Nil Λ.Coinitial-iff-Con Λ.is-leftmost-reduction-def Λ.sources-char_Λ)
  ultimately have Λ.leftmost-strategy (Λ.Src t) \ t = u
    by blast
  thus ?thesis
    using assms(1) * by blast
qed
thus ?thesis
  using assms(1) Λ.is-leftmost-reduction-def Λ.sseq-imp-elementary-reduction1 by force
qed

lemma elementary-reduction-to-NF-is-leftmost:
shows ⟦Λ.elementary-reduction t; Λ.NF (Trg [t])⟧ ⟹ Λ.leftmost-strategy (Λ.Src t) = t
proof (induct t)
  show Λ.leftmost-strategy (Λ.Src ♯) = ♯
    by simp
  show ⋀x. ⟦Λ.elementary-reduction «x»; Λ.NF (Trg [«x»])⟧
             ⟹ Λ.leftmost-strategy (Λ.Src «x») = «x»
    by auto
  show ⋀t. ⟦⟦Λ.elementary-reduction t; Λ.NF (Trg [t])⟧
             ⟹ Λ.leftmost-strategy (Λ.Src t) = t;
            Λ.elementary-reduction λ[t]; Λ.NF (Trg [λ[t]])⟧
             ⟹ Λ.leftmost-strategy (Λ.Src λ[t]) = λ[t]
    using lambda-calculus.NF-Lam-iff lambda-calculus.elementary-reduction-is-arr by force
  show ⋀t1 t2. ⟦⟦Λ.elementary-reduction t1; Λ.NF (Trg [t1])⟧
                  ⟹ Λ.leftmost-strategy (Λ.Src t1) = t1;
                ⟦Λ.elementary-reduction t2; Λ.NF (Trg [t2])⟧
                  ⟹ Λ.leftmost-strategy (Λ.Src t2) = t2;
                Λ.elementary-reduction (λ[t1] • t2); Λ.NF (Trg [λ[t1] • t2])⟧
                  ⟹ Λ.leftmost-strategy (Λ.Src (λ[t1] • t2)) = λ[t1] • t2
    apply simp
    by (metis Λ.Ide-iff-Src-self Λ.Ide-implies-Arr)
  fix t1 t2
  assume ind1: ⟦Λ.elementary-reduction t1; Λ.NF (Trg [t1])⟧
                  ⟹ Λ.leftmost-strategy (Λ.Src t1) = t1
  assume ind2: ⟦Λ.elementary-reduction t2; Λ.NF (Trg [t2])⟧
                  ⟹ Λ.leftmost-strategy (Λ.Src t2) = t2
  assume t: Λ.elementary-reduction (Λ.App t1 t2)
  have t1: Λ.Arr t1
    using t Λ.Arr.simps(4) Λ.elementary-reduction-is-arr by blast
```

```
  have t2: Λ.Arr t2
    using t Λ.Arr.simps(4) Λ.elementary-reduction-is-arr by blast
  assume NF: Λ.NF (Trg [Λ.App t1 t2])
  have 1: ¬ Λ.is-Lam t1
    using NF Λ.NF-def
    apply (cases t1)
       apply simp-all
  by (metis (mono-tags) Λ.Ide.simps(1) Λ.NF-App-iff Λ.Trg.simps(2−3) Λ.lambda.discI(2))
  have 2: Λ.NF (Λ.Trg t1) ∧ Λ.NF (Λ.Trg t2)
    using NF t1 t2 1 Λ.NF-App-iff by simp
  show Λ.leftmost-strategy (Λ.Src (Λ.App t1 t2)) = Λ.App t1 t2
    using t t1 t2 1 2 ind1 ind2
    apply (cases t1)
       apply simp-all
     apply (metis Λ.Ide.simps(4) Λ.Ide-iff-Src-self Λ.Ide-iff-Trg-self
       Λ.NF-iff-has-no-redex Λ.elementary-reduction-not-ide Λ.eq-Ide-are-cong
       Λ.has-redex-iff-not-Ide-leftmost-strategy Λ.resid-Arr-Src t1)
    using Λ.Ide-iff-Src-self by blast
qed

lemma Std-path-to-NF-is-leftmost:
shows ⟦Std T; Λ.NF (Trg T)⟧ ⟹ set T ⊆ Collect Λ.is-leftmost-reduction
proof −
  have 1: ⋀t. ⟦Std (t # T); Λ.NF (Trg (t # T))⟧ ⟹ Λ.is-leftmost-reduction t for T
  proof (induct T)
    show ⋀t. ⟦Std [t]; Λ.NF (Trg [t])⟧ ⟹ Λ.is-leftmost-reduction t
      using elementary-reduction-to-NF-is-leftmost Λ.is-leftmost-reduction-def by simp
    fix t u T
    assume ind: ⋀t. ⟦Std (t # T); Λ.NF (Trg (t # T))⟧ ⟹ Λ.is-leftmost-reduction t
    assume Std: Std (t # u # T)
    assume Λ.NF (Trg (t # u # T))
    show Λ.is-leftmost-reduction t
      using Std ‹Λ.NF (Trg (t # u # T))› ind sseq-reflects-leftmost-reduction by auto
  qed
  show ⟦Std T; Λ.NF (Trg T)⟧ ⟹ set T ⊆ Collect Λ.is-leftmost-reduction
  proof (induct T)
    show 2: set [] ⊆ Collect Λ.is-leftmost-reduction
      by simp
    fix t T
    assume ind: ⟦Std T; Λ.NF (Trg T)⟧ ⟹ set T ⊆ Collect Λ.is-leftmost-reduction
    assume Std: Std (t # T) and NF: Λ.NF (Trg (t # T))
    show set (t # T) ⊆ Collect Λ.is-leftmost-reduction
      by (metis 1 2 NF Std Std-consE Trg.elims ind insert-subset list.inject list.simps(15)
          mem-Collect-eq)
  qed
qed

theorem leftmost-reduction-theorem:
shows Λ.normalizing-strategy Λ.leftmost-strategy
```

```
proof (unfold Λ.normalizing-strategy-def, intro allI impI)
  fix a
  assume a: Λ.normalizable a
  show ∃n. Λ.NF (Λ.reduce Λ.leftmost-strategy a n)
  proof (cases Λ.NF a)
    show Λ.NF a ⟹ ?thesis
      by (metis lambda-calculus.reduce.simps(1))
    assume 1: ¬ Λ.NF a
    obtain T where T: Arr T ∧ Src T = a ∧ Λ.NF (Trg T)
      using a Λ.normalizable-def red-iff by auto
    have 2: ¬ Ide T
      using T 1 Ide-imp-Src-eq-Trg by fastforce
    obtain U where U: Std U ∧ cong T U
      using T 2 standardization-theorem by blast
    have 3: set U ⊆ Collect Λ.is-leftmost-reduction
      using 1 U Std-path-to-NF-is-leftmost
      by (metis Con-Arr-self Resid-parallel Src-resid T cong-implies-coinitial)
    have ⋀U. ⟦Arr U; length U = n; set U ⊆ Collect Λ.is-leftmost-reduction⟧ ⟹
            U = apply-strategy Λ.leftmost-strategy (Src U) (length U) for n
    proof (induct n)
      show ⋀U. ⟦Arr U; length U = 0; set U ⊆ Collect Λ.is-leftmost-reduction⟧
                ⟹ U = apply-strategy Λ.leftmost-strategy (Src U) (length U)
        by simp
      fix n U
      assume ind: ⋀U. ⟦Arr U; length U = n; set U ⊆ Collect Λ.is-leftmost-reduction⟧
                    ⟹ U = apply-strategy Λ.leftmost-strategy (Src U) (length U)
      assume U: Arr U
      assume n: length U = Suc n
      assume set: set U ⊆ Collect Λ.is-leftmost-reduction
      show U = apply-strategy Λ.leftmost-strategy (Src U) (length U)
      proof (cases n = 0)
        show n = 0 ⟹ ?thesis
          using U n 1 set Λ.is-leftmost-reduction-def
          by (cases U) auto
        assume 5: n ≠ 0
        have 4: hd U = Λ.leftmost-strategy (Src U)
          using n U set Λ.is-leftmost-reduction-def
          by (cases U) auto
        have 6: tl U ≠ []
          using 4 5 n U
          by (metis Suc-length-conv list.sel(3) list.size(3))
        show ?thesis
          using 4 5 6 n U set ind [of tl U]
          apply (cases n)
           apply simp-all
          by (metis (no-types, lifting) Arr-consE Nil-tl Nitpick.size-list-simp(2)
              ind [of tl U] Λ.arr-char Λ.trg-char list.collapse list.set-sel(2)
              old.nat.inject reduction-paths.apply-strategy.simps(2) subset-code(1))
      qed
```

```
      qed
      hence U = apply-strategy Λ.leftmost-strategy (Src U) (length U)
        by (metis 3 Con-implies-Arr(1) Ide.simps(1) U ide-char)
      moreover have Src U = a
        using T U cong-implies-coinitial
        by (metis Con-imp-eq-Srcs Con-implies-Arr(2) Ide.simps(1) Srcs-simp_PWE empty-set
            ex-un-Src ide-char list.set-intros(1) list.simps(15))
      ultimately have Trg U = Λ.reduce Λ.leftmost-strategy a (length U)
        using reduce-eq-Trg-apply-strategy
        by (metis Arr.simps(1) Con-implies-Arr(1) Ide.simps(1) U a ide-char
            Λ.leftmost-strategy-is-reduction-strategy Λ.normalizable-def length-greater-0-conv)
      thus ?thesis
        by (metis Ide.simps(1) Ide-imp-Src-eq-Trg Src-resid T Trg-resid-sym U ide-char)
    qed
  qed

end

end
```

Bibliography

[1] H. Barendregt. *The Lambda-calculus: Its Syntax and Semantics*. North-Holland, 1984.

[2] M. Copes. A machine-checked proof of the standardization theorem in lambda calculus using multiple substitution. Master's thesis, Universidad ORT Uruguay, 2018. https://dspace.ort.edu.uy/bitstream/handle/20.500.11968/3725/Material%20completo.pdf.

[3] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

[4] N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 34(5):381–392, 1972.

[5] R. de Vrijer. A direct proof of the finite developments theorem. *The Journal of Symbolic Logic*, 50(2):339–343, June 1985.

[6] R. Hindley. Reductions of residuals are finite. *Transactions of the American Mathematical Society*, 240:345–361, June 1978.

[7] G. Huet. Residual theory in 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.