Process Composition

Filip Smola

March 10, 2025

Contents

theory *Util*
  imports *Main*
begin

1 Utility Theorems

This theory contains general facts that we use in our proof but which do not depend on our development.

list-all and *list-ex* are dual

lemma *not-list-all*:
  $(\neg\ \mathit{list\text{-}all}\ P\ xs) = \mathit{list\text{-}ex}\ (\lambda x.\ \neg\ P\ x)\ xs$
  ⟨*proof*⟩
lemma *not-list-ex*:
  $(\neg\ \mathit{list\text{-}ex}\ P\ xs) = \mathit{list\text{-}all}\ (\lambda x.\ \neg\ P\ x)\ xs$
  ⟨*proof*⟩

A list of length more than one starts with two elements

lemma *list-obtain-2*:
assumes $1 < length\ xs$
obtains $v\ vb\ vc$ **where** $xs = v \# vb \# vc$
⟨*proof*⟩

Generalise the theorem $[\![?k < ?l; ?m + ?l = ?k + ?n]\!] \Longrightarrow ?m < ?n$

lemma *less-add-eq-less-general*:
fixes $k\ l\ m\ n :: 'a :: \{comm\text{-}monoid\text{-}add,\ ordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add,\ linorder\}$
assumes $k < l$
and $m + l = k + n$
shows $m < n$
⟨*proof*⟩

Consider a list of elements and two functions, one of which is always at less-than or equal to the other on elements of that list. If for one element of that list the first function is strictly less than the other, then summing the list with the first function is also strictly less summing it with the second function.

lemma *sum-list-mono-one-strict*:
fixes $f\ g :: 'a \Rightarrow 'b :: \{comm\text{-}monoid\text{-}add,\ ordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add,\ linorder\}$
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x \leq g\ x$
and $x \in set\ xs$
and $f\ x < g\ x$
shows $sum\text{-}list\ (map\ f\ xs) < sum\text{-}list\ (map\ g\ xs)$
⟨*proof*⟩

Generalise $(\bigwedge x.\ x \in set\ ?xs \Longrightarrow ?f\ x \leq ?g\ x) \Longrightarrow sum\text{-}list\ (map\ ?f\ ?xs) \leq sum\text{-}list\ (map\ ?g\ ?xs)$ to allow for different lists

lemma *sum-list-mono-list-all2*:
fixes $f\ g :: 'a \Rightarrow 'b::\{monoid\text{-}add,\ ordered\text{-}ab\text{-}semigroup\text{-}add\}$
assumes $list\text{-}all2\ (\lambda x\ y.\ f\ x \leq g\ y)\ xs\ ys$
shows $(\sum x{\leftarrow}xs.\ f\ x) \leq (\sum x{\leftarrow}ys.\ g\ x)$
⟨*proof*⟩

Generalise $[\![\bigwedge x.\ x \in set\ ?xs \Longrightarrow ?f\ x \leq ?g\ x; ?x \in set\ ?xs; ?f\ ?x < ?g\ ?x]\!] \Longrightarrow sum\text{-}list\ (map\ ?f\ ?xs) < sum\text{-}list\ (map\ ?g\ ?xs)$ to allow for different lists

lemma *sum-list-mono-one-strict-list-all2*:
fixes $f\ g :: 'a \Rightarrow 'b :: \{comm\text{-}monoid\text{-}add,\ ordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add,\ linorder\}$
assumes $list\text{-}all2\ (\lambda x\ y.\ f\ x \leq g\ y)\ xs\ ys$
and $(x, y) \in set\ (zip\ xs\ ys)$
and $f\ x < g\ y$
shows $sum\text{-}list\ (map\ f\ xs) < sum\text{-}list\ (map\ g\ ys)$
⟨*proof*⟩

Define a function to count the number of list elements satisfying a predicate

primrec *count-if* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow nat$
  where
    $count\text{-}if\ P\ []\ =\ 0$
  $|\ count\text{-}if\ P\ (x\#xs) = (if\ P\ x\ then\ Suc\ (count\text{-}if\ P\ xs)\ else\ count\text{-}if\ P\ xs)$

lemma *count-if-append* [*simp*]:
  $count\text{-}if\ P\ (xs\ @\ ys) = count\text{-}if\ P\ xs + count\text{-}if\ P\ ys$
  $\langle proof \rangle$

lemma *count-if-0-conv*:
  $(count\text{-}if\ P\ xs = 0) = (\neg\ list\text{-}ex\ P\ xs)$
  $\langle proof \rangle$

Intersection of sets that are the same is any of those sets

lemma *Inter-all-same*:
  assumes $\bigwedge x\ y.\ [\![x \in A;\ y \in A]\!] \Longrightarrow f\ x = f\ y$
    and $x \in A$
   shows $(\bigcap x \in A.\ f\ x) = f\ x$
  $\langle proof \rangle$

end
theory *ResTerm*
  imports *Main*
begin

2 Resource Terms

Resource terms describe resources with atoms drawn from two types, linear and copyable, combined in a number of ways:

- Parallel resources represent their simultaneous presence,
- Non-deterministic resource represent exactly one of two options,
- Executable resources represent a single potential execution of a process transforming one resource into another,
- Repeatably executable resources represent an unlimited amount of such potential executions.

  We define two distinguished resources on top of the atoms:

- Empty, to represent the absence of a resource and serve as the unit for parallel combination,
- Anything, to represent a resource about which we have no information.

datatype (*discs-sels*) ($'a$, $'b$) *res-term* =
  Res $'a$
    — Linear resource atom
  | *Copyable* $'b$
    — Copyable resource atom
  | *is-Empty*: *Empty*
    — The absence of a resource
  | *is-Anything*: *Anything*
    — Resource about which we know nothing
  | *Parallel* ($'a$, $'b$) *res-term list*
    — Parallel combination
  | *NonD* ($'a$, $'b$) *res-term* ($'a$, $'b$) *res-term*
    — Non-deterministic combination
  | *Executable* ($'a$, $'b$) *res-term* ($'a$, $'b$) *res-term*
    — Executable resource
  | *Repeatable* ($'a$, $'b$) *res-term* ($'a$, $'b$) *res-term*
    — Repeatably executable resource

Every child of *Parallel* is smaller than it

lemma *parallel-child-smaller*:
  $x \in set\ xs \Longrightarrow size\text{-}res\text{-}term\ f\ g\ x < size\text{-}res\text{-}term\ f\ g\ (Parallel\ xs)$
$\langle proof \rangle$

No singleton *Parallel* is equal to its own child, because the child has to be smaller

lemma *parallel-neq-single* [*simp*]:
  $Parallel\ [a] \neq a$
$\langle proof \rangle$

2.1 Resource Term Equivalence

Some resource terms are different descriptions of the same situation. We express this by relating resource terms as follows:

- *Parallel* [] with *Empty*
- *Parallel* [x] with x
- *Parallel* (xs @ [*Parallel* ys] @ zs) with *Parallel* (xs @ ys @ zs)

  We extend this with the reflexive base cases, recursive cases and symmetric-transitive closure. As a result, we get an equivalence relation on resource terms, which we will later use to quotient the terms and form a type of resources.

inductive *res-term-equiv* :: ($'a$, $'b$) *res-term* $\Rightarrow$ ($'a$, $'b$) *res-term* $\Rightarrow$ *bool* (**infix** $\sim$ *100*)
  where

nil: $Parallel\ [] \sim Empty$
| $singleton$: $Parallel\ [a] \sim a$
| $merge$: $Parallel\ (x\ @\ [Parallel\ y]\ @\ z) \sim Parallel\ (x\ @\ y\ @\ z)$
| $empty$: $Empty \sim Empty$
| $anything$: $Anything \sim Anything$
| res: $Res\ x \sim Res\ x$
| $copyable$: $Copyable\ x \sim Copyable\ x$
| $parallel$: $list\text{-}all2\ (\sim)\ xs\ ys \Longrightarrow Parallel\ xs \sim Parallel\ ys$
| $nondet$: $[\![x \sim y;\ u \sim v]\!] \Longrightarrow NonD\ x\ u \sim NonD\ y\ v$
| $executable$: $[\![x \sim y;\ u \sim v]\!] \Longrightarrow Executable\ x\ u \sim Executable\ y\ v$
| $repeatable$: $[\![x \sim y;\ u \sim v]\!] \Longrightarrow Repeatable\ x\ u \sim Repeatable\ y\ v$
| sym $[sym]$: $x \sim y \Longrightarrow y \sim x$
| $trans$ $[trans]$: $[\![x \sim y;\ y \sim z]\!] \Longrightarrow x \sim z$

Add some of the rules for the simplifier

lemmas [*simp*] =
nil nil[*symmetric*]
singleton singleton[*symmetric*]

Constrain all these rules to the resource term equivalence namespace

hide-fact (**open**) *empty anything res copyable nil singleton merge parallel nondet executable*
repeatable sym trans

Next we derive a handful of rules for the equivalence, placing them in its namespace

⟨*ML*⟩

It can be shown to be reflexive

lemma *refl* [*simp*]:
$a \sim a$
⟨*proof*⟩

lemma *reflI*:
$a = b \Longrightarrow a \sim b$
⟨*proof*⟩

lemma *equivp* [*simp*]:
equivp res-term-equiv
⟨*proof*⟩

Parallel resource terms can be related by splitting them into parts

lemma *decompose*:
assumes $Parallel\ x1 \sim Parallel\ y1$
and $Parallel\ x2 \sim Parallel\ y2$
shows $Parallel\ (x1\ @\ x2) \sim Parallel\ (y1\ @\ y2)$
⟨*proof*⟩

We can drop a unit from any parallel resource term

lemma *drop*:
  Parallel $(x @ [Empty] @ y) \sim$ *Parallel* $(x @ y)$
⟨*proof*⟩

Equivalent resource terms remain equivalent wrapped in a parallel

lemma *singleton-both*:
  $x \sim y \Longrightarrow$ *Parallel* $[x] \sim$ *Parallel* $[y]$
  ⟨*proof*⟩

We can reduce a resource term equivalence given equivalences for both sides

lemma *trans-both*:
  $[\![a \sim x; y \sim b; x \sim y]\!] \Longrightarrow a \sim b$
  ⟨*proof*⟩

⟨*ML*⟩

experiment begin
lemma *Parallel* [*Parallel* [], *Empty*] $\sim$ *Empty*
⟨*proof*⟩
end

Inserting equivalent terms anywhere in equivalent parallel terms preserves the equivalence

lemma *res-term-parallel-insert*:
  assumes *Parallel* $x \sim$ *Parallel* y
    and *Parallel* $u \sim$ *Parallel* v
    and $a \sim b$
  shows *Parallel* $(x @ [a] @ u) \sim$ *Parallel* $(y @ [b] @ v)$
  ⟨*proof*⟩

With inserting at the start being just a special case

lemma *res-term-parallel-cons*:
  assumes *Parallel* $x \sim$ *Parallel* y
    and $a \sim b$
  shows *Parallel* $(a \# x) \sim$ *Parallel* $(b \# y)$
  ⟨*proof*⟩

Empty is a unit for binary *Parallel*

lemma *res-term-parallel-emptyR* [*simp*]: *Parallel* [x, *Empty*] $\sim x$
  ⟨*proof*⟩
lemma *res-term-parallel-emptyL* [*simp*]: *Parallel* [*Empty*, x] $\sim x$
  ⟨*proof*⟩

Term equivalence is preserved by parallel on either side

lemma *res-term-equiv-parallel* [*simp*]:
  $x \sim y \Longrightarrow x \sim$ *Parallel* $[y]$
  ⟨*proof*⟩
lemmas [*simp*] = *res-term-equiv-parallel*[*symmetric*]

Resource term map preserves equivalence:

lemma *map-res-term-preserves-equiv* [*simp*]:
  $x \sim y \Longrightarrow$ *map-res-term f g x* $\sim$ *map-res-term f g y*
⟨*proof*⟩

The other direction is not true in general, because they may be new equivalences created by mapping different atoms to the same one. However, the counter-example proof requires a decision procedure for the equivalence to prove that two distinct atoms are not equivalent terms. As such, we delay it until normalisation for the terms is established.

2.2 Parallel Parts

Parallel resources often arise in processes, because they describe the frequent situation of having multiple resources be simultaneously present. With resource terms, the way this situation is expressed can get complex. To simplify it, we define a function to extract the list of parallel resource terms, traversing nested *Parallel* terms and dropping any *Empty* resources in them. We call these the parallel parts.

primrec *parallel-parts* :: ('*a*, '*b*) *res-term* ⇒ ('*a*, '*b*) *res-term list*
  where
    parallel-parts Empty = []
  | *parallel-parts Anything* = [*Anything*]
  | *parallel-parts* (*Res a*) = [*Res a*]
  | *parallel-parts* (*Copyable a*) = [*Copyable a*]
  | *parallel-parts* (*Parallel xs*) = *concat* (*map parallel-parts xs*)
  | *parallel-parts* (*NonD a b*) = [*NonD a b*]
  | *parallel-parts* (*Executable a b*) = [*Executable a b*]
  | *parallel-parts* (*Repeatable a b*) = [*Repeatable a b*]

Every resource is equivalent to combining its parallel parts in parallel

lemma *parallel-parts-eq*:
  $x \sim$ *Parallel* (*parallel-parts x*)
⟨*proof*⟩

Equivalent parallel parts is the same as equivalent resource terms

lemma *equiv-parallel-parts*:
  list-all2 ($\sim$) (*parallel-parts a*) (*parallel-parts b*) = $a \sim b$
⟨*proof*⟩

Note that resource term equivalence does not imply parallel parts equality

lemma
  obtains *x y* **where** $x \sim y$ **and** *parallel-parts x* $\neq$ *parallel-parts y*
⟨*proof*⟩

But it does imply that both have equal number of parallel parts

lemma *parallel-parts-length-eq*:
  $x \sim y \Longrightarrow \mathit{length}\ (\mathit{parallel\text{-}parts}\ x) = \mathit{length}\ (\mathit{parallel\text{-}parts}\ y)$
  ⟨*proof*⟩

Empty parallel parts, however, is the same as equivalence to the unit

lemma *parallel-parts-nil-equiv-empty*:
  $(\mathit{parallel\text{-}parts}\ a = []) = a \sim \mathit{Empty}$
  ⟨*proof*⟩

Singleton parallel parts imply equivalence to the one element

lemma *parallel-parts-single-equiv-element*:
  $\mathit{parallel\text{-}parts}\ a = [x] \Longrightarrow a \sim x$
  ⟨*proof*⟩

No element of parallel parts is *Parallel* or *Empty*

lemma *parallel-parts-have-no-empty*:
  $x \in \mathit{set}\ (\mathit{parallel\text{-}parts}\ a) \Longrightarrow \neg\ \mathit{is\text{-}Empty}\ x$
  ⟨*proof*⟩
lemma *parallel-parts-have-no-par*:
  $x \in \mathit{set}\ (\mathit{parallel\text{-}parts}\ a) \Longrightarrow \neg\ \mathit{is\text{-}Parallel}\ x$
  ⟨*proof*⟩

Every parallel part of a resource is at most as big as it

lemma *parallel-parts-not-bigger*:
  $x \in \mathit{set}\ (\mathit{parallel\text{-}parts}\ a) \Longrightarrow \mathit{size\text{-}res\text{-}term}\ f\ g\ x \leq (\mathit{size\text{-}res\text{-}term}\ f\ g\ a)$
⟨*proof*⟩

Any resource that is not *Empty* or *Parallel* has itself as parallel part

lemma *parallel-parts-self* [*simp*]:
  $[\![\neg\ \mathit{is\text{-}Empty}\ x;\ \neg\ \mathit{is\text{-}Parallel}\ x]\!] \Longrightarrow \mathit{parallel\text{-}parts}\ x = [x]$
  ⟨*proof*⟩

List of terms with no *Empty* or *Parallel* elements is the same as parallel parts of the *Parallel* term build from it

lemma *parallel-parts-no-empty-parallel*:
  assumes $\neg\ \mathit{list\text{-}ex}\ \mathit{is\text{-}Empty}\ \mathit{xs}$
    and $\neg\ \mathit{list\text{-}ex}\ \mathit{is\text{-}Parallel}\ \mathit{xs}$
  shows $\mathit{parallel\text{-}parts}\ (\mathit{Parallel}\ \mathit{xs}) = \mathit{xs}$
  ⟨*proof*⟩

2.3 Parallelisation

In the opposite direction of parallel parts, we can take a list of resource terms and combine them in parallel in a way smarter than just using *Parallel*. This rests in checking the list length, using the *Empty* resource if it is empty and skipping the wrapping in *Parallel* if it has only a single element. We call this parallelisation.

fun *parallelise* :: $('a, 'b)$ *res-term list* $\Rightarrow$ $('a, 'b)$ *res-term*
where
parallelise [] = *Empty*
| *parallelise* [x] = x
| *parallelise* *xs* = *Parallel xs*

This produces equivalent results to the *Parallel* constructor

lemma *parallelise-equiv*:
parallelise xs $\sim$ *Parallel xs*
⟨*proof*⟩

Lists of equal length that parallelise to the same term must have been equal

lemma *parallelise-same-length*:
$[\![$*parallelise* x = *parallelise* y; *length* x = *length* $y]\!] \Longrightarrow x = y$
⟨*proof*⟩

Parallelisation and naive parallel combination have the same parallel parts

lemma *parallel-parts-parallelise-eq*:
parallel-parts (*parallelise xs*) = *parallel-parts* (*Parallel xs*)
⟨*proof*⟩

Parallelising to a *Parallel* term means the input is either:

- A singleton set containing just that resulting *Parallel* term, or
- Exactly the children of the output and with at least two elements.

lemma *parallelise-to-parallel-conv*:
(*parallelise xs* = *Parallel ys*) = (*xs* = [*Parallel ys*] $\vee$ ($1 <$ *length xs* $\wedge$ *xs* = *ys*))
⟨*proof*⟩

So parallelising to a *Parallel* term with the same children is the same as the list having at least two elements

lemma *parallelise-to-parallel-same-length*:
(*parallelise xs* = *Parallel xs*) = ($1 <$ *length xs*)
⟨*proof*⟩

If the output of parallelisation contains a nested *Parallel* term then so must have the input list

lemma *parallelise-to-parallel-has-paralell*:
assumes *parallelise xs* = *Parallel ys*
and *list-ex is-Parallel ys*
shows *list-ex is-Parallel xs*
⟨*proof*⟩

If the output of parallelisation contains *Empty* then so must have the input

lemma *parallelise-to-parallel-has-empty*:

assumes *parallelise xs* = *Parallel ys*
obtains *xs* = [*Parallel ys*]
| *xs* = *ys*
⟨*proof*⟩

Parallelising to *Empty* means the input list was either empty or contained just that

lemma *parallelise-to-empty-eq*:
assumes *parallelise xs* = *Empty*
obtains *xs* = []
| *xs* = [*Empty*]
⟨*proof*⟩

If a list parallelises to anything but *Parallel* or *Empty*, then it must have been a singleton of that term

lemma *parallelise-to-single-eq*:
assumes *parallelise xs* = *a*
and ¬ *is-Empty a*
and ¬ *is-Parallel a*
shows *xs* = [*a*]
⟨*proof*⟩

Sets of atoms after parallelisation are unions of those atoms sets for the inputs

lemma *set1-res-term-parallelise* [*simp*]:
set1-res-term (*ResTerm.parallelise xs*) = ⋃(*set1-res-term* ‘ *set xs*)
⟨*proof*⟩
lemma *set2-res-term-parallelise* [*simp*]:
set2-res-term (*ResTerm.parallelise xs*) = ⋃(*set2-res-term* ‘ *set xs*)
⟨*proof*⟩

2.4 Refinement

Resource term refinement applies two functions to the linear and copyable atoms in a term. Unlike *map-res-term*, the first function (applied to linear atoms) is allowed to produce full resource terms, not just other atoms. (The second function must still produce other atoms, because we cannot replace a copyable atom with an arbitrary, possibly not copyable, resource.) This allows us to refine atoms into potentially complex terms.

primrec *refine-res-term* ::
('*a* ⇒ ('*x*, '*y*) *res-term*) ⇒ ('*b* ⇒ '*y*) ⇒ ('*a*, '*b*) *res-term* ⇒ ('*x*, '*y*) *res-term*
where
refine-res-term f g Empty = *Empty*
| *refine-res-term f g Anything* = *Anything*
| *refine-res-term f g* (*Res a*) = *f a*
| *refine-res-term f g* (*Copyable x*) = *Copyable* (*g x*)
| *refine-res-term f g* (*Parallel xs*) = *Parallel* (*map* (*refine-res-term f g*) *xs*)

$|$ *refine-res-term f g* (*NonD x y*) = *NonD* (*refine-res-term f g x*) (*refine-res-term f g y*)
$|$ *refine-res-term f g* (*Executable x y*) =
Executable (*refine-res-term f g x*) (*refine-res-term f g y*)
$|$ *refine-res-term f g* (*Repeatable x y*) =
Repeatable (*refine-res-term f g x*) (*refine-res-term f g y*)

Two refined resources are equivalent if:

- the original resources were equivalent,
- the linear atom refinements produce equivalent terms and
- the copyable atom refinements produce identical atoms.

lemma *refine-res-term-eq*:
assumes $x \sim y$
and $\bigwedge x.\ f\ x \sim f'\ x$
and $\bigwedge x.\ g\ x = g'\ x$
shows *refine-res-term f g x* $\sim$ *refine-res-term f' g' y*
⟨*proof*⟩

2.5 Removing *Empty* Terms From a List

As part of simplifying resource terms, it is sometimes useful to be able to take a list of terms and drop from it any empty resource.

primrec *remove-all-empty* :: ('*a*, '*b*) *res-term list* ⇒ ('*a*, '*b*) *res-term list*
where
remove-all-empty [] = []
$|$ *remove-all-empty* (*x#xs*) = (*if is-Empty x then remove-all-empty xs else x#remove-all-empty xs*)

The result of dropping *Empty* terms from a list of resource terms is a subset of the original list

lemma *remove-all-empty-subset*:
$x \in$ *set* (*remove-all-empty xs*) $\Longrightarrow x \in$ *set xs*
⟨*proof*⟩

If there are no *Empty* terms then removing them is the same as not doing anything

lemma *remove-all-empty-none*:
¬ *list-ex is-Empty xs* $\Longrightarrow$ *remove-all-empty xs* = *xs*
⟨*proof*⟩

There are no *Empty* terms left after they are removed

lemma *remove-all-empty-result*:
¬ *list-ex is-Empty* (*remove-all-empty xs*)

⟨*proof*⟩

Removing *Empty* terms distributes over appending lists

lemma *remove-all-empty-append*:
$remove\text{-}all\text{-}empty\ (xs\ @\ ys) = remove\text{-}all\text{-}empty\ xs\ @\ remove\text{-}all\text{-}empty\ ys$
⟨*proof*⟩

Removing *Empty* terms distributes over constructing lists

lemma *remove-all-empty-Cons*:
$remove\text{-}all\text{-}empty\ (x\ \#\ xs) = remove\text{-}all\text{-}empty\ [x]\ @\ remove\text{-}all\text{-}empty\ xs$
⟨*proof*⟩

Removing *Empty* terms from children of a parallel resource term results in an equivalent term

lemma *remove-all-empty-equiv*:
$Parallel\ xs \sim Parallel\ (remove\text{-}all\text{-}empty\ xs)$
⟨*proof*⟩

Removing *Empty* terms does not affect the atom sets

lemma *set1-res-term-remove-all-empty* [*simp*]:
$\bigcup(set1\text{-}res\text{-}term\ `\ set\ (remove\text{-}all\text{-}empty\ xs)) = \bigcup(set1\text{-}res\text{-}term\ `\ set\ xs)$
⟨*proof*⟩
lemma *set2-res-term-remove-all-empty* [*simp*]:
$\bigcup(set2\text{-}res\text{-}term\ `\ set\ (remove\text{-}all\text{-}empty\ xs)) = \bigcup(set2\text{-}res\text{-}term\ `\ set\ xs)$
⟨*proof*⟩

2.6 Merging Nested *Parallel* Terms in a List

Similarly, it is sometimes useful to be able to take a list of terms and merge the children of any *Parallel* term in it up into the list itself

primrec $merge\text{-}all\text{-}parallel :: ('a,\ 'b)\ res\text{-}term\ list \Rightarrow ('a,\ 'b)\ res\text{-}term\ list$
where
$merge\text{-}all\text{-}parallel\ [] = []$
$|\ merge\text{-}all\text{-}parallel\ (x\#xs) =$
$(case\ x\ of\ Parallel\ y \Rightarrow y\ @\ merge\text{-}all\text{-}parallel\ xs\ |\ \text{-} \Rightarrow x\ \#\ merge\text{-}all\text{-}parallel\ xs)$

If there are no *Parallel* terms then merging them is the same as not doing anything

lemma *merge-all-parallel-none*:
$\neg\ list\text{-}ex\ is\text{-}Parallel\ xs \Longrightarrow merge\text{-}all\text{-}parallel\ xs = xs$
⟨*proof*⟩

If no element of the input list has itself nested *Parallel* terms then there will be none left after merging *Parallel* terms in the list

lemma *merge-all-parallel-result*:
assumes $\bigwedge ys.\ Parallel\ ys \in set\ xs \Longrightarrow \neg\ list\text{-}ex\ is\text{-}Parallel\ ys$

shows ¬ *list-ex is-Parallel* (*merge-all-parallel xs*)
⟨*proof*⟩

Merging nested *Parallel* terms distributes over appending lists

lemma *merge-all-parallel-append*:
merge-all-parallel (*xs* @ *ys*) = *merge-all-parallel xs* @ *merge-all-parallel ys*
⟨*proof*⟩

Merging *Parallel* terms distributes over constructing lists

lemma *merge-all-parallel-Cons*:
merge-all-parallel (*x* # *xs*) = *merge-all-parallel* [*x*] @ *merge-all-parallel xs*
⟨*proof*⟩

Merging *Parallel* terms nested in another *Parallel* term results in an equivalent term

lemma *merge-all-parallel-equiv*:
Parallel xs ∼ *Parallel* (*merge-all-parallel xs*)
⟨*proof*⟩

If the output of *merge-all-parallel* contains *Empty* then:

- It was nested in one of the input elements, or
- It was in the input.

lemma *merge-all-parallel-has-empty*:
assumes *list-ex is-Empty* (*merge-all-parallel xs*)
obtains *ys* **where** *Parallel ys* ∈ *set xs* **and** *list-ex is-Empty ys*
| *list-ex is-Empty xs*
⟨*proof*⟩

Merging *Parallel* terms does not affect the atom sets

lemma *set1-res-term-merge-all-parallel* [*simp*]:
$\bigcup$(*set1-res-term* ‘ *set* (*merge-all-parallel xs*)) = $\bigcup$(*set1-res-term* ‘ *set xs*)
⟨*proof*⟩
lemma *set2-res-term-merge-all-parallel* [*simp*]:
$\bigcup$(*set2-res-term* ‘ *set* (*merge-all-parallel xs*)) = $\bigcup$(*set2-res-term* ‘ *set xs*)
⟨*proof*⟩

end
theory *ResNormalForm*
imports
ResTerm
Util
begin

3 Resource Term Normal Form

A resource term is normalised when:

- It is a leaf node, or
- It is an internal node with all children normalised and additionally:
  - If it is a parallel resource then none of its children are *Empty* or *Parallel* and it has more than one child.

primrec *normalised* :: $('a, 'b)$ *res-term* $\Rightarrow$ *bool*
where
normalised Empty = *True*
| *normalised Anything* = *True*
| *normalised* (*Res x*) = *True*
| *normalised* (*Copyable x*) = *True*
| *normalised* (*Parallel xs*) =
(*list-all normalised xs* $\wedge$
list-all ($\lambda x.\ \neg$ *is-Empty x*) *xs* $\wedge$ *list-all* ($\lambda x.\ \neg$ *is-Parallel x*) *xs* $\wedge$
1 $<$ *length xs*)
| *normalised* (*NonD x y*) = (*normalised x* $\wedge$ *normalised y*)
| *normalised* (*Executable x y*) = (*normalised x* $\wedge$ *normalised y*)
| *normalised* (*Repeatable x y*) = (*normalised x* $\wedge$ *normalised y*)

The fact that a term is not normalised can be split into cases

lemma *not-normalised-cases*:
assumes $\neg$ *normalised x*
obtains
(*Parallel-Child*) *xs* **where** *x* = *Parallel xs* **and** *list-ex* ($\lambda x.\ \neg$ *normalised x*) *xs*
| (*Parallel-Empty*) *xs* **where** *x* = *Parallel xs* **and** *list-ex is-Empty xs*
| (*Parallel-Par*) *xs* **where** *x* = *Parallel xs* **and** *list-ex is-Parallel xs*
| (*Parallel-Nil*) *x* = *Parallel* []
| (*Parallel-Singleton*) *a* **where** *x* = *Parallel* [*a*]
| (*NonD-L*) *a b* **where** *x* = *NonD a b* **and** $\neg$ *normalised a*
| (*NonD-R*) *a b* **where** *x* = *NonD a b* **and** $\neg$ *normalised b*
| (*Executable-L*) *a b* **where** *x* = *Executable a b* **and** $\neg$ *normalised a*
| (*Executable-R*) *a b* **where** *x* = *Executable a b* **and** $\neg$ *normalised b*
| (*Repeatable-L*) *a b* **where** *x* = *Repeatable a b* **and** $\neg$ *normalised a*
| (*Repeatable-R*) *a b* **where** *x* = *Repeatable a b* **and** $\neg$ *normalised b*
$\langle$*proof*$\rangle$

When a *Parallel* term is not normalised then it can be useful to obtain the first term in it that is *Empty*, *Parallel* or not normalised.

lemma *obtain-first-parallel*:
assumes *list-ex is-Parallel xs*
obtains *a b c* **where** *xs* = *a* @ [*Parallel b*] @ *c* **and** *list-all* ($\lambda x.\ \neg$ *is-Parallel x*) *a*

$\langle proof \rangle$
lemma *obtain-first-empty*:
assumes *list-ex is-Empty xs*
obtains $a\ b\ c$ **where** $xs = a\ @\ [Empty]\ @\ c$ **and** *list-all* $(\lambda x.\ \neg\ is\text{-}Empty\ x)\ a$
$\langle proof \rangle$
lemma *obtain-first-unnormalised*:
assumes *list-ex* $(\lambda x.\ \neg\ normalised\ x)\ xs$
obtains $a\ b\ c$ **where** $xs = a\ @\ [b]\ @\ c$ **and** *list-all normalised a* **and** $\neg$ *normalised b*
$\langle proof \rangle$

Mapping functions over a resource term does not change whether it is normalised

lemma *normalised-map*:
normalised $(map\text{-}res\text{-}term\ f\ g\ x) = normalised\ x$
$\langle proof \rangle$

If a *Parallel* term is normalised then so are all its children

lemma *normalised-parallel-children*:
$[\![normalised\ (Parallel\ xs);\ x \in set\ xs]\!] \Longrightarrow normalised\ x$
$\langle proof \rangle$

Normalised *Parallel* term has as parallel parts exactly its direct children

lemma *normalised-parallel-parts-eq*:
$normalised\ (Parallel\ xs) \Longrightarrow parallel\text{-}parts\ (Parallel\ xs) = xs$
$\langle proof \rangle$

Parallelising a list of normalised terms with no nested *Empty* or *Parallel* terms gives normalised result.

lemma *normalised-parallelise*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow normalised\ x$
and $\neg$ *list-ex is-Empty xs*
and $\neg$ *list-ex is-Parallel xs*
shows *normalised* $(parallelise\ xs)$
$\langle proof \rangle$

end
theory *ResNormRewrite*
imports
ResNormalForm
Abstract−Rewriting.Abstract-Rewriting
Util
begin

4 Rewriting Resource Term Normalisation

This resource term normalisation procedure is based on the following rewrite rules:

- $\mathit{Parallel}\ [] \rightarrow \mathit{Empty}$
- $\mathit{Parallel}\ [a] \rightarrow a$
- $\mathit{Parallel}\ (x\ @\ [\mathit{Parallel}\ y]\ @\ z) \rightarrow \mathit{Parallel}\ (x\ @\ y\ @\ z)$
- $\mathit{Parallel}\ (x\ @\ [\mathit{Empty}]\ @\ y) \rightarrow \mathit{Parallel}\ (x\ @\ y)$

  This represents the one-directional, single-step version of resource term equivalence. Note that the last rule must be made explicit here, because its counterpart theorem $\mathit{Parallel}\ (?x\ @\ [\mathit{Empty}]\ @\ ?y) \sim \mathit{Parallel}\ (?x\ @\ ?y)$ can only be derived thanks to symmetry.

4.1 Rewriting Relation

The rewriting relation contains a rewriting rule for each introduction rule of $(\sim)$ except for symmetry and transitivity, and an explicit rule for $\mathit{Parallel}\ (?x\ @\ [\mathit{Empty}]\ @\ ?y) \sim \mathit{Parallel}\ (?x\ @\ ?y)$.

inductive *res-term-rewrite* :: $('a,\ 'b)$ *res-term* $\Rightarrow$ $('a,\ 'b)$ *res-term* $\Rightarrow$ *bool* **where**
empty: *res-term-rewrite Empty Empty*
| *anything*: *res-term-rewrite Anything Anything*
| *res*: *res-term-rewrite* (*Res x*) (*Res x*)
| *copyable*: *res-term-rewrite* (*Copyable x*) (*Copyable x*)
| *nil*: *res-term-rewrite* (*Parallel* []) *Empty*
| *singleton*: *res-term-rewrite* (*Parallel* [*a*]) *a*
| *merge*: *res-term-rewrite* (*Parallel* (*x* @ [*Parallel y*] @ *z*)) (*Parallel* (*x* @ *y* @ *z*))
| *drop*: *res-term-rewrite* (*Parallel* (*x* @ [*Empty*] @ *z*)) (*Parallel* (*x* @ *z*))
| *parallel*: *list-all2 res-term-rewrite xs ys* $\Longrightarrow$ *res-term-rewrite* (*Parallel xs*) (*Parallel ys*)
| *nondet*: $[\![$*res-term-rewrite x y*; *res-term-rewrite u v*$]\!]$ $\Longrightarrow$ *res-term-rewrite* (*NonD x u*) (*NonD y v*)
| *executable*: $[\![$*res-term-rewrite x y*; *res-term-rewrite u v*$]\!]$ $\Longrightarrow$
res-term-rewrite (*Executable x u*) (*Executable y v*)
| *repeatable*: $[\![$*res-term-rewrite x y*; *res-term-rewrite u v*$]\!]$ $\Longrightarrow$
res-term-rewrite (*Repeatable x u*) (*Repeatable y v*)

hide-fact (**open**) *empty anything res copyable nil singleton merge drop parallel nondet executable*
repeatable

⟨*ML*⟩

The rewrite relation is reflexive

lemma *refl* [*simp*]:
res-term-rewrite x x
⟨*proof*⟩

lemma *parallel-one*:
$res\text{-}term\text{-}rewrite\ a\ b \Longrightarrow res\text{-}term\text{-}rewrite\ (Parallel\ (xs\ @\ [a]\ @\ ys))\ (Parallel\ (xs\ @\ [b]\ @\ ys))$
⟨*proof*⟩

⟨*ML*⟩

Every term rewrites to an equivalent term

lemma *res-term-rewrite-imp-equiv*:
$res\text{-}term\text{-}rewrite\ x\ y \Longrightarrow x \sim y$
⟨*proof*⟩

By transitivity of the equivalence this holds for transitive closure of the rewriting

lemma *res-term-rewrite-trancl-imp-equiv*:
$res\text{-}term\text{-}rewrite^{++}\ x\ y \Longrightarrow x \sim y$
⟨*proof*⟩

Normalised terms have no distinct term to which they transition

lemma *res-term-rewrite-normalised*:
assumes *normalised x*
shows $\nexists y.\ res\text{-}term\text{-}rewrite\ x\ y \wedge x \neq y$
⟨*proof*⟩

lemma *res-term-rewrite-normalisedD*:
$[\![res\text{-}term\text{-}rewrite\ x\ y;\ normalised\ x]\!] \Longrightarrow x = y$
⟨*proof*⟩

Whereas other terms have a distinct term to which they transition

lemma *res-term-rewrite-not-normalised*:
assumes $\neg$ *normalised x*
shows $\exists y.\ res\text{-}term\text{-}rewrite\ x\ y \wedge x \neq y$
⟨*proof*⟩

Therefore a term is normalised iff it rewrites only back to itself

lemma *normalised-is-rewrite-refl*:
$normalised\ x = (\forall y.\ res\text{-}term\text{-}rewrite\ x\ y \longrightarrow x = y)$
⟨*proof*⟩

Every term rewrites to one of at most equal size

lemma *res-term-rewrite-not-increase-size*:
$res\text{-}term\text{-}rewrite\ x\ y \Longrightarrow size\text{-}res\text{-}term\ f\ g\ y \leq size\text{-}res\text{-}term\ f\ g\ x$
⟨*proof*⟩

4.2 Rewriting Bound

There is an upper bound to how many rewriting steps could be applied to a term. We find it by considering the worst (most un-normalised) possible case of each node.

primrec *res-term-rewrite-bound* :: $('a, 'b)$ *res-term* $\Rightarrow$ *nat*
where
res-term-rewrite-bound Empty = *0*
| *res-term-rewrite-bound Anything* = *0*
| *res-term-rewrite-bound* (*Res a*) = *0*
| *res-term-rewrite-bound* (*Copyable x*) = *0*
| *res-term-rewrite-bound* (*Parallel xs*) =
sum-list (*map res-term-rewrite-bound xs*) + *length xs* + *1*
— All the steps of the children, plus one for every child that could need to be merged/dropped and another if in the end there are less than two children.
| *res-term-rewrite-bound* (*NonD x y*) = *res-term-rewrite-bound x* + *res-term-rewrite-bound y*
| *res-term-rewrite-bound* (*Executable x y*) = *res-term-rewrite-bound x* + *res-term-rewrite-bound y*
| *res-term-rewrite-bound* (*Repeatable x y*) = *res-term-rewrite-bound x* + *res-term-rewrite-bound y*

For un-normalised terms the bound is non-zero

lemma *res-term-rewrite-bound-not-normalised*:
$\neg$ *normalised x* $\Longrightarrow$ *res-term-rewrite-bound x* $\neq$ *0*
⟨*proof*⟩

Rewriting relation does not increase this bound

lemma *res-term-rewrite-non-increase-bound*:
res-term-rewrite x y $\Longrightarrow$ *res-term-rewrite-bound y* $\leq$ *res-term-rewrite-bound x*
⟨*proof*⟩

4.3 Step

The rewriting step function implements a specific algorithm for the rewriting relation by picking one approach where the relation allows multiple rewriting paths. To help define its parallel resource case, we first define a function to remove one *Empty* term from a list and another to merge the children of one *Parallel* term up into the containing list of terms.

4.3.1 Removing One Empty

Remove the first *Empty* from a list of term

fun *remove-one-empty* :: $('a, 'b)$ *res-term list* $\Rightarrow$ $('a, 'b)$ *res-term list*
where
remove-one-empty [] = []
| *remove-one-empty* (*Empty* # *xs*) = *xs*
| *remove-one-empty* (*x* # *xs*) = *x* # *remove-one-empty xs*

lemma *remove-one-empty-cons* [*simp*]:
is-Empty x $\Longrightarrow$ *remove-one-empty* (*x* # *xs*) = *xs*
$\neg$ *is-Empty x* $\Longrightarrow$ *remove-one-empty* (*x* # *xs*) = *x* # *remove-one-empty xs*

⟨*proof*⟩

lemma *remove-one-empty-append*:
$list\text{-}all\ (\lambda x.\ \neg\ is\text{-}Empty\ x)\ a \Longrightarrow remove\text{-}one\text{-}empty\ (a\ @\ d) = a\ @\ remove\text{-}one\text{-}empty\ d$
⟨*proof*⟩

lemma *remove-one-empty-distinct*:
$list\text{-}ex\ is\text{-}Empty\ xs \Longrightarrow remove\text{-}one\text{-}empty\ xs \neq xs$
⟨*proof*⟩

This is identity when there are no *Empty* terms

lemma *remove-one-empty-none* [*simp*]:
$\neg\ list\text{-}ex\ is\text{-}Empty\ xs \Longrightarrow remove\text{-}one\text{-}empty\ xs = xs$
⟨*proof*⟩

This decreases length by one when there are *Empty* terms

lemma *length-remove-one-empty* [*simp*]:
$list\text{-}ex\ is\text{-}Empty\ xs \Longrightarrow length\ (remove\text{-}one\text{-}empty\ xs) + 1 = length\ xs$
⟨*proof*⟩

Removing an *Empty* term does not increase the size

lemma *remove-one-empty-not-increase-size*:
$size\text{-}res\text{-}term\ f\ g\ (Parallel\ (remove\text{-}one\text{-}empty\ xs)) \leq size\text{-}res\text{-}term\ f\ g\ (Parallel\ xs)$
⟨*proof*⟩

Any *Parallel* term is equivalent to itself with an *Empty* term removed

lemma *remove-one-empty-equiv*:
$Parallel\ xs \sim Parallel\ (remove\text{-}one\text{-}empty\ xs)$
⟨*proof*⟩

Removing an *Empty* term commutes with the resource term map

lemma *remove-one-empty-map*:
$map\ (map\text{-}res\text{-}term\ f\ g)\ (remove\text{-}one\text{-}empty\ xs) = remove\text{-}one\text{-}empty\ (map\ (map\text{-}res\text{-}term\ f\ g)\ xs)$
⟨*proof*⟩

The result of dropping an *Empty* from a list of resource terms is a subset of the original list

lemma *remove-one-empty-subset*:
$x \in set\ (remove\text{-}one\text{-}empty\ xs) \Longrightarrow x \in set\ xs$
⟨*proof*⟩

4.3.2 Merging One Parallel

Merge the first *Parallel* in a list of terms

fun *merge-one-parallel* :: ('a, 'b) *res-term list* ⇒ ('a, 'b) *res-term list*
where
merge-one-parallel [] = []
| *merge-one-parallel* (*Parallel x* # *xs*) = *x* @ *xs*
| *merge-one-parallel* (*x* # *xs*) = *x* # *merge-one-parallel xs*

lemma *merge-one-parallel-cons-not* [*simp*]:
¬ *is-Parallel x* ⟹ *merge-one-parallel* (*x* # *xs*) = *x* # *merge-one-parallel xs*
⟨*proof*⟩

lemma *merge-one-parallel-append*:
list-all (λ*x*. ¬ *is-Parallel x*) *a* ⟹ *merge-one-parallel* (*a* @ *d*) = *a* @ *merge-one-parallel* *d*
for *a d*
⟨*proof*⟩

lemma *merge-one-parallel-distinct*:
list-ex is-Parallel xs ⟹ *merge-one-parallel xs* ≠ *xs*
⟨*proof*⟩

This is identity when there are no *Parallel* terms

lemma *merge-one-parallel-none* [*simp*]:
¬ *list-ex is-Parallel xs* ⟹ *merge-one-parallel xs* = *xs*
⟨*proof*⟩

Merging a *Parallel* term does not increase the size

lemma *merge-one-parallel-not-increase-size*:
size-res-term f g (*Parallel* (*merge-one-parallel xs*)) ≤ *size-res-term f g* (*Parallel* *xs*)
⟨*proof*⟩

Any *Parallel* term is equivalent to itself with a *Parallel* term merged

lemma *merge-one-parallel-equiv*:
Parallel xs ∼ *Parallel* (*merge-one-parallel xs*)
⟨*proof*⟩

Merging a *Parallel* term commutes with the resource term map

lemma *merge-one-parallel-map*:
map (*map-res-term f g*) (*merge-one-parallel xs*) = *merge-one-parallel* (*map* (*map-res-term* *f g*) *xs*)
⟨*proof*⟩

4.3.3 Rewriting Step Function

The rewriting step function itself performs one rewrite for any un-normalised input term. Where there are multiple choices, it proceeds as follows:

- For binary internal nodes (*NonD*, *Executable* and *Repeatable*), first fully rewrite the first child until normalised and only then start rewriting the second.

- For *Parallel* nodes proceed in phases:

  - If any child is not normalised, rewrite all children; otherwise
  - If there is some nested *Parallel* node in the children, merge one up; otherwise
  - If there is some *Empty* node in the children, remove one; otherwise
  - If there are no children, then return *Empty*; otherwise
  - If there is exactly one child, then return that term; otherwise
  - Do nothing and return the same term.

primrec *step* :: ('*a*, '*b*) *res-term* ⇒ ('*a*, '*b*) *res-term*
  where
    step Empty = *Empty*
  | *step Anything* = *Anything*
  | *step* (*Res x*) = *Res x*
  | *step* (*Copyable x*) = *Copyable x*
  | *step* (*NonD x y*) =
      (*if* ¬ *normalised x then NonD* (*step x*) *y*
        else if ¬ *normalised y then NonD x* (*step y*)
        else NonD x y)
  | *step* (*Executable x y*) =
      (*if* ¬ *normalised x then Executable* (*step x*) *y*
        else if ¬ *normalised y then Executable x* (*step y*)
        else Executable x y)
  | *step* (*Repeatable x y*) =
      (*if* ¬ *normalised x then Repeatable* (*step x*) *y*
        else if ¬ *normalised y then Repeatable x* (*step y*)
        else Repeatable x y)
  | *step* (*Parallel xs*) =
      (*if list-ex* (λ*x*. ¬ *normalised x*) *xs then Parallel* (*map step xs*)
        else if list-ex is-Parallel xs then Parallel (*merge-one-parallel xs*)
        else if list-ex is-Empty xs then Parallel (*remove-one-empty xs*)
        else (*case xs of*
              [] ⇒ *Empty*
            | [*a*] ⇒ *a*
            | - ⇒ *Parallel xs*))

Case split and induction for step fully expanded

lemma *step-cases*
  [*case-names Empty Anything Res Copyable NonD-L NonD-R NonD Executable-L Executable-R Executable*
        Repeatable-L Repeatable-R Repeatable Par-Norm Par-Par Par-Empty Par-Nil Par-Single

Par]:
assumes $x = Empty \Longrightarrow P$
and $x = Anything \Longrightarrow P$
and $\bigwedge a.\ x = Res\ a \Longrightarrow P$
and $\bigwedge u.\ x = Copyable\ u \Longrightarrow P$
and $\bigwedge u\ v.\ [\![\neg\ normalised\ u;\ x = NonD\ u\ v]\!] \Longrightarrow P$
and $\bigwedge u\ v.\ [\![normalised\ u;\ \neg\ normalised\ v;\ x = NonD\ u\ v]\!] \Longrightarrow P$
and $\bigwedge u\ v.\ [\![normalised\ u;\ normalised\ v;\ x = NonD\ u\ v]\!] \Longrightarrow P$
and $\bigwedge u\ v.\ [\![\neg\ normalised\ u;\ x = Executable\ u\ v]\!] \Longrightarrow P$
and $\bigwedge u\ v.\ [\![normalised\ u;\ \neg\ normalised\ v;\ x = Executable\ u\ v]\!] \Longrightarrow P$
and $\bigwedge u\ v.\ [\![normalised\ u;\ normalised\ v;\ x = Executable\ u\ v]\!] \Longrightarrow P$
and $\bigwedge u\ v.\ [\![\neg\ normalised\ u;\ x = Repeatable\ u\ v]\!] \Longrightarrow P$
and $\bigwedge u\ v.\ [\![normalised\ u;\ \neg\ normalised\ v;\ x = Repeatable\ u\ v]\!] \Longrightarrow P$
and $\bigwedge u\ v.\ [\![normalised\ u;\ normalised\ v;\ x = Repeatable\ u\ v]\!] \Longrightarrow P$
and $\bigwedge xs.\ [\![x = Parallel\ xs;\ \exists\,a.\ a \in set\ xs \wedge \neg\ normalised\ a]\!] \Longrightarrow P$
and $\bigwedge xs.\ [\![x = Parallel\ xs;\ \forall\,a.\ a \in set\ xs \longrightarrow normalised\ a;\ list\text{-}ex\ is\text{-}Parallel\ xs]\!] \Longrightarrow P$
and $\bigwedge xs.\ [\![x = Parallel\ xs;\ \forall\,a.\ a \in set\ xs \longrightarrow normalised\ a;$
$list\text{-}all\ (\lambda x.\ \neg\ is\text{-}Parallel\ x)\ xs;\ list\text{-}ex\ is\text{-}Empty\ xs]\!] \Longrightarrow P$
and $x = Parallel\ [] \Longrightarrow P$
and $\bigwedge u.\ [\![x = Parallel\ [u];\ normalised\ u;\ \neg\ is\text{-}Parallel\ u;\ \neg\ is\text{-}Empty\ u]\!] \Longrightarrow P$
and $\bigwedge v\ vb\ vc.\ [\![x = Parallel\ (v\ \#\ vb\ \#\ vc);\ \forall\,a.\ a \in set\ (v\ \#\ vb\ \#\ vc) \longrightarrow normalised\ a;$
$list\text{-}all\ (\lambda x.\ \neg\ is\text{-}Parallel\ x)\ (v\ \#\ vb\ \#\ vc);$
$list\text{-}all\ (\lambda x.\ \neg\ is\text{-}Empty\ x)\ (v\ \#\ vb\ \#\ vc)]\!]$
$\Longrightarrow P$
shows P
⟨*proof*⟩

lemma *step-induct*
[*case-names Empty Anything Res Copyable NonD-L NonD-R NonD Executable-L Executable-R Executable*
Repeatable-L Repeatable-R Repeatable Par-Norm Par-Par Par-Empty Par-Nil Par-Single
Par]:
assumes $P\ Empty$
and $P\ Anything$
and $\bigwedge a.\ P\ (Res\ a)$
and $\bigwedge x.\ P\ (Copyable\ x)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ \neg\ normalised\ x]\!] \Longrightarrow P\ (NonD\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ normalised\ x;\ \neg\ normalised\ y]\!] \Longrightarrow P\ (NonD\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ normalised\ x;\ normalised\ y]\!] \Longrightarrow P\ (NonD\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ \neg\ normalised\ x]\!] \Longrightarrow P\ (Executable\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ normalised\ x;\ \neg\ normalised\ y]\!] \Longrightarrow P\ (Executable\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ normalised\ x;\ normalised\ y]\!] \Longrightarrow P\ (Executable\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ \neg\ normalised\ x]\!] \Longrightarrow P\ (Repeatable\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ normalised\ x;\ \neg\ normalised\ y]\!] \Longrightarrow P\ (Repeatable\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ normalised\ x;\ normalised\ y]\!] \Longrightarrow P\ (Repeatable\ x\ y)$

and $\bigwedge xs.\ [\![\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x;\ \exists\, a.\ a \in set\ xs \wedge \neg\ normalised\ a]\!] \Longrightarrow P\ (Parallel\ xs)$
and $\bigwedge xs.\ [\![\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x;\ \forall\, a.\ a \in set\ xs \longrightarrow normalised\ a;\ list\text{-}ex\ is\text{-}Parallel\ xs]\!]$
$\Longrightarrow P\ (Parallel\ xs)$
and $\bigwedge xs.\ [\![\ \bigwedge x.\ x \in set\ xs \Longrightarrow P\ x;\ \forall\, a.\ a \in set\ xs \longrightarrow normalised\ a$
$;\ list\text{-}all\ (\lambda x.\ \neg\ is\text{-}Parallel\ x)\ xs;\ list\text{-}ex\ is\text{-}Empty\ xs]\!]$
$\Longrightarrow P\ (Parallel\ xs)$
and $P\ (Parallel\ [])$
and $\bigwedge u.\ [\![P\ u;\ normalised\ u;\ \neg\ is\text{-}Parallel\ u;\ \neg\ is\text{-}Empty\ u]\!] \Longrightarrow P\ (Parallel\ [u])$
and $\bigwedge v\ vb\ vc.$
$[\![\ \bigwedge x.\ x \in set\ (v\ \#\ vb\ \#\ vc) \Longrightarrow P\ x;\ \forall\, a.\ a \in set\ (v\ \#\ vb\ \#\ vc) \longrightarrow normalised\ a$
$;\ list\text{-}all\ (\lambda x.\ \neg\ is\text{-}Parallel\ x)\ (v\ \#\ vb\ \#\ vc)$
$;\ list\text{-}all\ (\lambda x.\ \neg\ is\text{-}Empty\ x)\ (v\ \#\ vb\ \#\ vc)]\!]$
$\Longrightarrow P\ (Parallel\ (v\ \#\ vb\ \#\ vc))$
shows $P\ x$
⟨*proof*⟩

Variant of induction with some relevant step results is also useful

lemma *step-induct′*
[*case-names Empty Anything Res Copyable NonD-L NonD-R NonD Executable-L Executable-R Executable Repeatable-L Repeatable-R Repeatable Par-Norm Par-Par Par-Empty Par-Nil Par-Single Par*]:
assumes $P\ Empty$
and $P\ Anything$
and $\bigwedge a.\ P\ (Res\ a)$
and $\bigwedge x.\ P\ (Copyable\ x)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ \neg\ normalised\ x;\ step\ (NonD\ x\ y) = NonD\ (step\ x)\ y]\!] \Longrightarrow P\ (NonD\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ normalised\ x;\ \neg\ normalised\ y;\ step\ (NonD\ x\ y) = NonD\ x\ (step\ y)]\!]$
$\Longrightarrow P\ (NonD\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ normalised\ x;\ normalised\ y;\ step\ (NonD\ x\ y) = NonD\ x\ y]\!]$
$\Longrightarrow P\ (NonD\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ \neg\ normalised\ x;\ step\ (Executable\ x\ y) = Executable\ (step\ x)\ y]\!]$
$\Longrightarrow P\ (Executable\ x\ y)$
and $\bigwedge x\ y.\ [\![\ P\ x;\ P\ y;\ normalised\ x;\ \neg\ normalised\ y$
$;\ step\ (Executable\ x\ y) = Executable\ x\ (step\ y)]\!]$
$\Longrightarrow P\ (Executable\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ normalised\ x;\ normalised\ y;\ step\ (Executable\ x\ y) = Executable\ x\ y]\!]$
$\Longrightarrow P\ (Executable\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y;\ \neg\ normalised\ x;\ step\ (Repeatable\ x\ y) = Repeatable\ (step$

$x)$ $y]\!]$
$\Longrightarrow P$ $(Repeatable\ x\ y)$
and $\bigwedge x\ y.$ $[\![$ $P\ x;$ $P\ y;$ $normalised\ x;$ $\neg$ $normalised\ y$
; $step$ $(Repeatable\ x\ y) = Repeatable\ x\ (step\ y)]\!]$
$\Longrightarrow P$ $(Repeatable\ x\ y)$
and $\bigwedge x\ y.$ $[\![P\ x;$ $P\ y;$ $normalised\ x;$ $normalised\ y;$ $step$ $(Repeatable\ x\ y) =$ $Repeatable\ x\ y]\!]$
$\Longrightarrow P$ $(Repeatable\ x\ y)$
and $\bigwedge xs.$ $[\![\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x;$ $\exists\, a.\ a \in set\ xs \wedge \neg\ normalised\ a$
; $step$ $(Parallel\ xs) = Parallel\ (map\ step\ xs)]\!]$
$\Longrightarrow P$ $(Parallel\ xs)$
and $\bigwedge xs.$ $[\![\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x;$ $\forall\, a.\ a \in set\ xs \longrightarrow normalised\ a;$ $list\text{-}ex$ $is\text{-}Parallel\ xs;$
$step$ $(Parallel\ xs) = Parallel\ (merge\text{-}one\text{-}parallel\ xs)]\!]$
$\Longrightarrow P$ $(Parallel\ xs)$
and $\bigwedge xs.$ $[\![\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x;$ $\forall\, a.\ a \in set\ xs \longrightarrow normalised\ a$
; $list\text{-}all$ $(\lambda x.\ \neg\ is\text{-}Parallel\ x)\ xs;$ $list\text{-}ex$ $is\text{-}Empty\ xs$
; $step$ $(Parallel\ xs) = Parallel\ (remove\text{-}one\text{-}empty\ xs)]\!]$
$\Longrightarrow P$ $(Parallel\ xs)$
and P $(Parallel\ [])$
and $\bigwedge u.$ $[\![P\ u;$ $normalised\ u;$ $\neg$ $is\text{-}Parallel\ u;$ $\neg$ $is\text{-}Empty\ u;$ $step$ $(Parallel\ [u])$ $= u]\!]$
$\Longrightarrow P$ $(Parallel\ [u])$
and $\bigwedge v\ vb\ vc.$
$[\![$ $\bigwedge x.\ x \in set\ (v\ \#\ vb\ \#\ vc) \Longrightarrow P\ x;$ $\forall\, a.\ a \in set\ (v\ \#\ vb\ \#\ vc) \longrightarrow$ $normalised\ a$
; $list\text{-}all$ $(\lambda x.\ \neg\ is\text{-}Parallel\ x)\ (v\ \#\ vb\ \#\ vc)$
; $list\text{-}all$ $(\lambda x.\ \neg\ is\text{-}Empty\ x)\ (v\ \#\ vb\ \#\ vc)$
; $step$ $(Parallel\ (v\ \#\ vb\ \#\ vc)) = Parallel\ (v\ \#\ vb\ \#\ vc)]\!]$
$\Longrightarrow P$ $(Parallel\ (v\ \#\ vb\ \#\ vc))$
shows $P\ x$
⟨*proof*⟩

Set of atoms remains unchanged by rewriting step

lemma *set1-res-term-step* [*simp*]:
set1-res-term (*step* *x*) = *set1-res-term* *x*
⟨*proof*⟩

lemma *set2-res-term-step* [*simp*]:
set2-res-term (*step* *x*) = *set2-res-term* *x*
⟨*proof*⟩

Resource term rewriting relation contains the step function graph. In other words, the step function is a particular strategy implementing that rewriting.

lemma *res-term-rewrite-contains-step*:
res-term-rewrite *x* (*step* *x*)
⟨*proof*⟩

Resource term being normalised is the same as the step not changing it

lemma *normalised-is-step-id*:
  $normalised\ x = (step\ x = x)$
⟨*proof*⟩

So, for normalised terms we can drop any step applied to them

lemma *step-normalised* [*simp*]:
  $normalised\ x \Longrightarrow step\ x = x$
  ⟨*proof*⟩

Rewriting step never increases the term size

lemma *step-not-increase-size*:
  $size\text{-}res\text{-}term\ f\ g\ (step\ x) \leq size\text{-}res\text{-}term\ f\ g\ x$
  ⟨*proof*⟩

Every resource is equivalent to itself after the step

lemma *res-term-equiv-step*:
  $x \sim step\ x$
  ⟨*proof*⟩

Normalisation step commutes with the resource term map

lemma *step-map*:
  $map\text{-}res\text{-}term\ f\ g\ (step\ x) = step\ (map\text{-}res\text{-}term\ f\ g\ x)$
⟨*proof*⟩

Because it implements the rewriting relation, the non-increasing of bound extends to the step

lemmas *res-term-rewrite-bound-step-non-increase* =
  res-term-rewrite-non-increase-bound[*OF res-term-rewrite-contains-step*]

On un-normalised terms, the step actually strictly decreases the bound. While this should also be true of the rewriting relation it implements, the stricter way the step proceeds makes this proof more tractable.

lemma *res-term-rewrite-bound-step-decrease*:
  $\neg\ normalised\ x \Longrightarrow res\text{-}term\text{-}rewrite\text{-}bound\ (step\ x) < res\text{-}term\text{-}rewrite\text{-}bound\ x$
⟨*proof*⟩

4.4 Normalisation Procedure

Rewrite a resource term until normalised

function *normal-rewr* :: $('a,\ 'b)\ res\text{-}term \Rightarrow ('a,\ 'b)\ res\text{-}term$
  where $normal\text{-}rewr\ x = (if\ normalised\ x\ then\ x\ else\ normal\text{-}rewr\ (step\ x))$
  ⟨*proof*⟩

This terminates with the rewriting bound as measure, because the step keeps decreasing it

termination *normal-rewr*

$\langle proof \rangle$

We remove the normalisation procedure definition from the simplifier, because it can loop

lemmas [*simp del*] = *normal-rewr.simps*

However, the terminal case can be safely used for simplification

lemma *normalised-normal-rewr* [*simp*]:
  normalised x $\Longrightarrow$ *normal-rewr x* = *x*
  $\langle proof \rangle$

Normalisation produces actually normalised terms

lemma *normal-rewr-normalised*:
  normalised (*normal-rewr x*)
  $\langle proof \rangle$

Normalisation is idempotent

lemma *normal-rewr-idempotent* [*simp*]:
  normal-rewr (*normal-rewr x*) = *normal-rewr x*
  $\langle proof \rangle$

Normalisation absorbs rewriting step

lemma *normal-rewr-step*:
  normal-rewr x = *normal-rewr* (*step x*)
  $\langle proof \rangle$

Normalisation leaves leaf terms unchanged

lemma *normal-rewr-leaf*:
  normal-rewr Empty = *Empty*
  normal-rewr Anything = *Anything*
  normal-rewr (*Res x*) = *Res x*
  normal-rewr (*Copyable x*) = *Copyable x*
  $\langle proof \rangle$

Normalisation passes through *NonD*, *Executable* and *Repeatable* constructors

lemma *normal-rewr-nondet*:
  normal-rewr (*NonD x y*) = *NonD* (*normal-rewr x*) (*normal-rewr y*)
$\langle proof \rangle$
lemma *normal-rewr-executable*:
  normal-rewr (*Executable x y*) = *Executable* (*normal-rewr x*) (*normal-rewr y*)
$\langle proof \rangle$
lemma *normal-rewr-repeatable*:
  normal-rewr (*Repeatable x y*) = *Repeatable* (*normal-rewr x*) (*normal-rewr y*)
$\langle proof \rangle$

Normalisation simplifies empty *Parallel* terms

lemma *normal-rewr-parallel-empty*:

$normal\text{-}rewr\ (Parallel\ []) = Empty$
$\langle proof \rangle$

Every resource is equivalent to its normalisation

lemma *res-term-equiv-normal-rewr*:
$x \sim normal\text{-}rewr\ x$
$\langle proof \rangle$

And, by transitivity, resource terms with equal normalisations are equivalent

lemma *normal-rewr-imp-equiv*:
$normal\text{-}rewr\ x = normal\text{-}rewr\ y \Longrightarrow x \sim y$
$\langle proof \rangle$

Resource normalisation commutes with the resource map

lemma *normal-rewr-map*:
$map\text{-}res\text{-}term\ f\ g\ (normal\text{-}rewr\ x) = normal\text{-}rewr\ (map\text{-}res\text{-}term\ f\ g\ x)$
$\langle proof \rangle$

Normalisation is contained in transitive closure of the rewriting

lemma *res-term-rewrite-tranclp-normal-rewr*:
$res\text{-}term\text{-}rewrite^{++}\ x\ (normal\text{-}rewr\ x)$
$\langle proof \rangle$

4.5 As Abstract Rewriting System

The normalisation procedure described above implements an abstract rewriting system. Their theory allows us to prove that equality of normal forms is the same as term equivalence by reasoning about how equivalent terms are joinable by the rewriting.

4.5.1 Rewriting System Properties

In the ARS mechanisation normal forms are terminal elements of the rewriting relation, while in our case they are fixpoints. To interface with that property, we use the irreflexive graph of *step*.

definition $step\text{-}irr :: ('a, 'b)\ res\text{-}term\ rel$
where $step\text{-}irr = \{(x,y).\ x \neq y \wedge step\ x = y\}$

lemma *step-irr-inI*:
$x \neq step\ x \Longrightarrow (x,\ step\ x) \in step\text{-}irr$
$\langle proof \rangle$

Graph of *normal-rewr* is in the transitive-reflexive closure of irreflexive step

lemma *normal-rewr-in-step-rtrancl*:
$(x,\ normal\text{-}rewr\ x) \in step\text{-}irr^*$
$\langle proof \rangle$

Normal forms of irreflexive step are exactly the normalised terms

lemma *step-nf-is-normalised*:
  NF step-irr = {*x*. *normalised x*}
⟨*proof*⟩

As such, every value of *normal-rewr* is a normal form of irreflexive step

lemma *normal-rewr-NF* [*simp*]:
  normal-rewr x ∈ *NF step-irr*
  ⟨*proof*⟩

Terms related by reflexive-transitive step are equivalent

lemma *step-rtrancl-equivalent*:
  $(a,b) \in$ *step-irr** $\Longrightarrow a \sim b$
⟨*proof*⟩

Irreflexive step is locally and strongly confluent because it's part of a function

lemma *step-irr-locally-confluent*:
  WCR step-irr
  ⟨*proof*⟩

lemma *step-irr-strongly-confluent*:
  strongly-confluent step-irr
  ⟨*proof*⟩

Therefore it is Church-Rosser and has unique normal forms

lemma *step-CR*: *CR step-irr*
  ⟨*proof*⟩
lemma *step-UNC*: *UNC step-irr*
  ⟨*proof*⟩
lemma *step-UNF*: *UNF step-irr*
  ⟨*proof*⟩

Irreflexive step is strongly normalising because it decreases the well-founded rewriting bound

lemma *step-SN*:
  SN step-irr
  ⟨*proof*⟩

Normalisability relation of irreflexive step is exactly the graph of *normal-rewr*

lemma *step-normalizability-normal-rewr*:
  step-irr$^!$ = {(*x*, *y*). *y* = *normal-rewr x*}
⟨*proof*⟩

The unique normal form, *the-NF* in the ARS language, is *normal-rewr*

lemma *step-irr-the-NF* [*simp*]:
  the-NF step-irr x = *normal-rewr x*
  ⟨*proof*⟩

Terms related by reflexive-transitive step have the same normal form

lemma *step-rtrancl-eq-normal*:
$(x,y) \in \textit{step-irr}^* \Longrightarrow \textit{normal-rewr}\ x = \textit{normal-rewr}\ y$
$\langle \textit{proof} \rangle$

4.5.2 *NonD* Joinability

Two *NonD* terms are joinable if their corresponding children are joinable

lemma *step-rtrancl-nondL*:
$(x,u) \in \textit{step-irr}^* \Longrightarrow (\textit{NonD}\ x\ y,\ \textit{NonD}\ u\ y) \in \textit{step-irr}^*$
$\langle \textit{proof} \rangle$

lemma *step-rtrancl-nondR*:
$[\![(y,v) \in \textit{step-irr}^*;\ \textit{normalised}\ x]\!] \Longrightarrow (\textit{NonD}\ x\ y,\ \textit{NonD}\ x\ v) \in \textit{step-irr}^*$
$\langle \textit{proof} \rangle$

lemma *step-rtrancl-nond*:
$[\![(x,u) \in \textit{step-irr}^*;\ \textit{normalised}\ u;\ (y,v) \in \textit{step-irr}^*]\!] \Longrightarrow (\textit{NonD}\ x\ y,\ \textit{NonD}\ u\ v) \in \textit{step-irr}^*$
$\langle \textit{proof} \rangle$

lemma *step-join-apply-nondet*:
assumes $(x,u) \in \textit{step-irr}^{\downarrow}$ **and** $(y,v) \in \textit{step-irr}^{\downarrow}$ **shows** $(\textit{NonD}\ x\ y,\ \textit{NonD}\ u\ v) \in \textit{step-irr}^{\downarrow}$
$\langle \textit{proof} \rangle$

4.5.3 *Executable* and *Repeatable* Joinability

Two (repeatably) executable resource terms are joinable if their corresponding children are joinable

lemma *step-join-apply-executable*:
$[\![(x,u) \in \textit{step-irr}^{\downarrow};\ (y,v) \in \textit{step-irr}^{\downarrow}]\!] \Longrightarrow (\textit{Executable}\ x\ y,\ \textit{Executable}\ u\ v) \in \textit{step-irr}^{\downarrow}$
$\langle \textit{proof} \rangle$

lemma *step-join-apply-repeatable*:
$[\![(x,u) \in \textit{step-irr}^{\downarrow};\ (y,v) \in \textit{step-irr}^{\downarrow}]\!] \Longrightarrow (\textit{Repeatable}\ x\ y,\ \textit{Repeatable}\ u\ v) \in \textit{step-irr}^{\downarrow}$
$\langle \textit{proof} \rangle$

4.5.4 *Parallel* Joinability

From two lists of joinable terms we can obtain a list of common destination terms

lemma *list-all2-join*:
assumes $\textit{list-all2}\ (\lambda x\ y.\ (x,\ y) \in R^{\downarrow})\ \textit{xs}\ \textit{ys}$
obtains *cs*

where *list-all2* $(\lambda x\ c.\ (x,\ c) \in R^*)$ *xs cs*
and *list-all2* $(\lambda y\ c.\ (y,\ c) \in R^*)$ *ys cs*
⟨*proof*⟩

Every parallel resource term with at least two elements is related to a parallel resource term with the contents normalised

lemma *step-rtrancl-map-normal*:
(*Parallel xs*, *Parallel* (*map normal-rewr xs*)) $\in$ *step-irr**
⟨*proof*⟩

Two lists of joinable terms have the same normal forms

lemma *list-all2-join-normal-eq*:
list-all2 $(\lambda u\ v.\ (u,\ v) \in$ *step-irr*$^{\downarrow})$ *xs ys* $\Longrightarrow$ *map normal-rewr xs* $=$ *map normal-rewr ys*
⟨*proof*⟩

Parallel resource terms whose contents are joinable are themselves joinable

lemma *step-join-apply-parallel*:
assumes *list-all2* $(\lambda u\ v.\ (u,v) \in$ *step-irr*$^{\downarrow})$ *xs ys*
shows (*Parallel xs*, *Parallel ys*) $\in$ *step-irr*$^{\downarrow}$
⟨*proof*⟩

Removing all *Empty* terms absorbs the removal of one

lemma *remove-all-empty-subsumes-remove-one*:
remove-all-empty (*remove-one-empty xs*) $=$ *remove-all-empty xs*
⟨*proof*⟩

For any list with an *Empty* term, removing one strictly decreases their count

lemma *remove-one-empty-count-if-decrease*:
list-ex is-Empty xs $\Longrightarrow$ *count-if is-Empty* (*remove-one-empty xs*) $<$ *count-if is-Empty xs*
⟨*proof*⟩

Removing all *Empty* terms from children of a *Parallel* term, that are already all normalised and none of which are nested *Parallel* terms, is related by transitive and reflexive closure of irreflexive step.

lemma *step-rtrancl-remove-all-empty*:
assumes $\bigwedge x.\ x \in$ *set xs* $\Longrightarrow$ *normalised x*
and $\neg$ *list-ex is-Parallel xs*
shows (*Parallel xs*, *Parallel* (*remove-all-empty xs*)) $\in$ *step-irr**
⟨*proof*⟩

After merging all *Parallel* elements of a list of normalised terms, there remain no more *Parallel* terms in it

lemma *merge-all-parallel-map-normal-result*:
assumes $\bigwedge x.\ x \in$ *set xs* $\Longrightarrow$ *normalised x*
shows $\neg$ *list-ex is-Parallel* (*merge-all-parallel xs*)

$\langle proof \rangle$

For any list with a *Parallel* term, removing one strictly decreases their count if no element contains further nested *Parallel* terms within it

lemma *merge-one-parallel-count-if-decrease*:
assumes *list-ex is-Parallel xs*
and $\bigwedge y\ ys.\ [\![y \in set\ xs;\ y = Parallel\ ys]\!] \Longrightarrow \neg\ list\text{-}ex\ is\text{-}Parallel\ ys$
shows $count\text{-}if\ is\text{-}Parallel\ (merge\text{-}one\text{-}parallel\ xs) < count\text{-}if\ is\text{-}Parallel\ xs$
$\langle proof \rangle$

Merging all *Parallel* terms absorbs the merging of one if no element contains further nested *Parallel* terms within it

lemma *merge-all-parallel-subsumes-merge-one*:
assumes $\bigwedge y\ ys.\ [\![y \in set\ xs;\ y = Parallel\ ys]\!] \Longrightarrow \neg\ list\text{-}ex\ is\text{-}Parallel\ ys$
shows $merge\text{-}all\text{-}parallel\ (merge\text{-}one\text{-}parallel\ xs) = merge\text{-}all\text{-}parallel\ xs$
$\langle proof \rangle$

Merging one *Parallel* term in a list of normalised terms keeps them normalised

lemma *merge-one-parallel-preserve-normalised*:
$[\![\bigwedge x.\ x \in set\ xs \Longrightarrow normalised\ x;\ a \in set\ (merge\text{-}one\text{-}parallel\ xs)]\!] \Longrightarrow normalised\ a$
$\langle proof \rangle$

Merging all *Parallel* terms in a list of normalised terms keeps them normalised

lemma *merge-all-parallel-preserve-normalised*:
$[\![\bigwedge x.\ x \in set\ xs \Longrightarrow normalised\ x;\ a \in set\ (merge\text{-}all\text{-}parallel\ xs)]\!] \Longrightarrow normalised\ a$
$\langle proof \rangle$

Merging all *Parallel* terms from children of a *Parallel* term, that are already all normalised, is related by transitive and reflexive closure of irreflexive step.

lemma *step-rtrancl-merge-all-parallel*:
assumes $\bigwedge x.\ x \in set\ xs \Longrightarrow normalised\ x$
shows $(Parallel\ xs,\ Parallel\ (merge\text{-}all\text{-}parallel\ xs)) \in step\text{-}irr^*$
$\langle proof \rangle$

Thus, there is a general rewriting path that *Parallel* terms take

lemma *step-rtrancl-parallel*:
$(Parallel\ xs,\ Parallel\ (remove\text{-}all\text{-}empty\ (merge\text{-}all\text{-}parallel\ (map\ normal\text{-}rewr\ xs)))) \in step\text{-}irr^*$
$\langle proof \rangle$

4.5.5 Other Helpful Lemmas

For Church-Rosser strongly normalising rewriting systems, joinability is transitive

lemma *CR-SN-join-trans*:
  assumes *CR R*
    and *SN R*
    and $(x, y) \in R^{\downarrow}$
    and $(y, z) \in R^{\downarrow}$
  shows $(x, z) \in R^{\downarrow}$
⟨*proof*⟩

More generally, for such systems, two joinable pairs can be bridged by a third

lemma *CR-SN-join-both*:
  $[\![CR\ R;\ SN\ R;\ (a, b) \in R^{\downarrow};\ (x, y) \in R^{\downarrow};\ (b, y) \in R^{\downarrow}]\!] \Longrightarrow (a, x) \in R^{\downarrow}$
  ⟨*proof*⟩

With irreflexive step being one such rewriting system

lemmas *step-irr-join-trans* = *CR-SN-join-trans*[*OF step-CR step-SN*]
lemmas *step-irr-join-both* = *CR-SN-join-both*[*OF step-CR step-SN*]

Parallel term with no work left in children normalises in three possible ways

lemma *normal-rewr-parallel-cases*:
  assumes $\forall x.\ x \in set\ xs \longrightarrow normalised\ x$
    and ¬ *list-ex is-Empty xs*
    and ¬ *list-ex is-Parallel xs*
  obtains
    (*Parallel*) *normalised* (*Parallel xs*) **and** *normal-rewr* (*Parallel xs*) = *Parallel xs*
  | (*Empty*) *xs* = [] **and** *normal-rewr* (*Parallel xs*) = *Empty*
  | (*Single*) *a* **where** *xs* = [*a*] **and** *normal-rewr* (*Parallel xs*) = *a*
⟨*proof*⟩

For a list of already normalised terms with no *Empty* or *Parallel* terms, the normalisation procedure acts like *parallel-parts* followed by *parallelise*. It only does simplifications related to the number of elements.

lemma *normal-rewr-parallelise*:
  assumes $\forall x.\ x \in set\ xs \longrightarrow normalised\ x$
    and ¬ *list-ex is-Empty xs*
    and ¬ *list-ex is-Parallel xs*
  shows *normal-rewr* (*Parallel xs*) = *parallelise* (*parallel-parts* (*Parallel xs*))
⟨*proof*⟩

Removing all *Empty* terms has no effect on number of *Parallel* terms

lemma *parallel-remove-all-empty*:
  list-ex is-Parallel (*remove-all-empty xs*) = *list-ex is-Parallel xs*

$\langle proof \rangle$

Removing all *Empty* terms is idempotent because there are no *Empty* terms to remove on the second pass

lemma *remove-all-empty-idempotent*:
  shows $\mathit{remove\text{-}all\text{-}empty}\ (\mathit{remove\text{-}all\text{-}empty}\ \mathit{xs}) = \mathit{remove\text{-}all\text{-}empty}\ \mathit{xs}$
  $\langle proof \rangle$

Every *Parallel* term rewrites to the parallelisation of normalised children with all *Empty* terms removed and all *Parallel* terms merged

lemma *normal-rewr-to-parallelise*:
  $\mathit{normal\text{-}rewr}\ (\mathit{Parallel}\ \mathit{xs})$
  $= \mathit{parallelise}\ (\mathit{remove\text{-}all\text{-}empty}\ (\mathit{merge\text{-}all\text{-}parallel}\ (\mathit{map}\ \mathit{normal\text{-}rewr}\ \mathit{xs})))$
$\langle proof \rangle$

Parallel term that normalises to *Empty* must have had no children left after normalising them, merging *Parallel* terms and removing *Empty* terms

lemma *normal-rewr-to-empty*:
  assumes $\mathit{normal\text{-}rewr}\ (\mathit{Parallel}\ \mathit{xs}) = \mathit{Empty}$
    shows $\mathit{remove\text{-}all\text{-}empty}\ (\mathit{merge\text{-}all\text{-}parallel}\ (\mathit{map}\ \mathit{normal\text{-}rewr}\ \mathit{xs})) = []$
  $\langle proof \rangle$

Parallel term that normalises to another *Parallel* must have had those children left after normalising its own, merging *Parallel* terms and removing *Empty* terms

lemma *normal-rewr-to-parallel*:
  assumes $\mathit{normal\text{-}rewr}\ (\mathit{Parallel}\ \mathit{xs}) = \mathit{Parallel}\ \mathit{ys}$
   shows $\mathit{remove\text{-}all\text{-}empty}\ (\mathit{merge\text{-}all\text{-}parallel}\ (\mathit{map}\ \mathit{normal\text{-}rewr}\ \mathit{xs})) = \mathit{remove\text{-}all\text{-}empty}$
ys
$\langle proof \rangle$

Parallel that normalises to anything else must have had that as the only term left after normalising its own, merging *Parallel* terms and removing *Empty* terms

lemma *normal-rewr-to-other*:
  assumes $\mathit{normal\text{-}rewr}\ (\mathit{Parallel}\ \mathit{xs}) = a$
      and $\neg\ \mathit{is\text{-}Empty}\ a$
      and $\neg\ \mathit{is\text{-}Parallel}\ a$
    shows $\mathit{remove\text{-}all\text{-}empty}\ (\mathit{merge\text{-}all\text{-}parallel}\ (\mathit{map}\ \mathit{normal\text{-}rewr}\ \mathit{xs})) = [a]$
  $\langle proof \rangle$

4.5.6 Equivalent Term Joinability

Equivalent resource terms are joinable by irreflexive step

lemma *res-term-equiv-joinable*:
  $x \sim y \Longrightarrow (x,\ y) \in \mathit{step\text{-}irr}^{\downarrow}$

$\langle proof \rangle$

Therefore this rewriting-based normalisation brings equivalent terms to the same normal form

lemma *res-term-equiv-imp-normal-rewr*:
assumes $x \sim y$ **shows** $normal\text{-}rewr\ x = normal\text{-}rewr\ y$
$\langle proof \rangle$

And resource term equivalence is equal to having equal normal forms

theorem *res-term-equiv-is-normal-rewr*:
$x \sim y = (normal\text{-}rewr\ x = normal\text{-}rewr\ y)$
$\langle proof \rangle$

4.6 Term Equivalence as Rewriting Closure

We can now show that $(\sim)$ is the equivalence closure of *res-term-rewrite*.

An equivalence closure is a reflexive, transitive and symmetric closure. In our case, the rewriting is already reflexive, so we only need to verify the symmetric and transitive closure.

As such, the core difficulty in this section is to prove the following equality: $x \sim y = (symclp\ res\text{-}term\text{-}rewrite)^{++}\ x\ y$

One direction is simpler, because rewriting implies equivalence

lemma *res-term-rewrite-equivclp-imp-equiv*:
$(symclp\ res\text{-}term\text{-}rewrite)^{++}\ x\ y \Longrightarrow x \sim y$
$\langle proof \rangle$

Trying to prove the other direction purely through facts about the rewriting itself fails

lemma
$x \sim y \Longrightarrow (symclp\ res\text{-}term\text{-}rewrite)^{++}\ x\ y$
$\langle proof \rangle$

But, we can take advantage of the normalisation procedure to prove it

lemma *res-term-rewrite-equiv-imp-equivclp*:
assumes $x \sim y$
shows $(symclp\ res\text{-}term\text{-}rewrite)^{++}\ x\ y$
$\langle proof \rangle$

Thus, we prove that resource term equivalence is the equivalence closure of the rewriting

lemma *res-term-equiv-is-rewrite-closure*:
$(\sim) = equivclp\ res\text{-}term\text{-}rewrite$
$\langle proof \rangle$

end
theory *ResNormDirect*
  imports *ResNormalForm*
begin

5 Direct Resource Term Normalisation

In this section we define a normalisation procedure for resource terms that directly normalises a term in a single bottom-up pass. This could be considered normalisation by evaluation as opposed to by rewriting.

Note that, while this procedure is more computationally efficient, it is less useful in proofs. In this way it is complemented by rewriting-based normalisation that is less direct but more helpful in inductive proofs.

First, for a list of terms where no *Parallel* term contains an *Empty* term, the order of *merge-all-parallel* and *remove-all-empty* does not matter. This is specifically the case for a list of normalised terms. As such, our choice of order in the normalisation definition does not matter.

lemma *merge-all-parallel-remove-all-empty-comm*:
  assumes $\bigwedge ys.\ Parallel\ ys \in set\ xs \Longrightarrow \neg\ list\text{-}ex\ is\text{-}Empty\ ys$
  shows $merge\text{-}all\text{-}parallel\ (remove\text{-}all\text{-}empty\ xs) = remove\text{-}all\text{-}empty\ (merge\text{-}all\text{-}parallel\ xs)$
  ⟨*proof*⟩

Direct normalisation of resource terms proceeds in a single bottom-up pass. The interesting case is for *Parallel* terms, where any *Empty* and nested *Parallel* children are handled using *parallel-parts* and the resulting list is turned into the simplest term representing its parallel combination using *parallelise*.

primrec $normal\text{-}dir :: ('a,\ 'b)\ res\text{-}term \Rightarrow ('a,\ 'b)\ res\text{-}term$
  where
    $normal\text{-}dir\ Empty = Empty$
  $|\ normal\text{-}dir\ Anything = Anything$
  $|\ normal\text{-}dir\ (Res\ x) = Res\ x$
  $|\ normal\text{-}dir\ (Copyable\ x) = Copyable\ x$
  $|\ normal\text{-}dir\ (Parallel\ xs) =$
      $parallelise\ (merge\text{-}all\text{-}parallel\ (remove\text{-}all\text{-}empty\ (map\ normal\text{-}dir\ xs)))$
  $|\ normal\text{-}dir\ (NonD\ x\ y) = NonD\ (normal\text{-}dir\ x)\ (normal\text{-}dir\ y)$
  $|\ normal\text{-}dir\ (Executable\ x\ y) = Executable\ (normal\text{-}dir\ x)\ (normal\text{-}dir\ y)$
  $|\ normal\text{-}dir\ (Repeatable\ x\ y) = Repeatable\ (normal\text{-}dir\ x)\ (normal\text{-}dir\ y)$

Any resource term is equivalent to its direct normalisation

lemma *normal-dir-equiv*:
  $a \sim normal\text{-}dir\ a$
⟨*proof*⟩

Thus terms with equal normalisation are equivalent

lemma *normal-dir-eq-imp-equiv*:
$\textit{normal-dir } a = \textit{normal-dir } b \Longrightarrow a \sim b$
$\langle \textit{proof} \rangle$

If the output of *merge-all-parallel* still contains a *Parallel* term then it must have been nested in one of the input elements

lemma *merge-all-parallel-has-Parallel*:
assumes *list-ex is-Parallel* (*merge-all-parallel xs*)
obtains *ys*
where *Parallel ys* $\in$ *set xs*
and *list-ex is-Parallel ys*
$\langle \textit{proof} \rangle$

If the output of *remove-all-empty* contains a *Parallel* term then it must have been in the input

lemma *remove-all-empty-has-Parallel*:
assumes *Parallel ys* $\in$ *set* (*remove-all-empty xs*)
shows *Parallel ys* $\in$ *set xs*
$\langle \textit{proof} \rangle$

If a resource term normalises to a *Parallel* term then that does not contain any nested

lemma *normal-dir-no-nested-Parallel*:
$\textit{normal-dir } a = \textit{Parallel } xs \Longrightarrow \neg\ \textit{list-ex is-Parallel } xs$
$\langle \textit{proof} \rangle$

If a resource term normalises to a *Parallel* term then it does not contain *Empty*

lemma *normal-dir-no-nested-Empty*:
$\textit{normal-dir } a = \textit{Parallel } xs \Longrightarrow \neg\ \textit{list-ex is-Empty } xs$
$\langle \textit{proof} \rangle$

Merging *Parallel* terms in a list of normalised terms keeps all terms in the result normalised

lemma *normalised-merge-all-parallel*:
assumes $x \in$ *set* (*merge-all-parallel xs*)
and $\bigwedge x.\ x \in \textit{set } xs \Longrightarrow \textit{normalised } x$
shows *normalised x*
$\langle \textit{proof} \rangle$

Normalisation produces resources in normal form

lemma *normalised-normal-dir*:
normalised (*normal-dir a*)
$\langle \textit{proof} \rangle$

Normalisation does nothing to resource terms in normal form

lemma *normal-dir-normalised*:

$normalised\ x \Longrightarrow normal\text{-}dir\ x = x$
⟨*proof*⟩

Parallelising to anything but *Empty* or *Parallel* means the input list contained just that

lemma *parallelise-eq-Anything* [*simp*]: $(parallelise\ xs = Anything) = (xs = [Anything])$
and *parallelise-eq-Res* [*simp*]: $(parallelise\ xs = Res\ a) = (xs = [Res\ a])$
and *parallelise-eq-Copyable* [*simp*]: $(parallelise\ xs = Copyable\ b) = (xs = [Copyable\ b])$
and *parallelise-eq-NonD* [*simp*]: $(parallelise\ xs = NonD\ x\ y) = (xs = [NonD\ x\ y])$
and *parallelise-eq-Executable* [*simp*]:$(parallelise\ xs = Executable\ x\ y) = (xs = [Executable\ x\ y])$
and *parallelise-eq-Repeatable* [*simp*]:$(parallelise\ xs = Repeatable\ x\ y) = (xs = [Repeatable\ x\ y])$
⟨*proof*⟩

Equivalent resource terms normalise to equal results

lemma *res-term-equiv-normal-dir*:
$a \sim b \Longrightarrow normal\text{-}dir\ a = normal\text{-}dir\ b$
⟨*proof*⟩

Equivalence of resource term is equality of their normal forms

lemma *res-term-equiv-is-normal-dir*:
$a \sim b = (normal\text{-}dir\ a = normal\text{-}dir\ b)$
⟨*proof*⟩

We use this fact to give a code equation for $(\sim)$

lemmas [*code*] = *res-term-equiv-is-normal-dir*

The normal form is unique in each resource term equivalence class

lemma *normal-dir-unique*:
$[\![normal\text{-}dir\ x = x;\ normal\text{-}dir\ y = y;\ x \sim y]\!] \Longrightarrow x = y$
⟨*proof*⟩

end
theory *ResNormCompare*
imports
ResNormDirect
ResNormRewrite
begin

6 Comparison of Resource Term Normalisation

The two normalisation procedures have the same outcome, because they both normalise the term

lemma *normal-rewr-is-normal-dir*:

$normal\text{-}rewr = normal\text{-}dir$
⟨*proof*⟩

With resource term normalisation to decide the equvialence, we can prove that the resource term mapping may render terms equivalent.

lemma
fixes $a\ b :: 'a$ **and** $c :: 'b$
assumes $a \neq b$
obtains $f :: 'a \Rightarrow 'b$ **and** $x\ y$ **where** $map\text{-}res\text{-}term\ f\ g\ x \sim map\text{-}res\text{-}term\ f\ g\ y$
and $\neg\ x \sim y$
⟨*proof*⟩

end
theory *Resource*
imports
ResTerm
ResNormCompare
begin

7 Resources

We define resources as the quotient of resource terms by their equivalence. To decide the equivalence we use resource term normalisation procedures, primarily the one based on rewriting.

7.1 Quotient Type

Resource term mapper satisfies the functor assumptions: it commutes with function composition and mapping identities is itself identity

functor *map-res-term*
⟨*proof*⟩

Resources are resource terms modulo their equivalence

quotient-type $('a, 'b)\ resource = ('a, 'b)\ res\text{-}term\ /\ res\text{-}term\text{-}equiv$
⟨*proof*⟩

lemma *abs-resource-eqI* [*intro*]:
$x \sim y \Longrightarrow abs\text{-}resource\ x = abs\text{-}resource\ y$
⟨*proof*⟩
lemma *abs-resource-eqE* [*elim*]:
$[\![abs\text{-}resource\ x = abs\text{-}resource\ y;\ x \sim y \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

Resource representation then abstraction is identity

lemmas *resource-abs-of-rep* [*simp*] = *Quotient3-abs-rep*[*OF Quotient3-resource*]

Lifted normalisation gives a normalised representative term for a resource

lift-definition *of-resource* :: $('a,\ 'b)$ *resource* $\Rightarrow$ $('a,\ 'b)$ *res-term* **is** *normal-rewr*
⟨*proof*⟩

lemma *of-resource-absorb-normal-rewr* [*simp*]:
normal-rewr (*of-resource* x) = *of-resource* x
⟨*proof*⟩

lemma *of-resource-absorb-normal-dir* [*simp*]:
normal-dir (*of-resource* x) = *of-resource* x
⟨*proof*⟩

Equality of resources can be characterised by equality of representative terms

instantiation *resource* :: (*equal*, *equal*) *equal*
begin

definition *equal-resource* :: $('a,\ 'b)$ *resource* $\Rightarrow$ $('a,\ 'b)$ *resource* $\Rightarrow$ *bool*
where *equal-resource* a b = (*of-resource* a = *of-resource* b)

instance
⟨*proof*⟩
end

7.2 Lifting Bounded Natural Functor Structure

Equivalent terms have equal atom sets

lemma *res-term-equiv-set1* [*simp*]:
$x \sim y \Longrightarrow$ *set1-res-term* x = *set1-res-term* y
⟨*proof*⟩

lemma *res-term-equiv-set2* [*simp*]:
$x \sim y \Longrightarrow$ *set2-res-term* x = *set2-res-term* y
⟨*proof*⟩

BNF structure can be lifted. Proof inspired by Fürer et al. [1].

lift-bnf $('a,\ 'b)$ *resource*
⟨*proof*⟩

Resource map can be given a code equation through the term map

lemma *map-resource-code* [*code*]:
map-resource f g (*abs-resource* x) = *abs-resource* (*map-res-term* f g x)
⟨*proof*⟩

Atom sets of a resource are those sets of its representative term

lemma *set1-resource*:
fixes x :: $('a,\ 'b)$ *resource*
shows *set1-resource* x = *set1-res-term* (*of-resource* x)
⟨*proof*⟩
lemma *set2-resource*:

fixes x :: $('a,\ 'b)$ *resource*
shows *set2-resource* x = *set2-res-term* (*of-resource* x)
⟨*proof*⟩

7.3 Lifting Constructors

All term constructors are easily lifted thanks to the term equivalence being a congruence

lift-definition *Empty* :: $('a,\ 'b)$ *resource*
is *res-term.Empty* ⟨*proof*⟩
lift-definition *Anything* :: $('a,\ 'b)$ *resource*
is *res-term.Anything* ⟨*proof*⟩
lift-definition *Res* :: $'a \Rightarrow ('a,\ 'b)$ *resource*
is *res-term.Res* ⟨*proof*⟩
lift-definition *Copyable* :: $'b \Rightarrow ('a,\ 'b)$ *resource*
is *res-term.Copyable* ⟨*proof*⟩
lift-definition *Parallel* :: $('a,\ 'b)$ *resource list* ⇒ $('a,\ 'b)$ *resource*
is *res-term.Parallel* ⟨*proof*⟩
lift-definition *NonD* :: $('a,\ 'b)$ *resource* ⇒ $('a,\ 'b)$ *resource* ⇒ $('a,\ 'b)$ *resource*
is *res-term.NonD* ⟨*proof*⟩
lift-definition *Executable* :: $('a,\ 'b)$ *resource* ⇒ $('a,\ 'b)$ *resource* ⇒ $('a,\ 'b)$ *resource*
is *res-term.Executable* ⟨*proof*⟩
lift-definition *Repeatable* :: $('a,\ 'b)$ *resource* ⇒ $('a,\ 'b)$ *resource* ⇒ $('a,\ 'b)$ *resource*
is *res-term.Repeatable* ⟨*proof*⟩

lemmas *resource-constr-abs-eq* =
Empty.abs-eq Anything.abs-eq Res.abs-eq Copyable.abs-eq Parallel.abs-eq NonD.abs-eq Executable.abs-eq Repeatable.abs-eq

Resources can be split into cases like terms

lemma *resource-cases*:
fixes r :: $('a,\ 'b)$ *resource*
obtains
(*Empty*) r = *Empty*
| (*Anything*) r = *Anything*
| (*Res*) a **where** r = *Res* a
| (*Copyable*) x **where** r = *Copyable* x
| (*Parallel*) xs **where** r = *Parallel* xs
| (*NonD*) x y **where** r = *NonD* x y
| (*Executable*) x y **where** r = *Executable* x y
| (*Repeatable*) x y **where** r = *Repeatable* x y
⟨*proof*⟩

Resources can be inducted over like terms

lemma *resource-induct* [*case-names Empty Anything Res Copyable Parallel NonD Executable Repeatable*]:
assumes *P Empty*
and *P Anything*

and $\bigwedge a.\ P\ (Res\ a)$
and $\bigwedge x.\ P\ (Copyable\ x)$
and $\bigwedge xs.\ (\bigwedge x.\ x \in set\ xs \Longrightarrow P\ x) \Longrightarrow P\ (Parallel\ xs)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y]\!] \Longrightarrow P\ (NonD\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y]\!] \Longrightarrow P\ (Executable\ x\ y)$
and $\bigwedge x\ y.\ [\![P\ x;\ P\ y]\!] \Longrightarrow P\ (Repeatable\ x\ y)$
shows $P\ x$
⟨*proof*⟩

Representative terms of the lifted constructors apart from *Resource.Parallel* are known

lemma *of-resource-simps* [*simp*]:
of-resource Empty = *res-term.Empty*
of-resource Anything = *res-term.Anything*
of-resource (*Res a*) = *res-term.Res a*
of-resource (*Copyable b*) = *res-term.Copyable b*
of-resource (*NonD x y*) = *res-term.NonD* (*of-resource x*) (*of-resource y*)
of-resource (*Executable x y*) = *res-term.Executable* (*of-resource x*) (*of-resource y*)
of-resource (*Repeatable x y*) = *res-term.Repeatable* (*of-resource x*) (*of-resource y*)
⟨*proof*⟩

Basic resource term equivalences become resource equalities

lemma [*simp*]:
shows *resource-empty*: *Parallel* [] = *Empty*
and *resource-singleton*: *Parallel* [*x*] = *x*
and *resource-merge*: *Parallel* (*xs* @ [*Parallel ys*] @ *zs*) = *Parallel* (*xs* @ *ys* @ *zs*)
and *resource-drop*: *Parallel* (*xs* @ [*Empty*] @ *zs*) = *Parallel* (*xs* @ *zs*)
⟨*proof*⟩

lemma *resource-parallel-nested* [*simp*]:
Parallel (*Parallel xs* # *ys*) = *Parallel* (*xs* @ *ys*)
⟨*proof*⟩

lemma *resource-decompose*:
assumes *Parallel xs* = *Parallel ys*
and *Parallel us* = *Parallel vs*
shows *Parallel* (*xs* @ *us*) = *Parallel* (*ys* @ *vs*)
⟨*proof*⟩

lemma *resource-drop-list*:
$(\bigwedge y.\ y \in set\ ys \Longrightarrow y = Empty) \Longrightarrow Parallel\ (xs\ @\ ys\ @\ zs) = Parallel\ (xs\ @\ zs)$
⟨*proof*⟩

Equality of resources except *Resource.Parallel* implies equality of their children

lemma

shows *resource-res-eq*: $Res\ x = Res\ y \Longrightarrow x = y$
and *resource-copyable-eq*: $Copyable\ x = Copyable\ y \Longrightarrow x = y$
$\langle proof \rangle$

lemma *resource-nondet-eq*:
$NonD\ a\ b = NonD\ x\ y \Longrightarrow a = x$
$NonD\ a\ b = NonD\ x\ y \Longrightarrow b = y$
$\langle proof \rangle$

lemma *resource-executable-eq*:
$Executable\ a\ b = Executable\ x\ y \Longrightarrow a = x$
$Executable\ a\ b = Executable\ x\ y \Longrightarrow b = y$
$\langle proof \rangle$

lemma *resource-repeatable-eq*:
$Repeatable\ a\ b = Repeatable\ x\ y \Longrightarrow a = x$
$Repeatable\ a\ b = Repeatable\ x\ y \Longrightarrow b = y$
$\langle proof \rangle$

Many resource inequalities not involving *Resource.Parallel* are simple to prove

lemma *resource-neq* [*simp*]:
$Empty \neq Anything$
$Empty \neq Res\ a$
$Empty \neq Copyable\ b$
$Empty \neq NonD\ x\ y$
$Empty \neq Executable\ x\ y$
$Empty \neq Repeatable\ x\ y$
$Anything \neq Res\ a$
$Anything \neq Copyable\ b$
$Anything \neq NonD\ x\ y$
$Anything \neq Executable\ x\ y$
$Anything \neq Repeatable\ x\ y$
$Res\ a \neq Copyable\ b$
$Res\ a \neq NonD\ x\ y$
$Res\ a \neq Executable\ x\ y$
$Res\ a \neq Repeatable\ x\ y$
$Copyable\ b \neq NonD\ x\ y$
$Copyable\ b \neq Executable\ x\ y$
$Copyable\ b \neq Repeatable\ x\ y$
$NonD\ x\ y \neq Executable\ u\ v$
$NonD\ x\ y \neq Repeatable\ u\ v$
$Executable\ x\ y \neq Repeatable\ u\ v$
$\langle proof \rangle$

Resource map of lifted constructors can be simplified

lemma *map-resource-simps* [*simp*]:
$map\text{-}resource\ f\ g\ Empty = Empty$
$map\text{-}resource\ f\ g\ Anything = Anything$

map-resource f g (*Res a*) = *Res* (*f a*)
map-resource f g (*Copyable b*) = *Copyable* (*g b*)
map-resource f g (*Parallel xs*) = *Parallel* (*map* (*map-resource f g*) *xs*)
map-resource f g (*NonD x y*) = *NonD* (*map-resource f g x*) (*map-resource f g y*)
map-resource f g (*Executable x y*) = *Executable* (*map-resource f g x*) (*map-resource f g y*)
map-resource f g (*Repeatable x y*) = *Repeatable* (*map-resource f g x*) (*map-resource f g y*)
⟨*proof*⟩

Note that resource term size doesn't lift, because *res-term.Parallel* [*res-term.Empty*] is equivalent to *Resource.Empty* but their sizes are 2 and 1 respectively.

7.4 Parallel Product

We introduce infix syntax for binary *Resource.Parallel*, forming a resource product

definition *resource-par* :: ('*a*, '*b*) *resource* ⇒ ('*a*, '*b*) *resource* ⇒ ('*a*, '*b*) *resource*
(**infixr** ⊙ *120*)
where *x* ⊙ *y* = *Parallel* [*x*, *y*]

For the purposes of code generation we act as if we lifted it

lemma *resource-par-code* [*code*]:
abs-resource x ⊙ *abs-resource y* = *abs-resource* (*ResTerm.Parallel* [*x*, *y*])
⟨*proof*⟩

Parallel product can be merged with *Resource.Parallel* resources on either side or around it

lemma *resource-par-is-parallel* [*simp*]:
x ⊙ *Parallel xs* = *Parallel* (*x* # *xs*)
Parallel xs ⊙ *x* = *Parallel* (*xs* @ [*x*])
⟨*proof*⟩

lemma *resource-par-nested-start* [*simp*]:
Parallel (*x* ⊙ *y* # *zs*) = *Parallel* (*x* # *y* # *zs*)
⟨*proof*⟩

lemma *resource-par-nested* [*simp*]:
Parallel (*xs* @ *a* ⊙ *b* # *ys*) = *Parallel* (*xs* @ *a* # *b* # *ys*)
⟨*proof*⟩

Lifted constructor *Resource.Parallel*, which does not have automatic code equations, can be given code equations using this resource product

lemmas [*code*] = *resource-empty resource-par-is-parallel*(*1*)[*symmetric*]

This resource product sometimes leads to overly long expressions when generating code for formalised models, but these can be limited by code unfolding

lemma *resource-par-res* [*code-unfold*]:
$Res\ x \odot y = Parallel\ [Res\ x,\ y]$
⟨*proof*⟩
lemma *resource-parallel-res* [*code-unfold*]:
$Parallel\ [Res\ x,\ Parallel\ ys] = Parallel\ (Res\ x\ \#\ ys)$
⟨*proof*⟩

We show that this resource product is a monoid, meaning it is unital and associative

lemma *resource-par-unitL* [*simp*]:
$Empty \odot x = x$
⟨*proof*⟩

lemma *resource-par-unitR* [*simp*]:
$x \odot Empty = x$
⟨*proof*⟩

lemma *resource-par-assoc* [*simp*]:
$(a \odot b) \odot c = a \odot (b \odot c)$
⟨*proof*⟩

Resource map passes through resource product

lemma *resource-par-map* [*simp*]:
map-resource f g (*resource-par a b*) = *resource-par* (*map-resource f g a*) (*map-resource f g b*)
⟨*proof*⟩

Representative of resource product is normalised *res-term.Parallel* term of the two children's representations

lemma *of-resource-par*:
of-resource (*resource-par x y*) = *normal-rewr* (*res-term.Parallel* [*of-resource x*, *of-resource y*])
⟨*proof*⟩

7.5 Lifting Parallel Parts

lift-definition *parallel-parts* :: ('*a*, '*b*) *resource* ⇒ ('*a*, '*b*) *resource list*
is *ResTerm.parallel-parts* ⟨*proof*⟩

Parallel parts of the lifted constructors can be simplified like the term version

lemma *parallel-parts-simps*:
parallel-parts Empty = []
parallel-parts Anything = [*Anything*]
parallel-parts (*Res a*) = [*Res a*]
parallel-parts (*Copyable b*) = [*Copyable b*]
parallel-parts (*Parallel xs*) = *concat* (*map parallel-parts xs*)
parallel-parts (*NonD x y*) = [*NonD x y*]
parallel-parts (*Executable x y*) = [*Executable x y*]

parallel-parts (*Repeatable x y*) = [*Repeatable x y*]
⟨*proof*⟩

Every resource is the same as *Resource.Parallel* resource formed from its parallel parts

lemma *resource-eq-parallel-parts*:
x = *Parallel* (*parallel-parts x*)
⟨*proof*⟩

Resources with equal parallel parts are equal

lemma *parallel-parts-cong*:
parallel-parts x = *parallel-parts y* $\Longrightarrow$ *x* = *y*
⟨*proof*⟩

Parallel parts of the resource product are the two resources' parallel parts

lemma *parallel-parts-par*:
parallel-parts (*a* $\odot$ *b*) = *parallel-parts a* @ *parallel-parts b*
⟨*proof*⟩

7.6 Lifting Parallelisation

lift-definition *parallelise* :: ('*a*, '*b*) *resource list* $\Rightarrow$ ('*a*, '*b*) *resource*
is *ResTerm.parallelise*
⟨*proof*⟩

Parallelisation of the lifted constructors can be simplified like the term version

lemma *parallelise-resource-simps* [*code*]:
parallelise [] = *Empty*
parallelise [*x*] = *x*
parallelise (*x*#*y*#*zs*) = *Parallel* (*x*#*y*#*zs*)
⟨*proof*⟩

7.7 Representative of Parallel Resource

By relating to direct normalisation, representative term for *Resource.Parallel* is parallelisation of representatives of its parallel parts

lemma *of-resource-parallel*:
of-resource (*Parallel xs*)
= *ResTerm.parallelise* (*merge-all-parallel* (*remove-all-empty* (*map of-resource xs*)))
⟨*proof*⟩

Equality of *Resource.Parallel* resources implies equality of their parallel parts

lemma *resource-parallel-eq*:

Parallel xs = Parallel ys $\Longrightarrow$ *concat* (*map parallel-parts xs*) = *concat* (*map parallel-parts ys*)
⟨*proof*⟩

With this, we can prove simplification equations for atom sets

lemma *set1-resource-simps* [*simp*]:
set1-resource Empty = {}
set1-resource Anything = {}
set1-resource (*Res a*) = {*a*}
set1-resource (*Copyable b*) = {}
set1-resource (*Parallel xs*) = $\bigcup$(*set1-resource* ‘ *set xs*)
set1-resource (*NonD x y*) = *set1-resource x* ∪ *set1-resource y*
set1-resource (*Executable x y*) = *set1-resource x* ∪ *set1-resource y*
set1-resource (*Repeatable x y*) = *set1-resource x* ∪ *set1-resource y*
⟨*proof*⟩
lemma *set2-resource-simps* [*simp*]:
set2-resource Empty = {}
set2-resource Anything = {}
set2-resource (*Res a*) = {}
set2-resource (*Copyable b*) = {*b*}
set2-resource (*Parallel xs*) = $\bigcup$(*set2-resource* ‘ *set xs*)
set2-resource (*NonD x y*) = *set2-resource x* ∪ *set2-resource y*
set2-resource (*Executable x y*) = *set2-resource x* ∪ *set2-resource y*
set2-resource (*Repeatable x y*) = *set2-resource x* ∪ *set2-resource y*
⟨*proof*⟩

7.8 Replicated Resources

Replicate a resource several times in a *Resource.Parallel*

fun *nres-term* :: *nat* ⇒ (*'a*, *'b*) *res-term* ⇒ (*'a*, *'b*) *res-term*
where *nres-term n x* = *ResTerm.Parallel* (*replicate n x*)

lift-definition *nresource* :: *nat* ⇒ (*'a*, *'b*) *resource* ⇒ (*'a*, *'b*) *resource*
is *nres-term* ⟨*proof*⟩

At the resource level this can be simplified just like at the term level

lemma *nresource-simp*:
nresource n x = *Parallel* (*replicate n x*)
⟨*proof*⟩

Parallel product of replications is a replication for the combined amount

lemma *nresource-par*:
nresource x a ⊙ *nresource y a* = *nresource* (*x*+*y*) *a*
⟨*proof*⟩

7.9 Lifting Resource Refinement

lift-definition *refine-resource*

$:: ('a \Rightarrow ('x, 'y)\ resource) \Rightarrow ('b \Rightarrow 'y) \Rightarrow ('a, 'b)\ resource \Rightarrow ('x, 'y)\ resource$
is *refine-res-term* ⟨*proof*⟩

Refinement of lifted constructors can be simplified like the term version

lemma *refine-resource-simps* [*simp*]:
refine-resource f g Empty = Empty
refine-resource f g Anything = Anything
refine-resource f g (*Res a*) = *f a*
refine-resource f g (*Copyable b*) = *Copyable* (*g b*)
refine-resource f g (*Parallel xs*) = *Parallel* (*map* (*refine-resource f g*) *xs*)
refine-resource f g (*NonD x y*) = *NonD* (*refine-resource f g x*) (*refine-resource f g y*)
refine-resource f g (*Executable x y*) =
Executable (*refine-resource f g x*) (*refine-resource f g y*)
refine-resource f g (*Repeatable x y*) =
Repeatable (*refine-resource f g x*) (*refine-resource f g y*)
⟨*proof*⟩

Code for refinement performs the term-level refinement on the normalised representative

lemma *refine-resource-code* [*code*]:
refine-resource f g (*abs-resource x*) = *abs-resource* (*refine-res-term* (*of-resource* ∘ *f*) *g x*)
⟨*proof*⟩

Refinement passes through resource product

lemma *refine-resource-par*:
refine-resource f g (*x* ⊙ *y*) = *refine-resource f g x* ⊙ *refine-resource f g y*
⟨*proof*⟩

end
theory *Process*
imports *Resource*
begin

8 Process Compositions

We define process compositions to describe how larger processes are built from smaller ones from the perspective of how outputs of some actions serve as inputs for later actions. Our process compositions form a tree, with actions as leaves and composition operations as internal nodes. We use resources to represent the inputs and outputs of processes.

8.1 Datatype, Input, Output and Validity

Process composition datatype with primitive actions, composition operations and resource actions. We use the following type variables:

- $'a$ for linear resource atoms,
- $'b$ for copyable resource atoms,
- $'l$ for primitive action labels, and
- $'m$ for primitive action metadata.

datatype $('a, 'b, 'l, 'm)$ *process* =
  Primitive $('a, 'b)$ *resource* $('a, 'b)$ *resource* $'l$ $'m$
  — Primitive action with given input, ouptut, label and metadata
| *Seq* $('a, 'b, 'l, 'm)$ *process* $('a, 'b, 'l, 'm)$ *process*
  — Sequential composition
| *Par* $('a, 'b, 'l, 'm)$ *process* $('a, 'b, 'l, 'm)$ *process*
  — Parallel composition
| *Opt* $('a, 'b, 'l, 'm)$ *process* $('a, 'b, 'l, 'm)$ *process*
  — Optional composition
| *Represent* $('a, 'b, 'l, 'm)$ *process*
  — Representation of a process composition as a repeatably exectuable resource
| *Identity* $('a, 'b)$ *resource*
  — Identity action
| *Swap* $('a, 'b)$ *resource* $('a, 'b)$ *resource*
  — Swap action
| *InjectL* $('a, 'b)$ *resource* $('a, 'b)$ *resource*
  — Left injection
| *InjectR* $('a, 'b)$ *resource* $('a, 'b)$ *resource*
  — Right injection
| *OptDistrIn* $('a, 'b)$ *resource* $('a, 'b)$ *resource* $('a, 'b)$ *resource*
  — Distribution into branches of a non-deterministic resource
| *OptDistrOut* $('a, 'b)$ *resource* $('a, 'b)$ *resource* $('a, 'b)$ *resource*
  — Distribution out of branches of a non-deterministic resource
| *Duplicate* $'b$
  — Duplication of a copyable resource
| *Erase* $'b$
  — Discarding a copyable resource
| *Apply* $('a, 'b)$ *resource* $('a, 'b)$ *resource*
  — Applying an executable resource
| *Repeat* $('a, 'b)$ *resource* $('a, 'b)$ *resource*
  — Duplicating a repeatably executable resource
| *Close* $('a, 'b)$ *resource* $('a, 'b)$ *resource*
  — Discarding a repeatably executable resource
| *Once* $('a, 'b)$ *resource* $('a, 'b)$ *resource*
  — Converting a repeatably executable resource into a plain execuable resource
| *Forget* $('a, 'b)$ *resource*
  — Forgetting all details about a resource

Each process composition has a well defined input and output resource, derived recursively from the individual actions that constitute it.

primrec *input* :: $('a, 'b, 'l, 'm)$ *process* $\Rightarrow$ $('a, 'b)$ *resource*

where
  input (*Primitive ins outs l m*) = *ins*
| *input* (*Seq p q*) = *input p*
| *input* (*Par p q*) = *input p* ⊙ *input q*
| *input* (*Opt p q*) = *NonD* (*input p*) (*input q*)
| *input* (*Represent p*) = *Empty*
| *input* (*Identity a*) = *a*
| *input* (*Swap a b*) = *a* ⊙ *b*
| *input* (*InjectL a b*) = *a*
| *input* (*InjectR a b*) = *b*
| *input* (*OptDistrIn a b c*) = *a* ⊙ (*NonD b c*)
| *input* (*OptDistrOut a b c*) = *NonD* (*a* ⊙ *b*) (*a* ⊙ *c*)
| *input* (*Duplicate a*) = *Copyable a*
| *input* (*Erase a*) = *Copyable a*
| *input* (*Apply a b*) = *a* ⊙ (*Executable a b*)
| *input* (*Repeat a b*) = *Repeatable a b*
| *input* (*Close a b*) = *Repeatable a b*
| *input* (*Once a b*) = *Repeatable a b*
| *input* (*Forget a*) = *a*

Input of mapped process is accordingly mapped input

lemma *map-process-input* [*simp*]:
input (*map-process f g h i x*) = *map-resource f g* (*input x*)
⟨*proof*⟩

primrec *output* :: (′*a*, ′*b*, ′*l*, ′*m*) *process* ⇒ (′*a*, ′*b*) *resource*
where
  output (*Primitive ins outs l m*) = *outs*
| *output* (*Seq p q*) = *output q*
| *output* (*Par p q*) = *output p* ⊙ *output q*
| *output* (*Opt p q*) = *output p*
| *output* (*Represent p*) = *Repeatable* (*input p*) (*output p*)
| *output* (*Identity a*) = *a*
| *output* (*Swap a b*) = *b* ⊙ *a*
| *output* (*InjectL a b*) = *NonD a b*
| *output* (*InjectR a b*) = *NonD a b*
| *output* (*OptDistrIn a b c*) = *NonD* (*a* ⊙ *b*) (*a* ⊙ *c*)
| *output* (*OptDistrOut a b c*) = *a* ⊙ (*NonD b c*)
| *output* (*Duplicate a*) = *Copyable a* ⊙ *Copyable a*
| *output* (*Erase a*) = *Empty*
| *output* (*Apply a b*) = *b*
| *output* (*Repeat a b*) = (*Repeatable a b*) ⊙ (*Repeatable a b*)
| *output* (*Close a b*) = *Empty*
| *output* (*Once a b*) = *Executable a b*
| *output* (*Forget a*) = *Anything*

Output of mapped process is accordingly mapped output

lemma *map-process-output* [*simp*]:
output (*map-process f g h i x*) = *map-resource f g* (*output x*)

⟨*proof*⟩

Not all process compositions are valid. While we consider all individual actions to be valid, we impose two conditions on composition operations beyond the validity of their children:

- Sequential composition requires that the output of the first process be the input of the second.
- Optional composition requires that the two processes arrive at the same output.

primrec *valid* :: $('a, 'b, 'l, 'm)$ *process* $\Rightarrow$ *bool*
where
valid (*Primitive ins outs l m*) = *True*
| *valid* (*Seq p q*) = (*output p* = *input q* $\wedge$ *valid p* $\wedge$ *valid q*)
| *valid* (*Par p q*) = (*valid p* $\wedge$ *valid q*)
| *valid* (*Opt p q*) = (*valid p* $\wedge$ *valid q* $\wedge$ *output p* = *output q*)
| *valid* (*Represent p*) = *valid p*
| *valid* (*Identity a*) = *True*
| *valid* (*Swap a b*) = *True*
| *valid* (*InjectL a b*) = *True*
| *valid* (*InjectR a b*) = *True*
| *valid* (*OptDistrIn a b c*) = *True*
| *valid* (*OptDistrOut a b c*) = *True*
| *valid* (*Duplicate a*) = *True*
| *valid* (*Erase a*) = *True*
| *valid* (*Apply a b*) = *True*
| *valid* (*Repeat a b*) = *True*
| *valid* (*Close a b*) = *True*
| *valid* (*Once a b*) = *True*
| *valid* (*Forget a*) = *True*

Process mapping preserves validity

lemma *map-process-valid* [*simp*]:
valid x $\Longrightarrow$ *valid* (*map-process f g h i x*)
⟨*proof*⟩

However, it does not necessarily preserve invalidity if there exist two distinct linear or copyable resource atoms

lemma
fixes *g h i* **and** *a b* :: *'a*
assumes $a \neq b$
obtains *f* **and** *x* :: $('a, 'b, 'l, 'm)$ *process*
where $\neg$ *valid x* **and** *valid* (*map-process f g h i x*)
⟨*proof*⟩
lemma
fixes *f h i* **and** *a b* :: *'b*

assumes $a \neq b$
obtains g **and** x :: $('a, 'b, 'l, 'm)$ *process*
where $\neg$ *valid* x **and** *valid* (*map-process* $f\ g\ h\ i\ x$)
⟨*proof*⟩

If the resource map is injective then mapping with it does not change validity

lemma *map-process-valid-eq*:
assumes *inj* f
and *inj* g
shows *valid* x = *valid* (*map-process* $f\ g\ h\ i\ x$)
⟨*proof*⟩

8.2 Gathering Primitive Actions

As primitive actions represent assumptions about what we can do in the modelling domain, it is often useful to gather them.

When we want to talk about only primitive actions, we represent them with a quadruple of input, output, label and metadata, just as the parameters to the *Primitive* constructor.

type-synonym $('a, 'b, 'l, 'm)$ *prim-pars* = $('a, 'b)$ *resource* $\times$ $('a, 'b)$ *resource* $\times$ $'l \times 'm$

Uncurried version of *Primitive* to use with *prim-pars*

fun *Primitive-unc* :: $('a, 'b, 'l, 'm)$ *prim-pars* $\Rightarrow$ $('a, 'b, 'l, 'm)$ *process*
where *Primitive-unc* (a, b, l, m) = *Primitive* $a\ b\ l\ m$

Gather the primitives recursively from the composition, preserving their order

primrec *primitives* :: $('a, 'b, 'l, 'm)$ *process* $\Rightarrow$ $('a, 'b, 'l, 'm)$ *prim-pars list*
where
primitives (*Primitive ins outs* $l\ m$) = [(*ins*, *outs*, l, m)]
| *primitives* (*Seq* $p\ q$) = *primitives* p @ *primitives* q
| *primitives* (*Par* $p\ q$) = *primitives* p @ *primitives* q
| *primitives* (*Opt* $p\ q$) = *primitives* p @ *primitives* q
| *primitives* (*Represent* p) = *primitives* p
| *primitives* (*Identity* a) = []
| *primitives* (*Swap* $a\ b$) = []
| *primitives* (*InjectL* $a\ b$) = []
| *primitives* (*InjectR* $a\ b$) = []
| *primitives* (*OptDistrIn* $a\ b\ c$) = []
| *primitives* (*OptDistrOut* $a\ b\ c$) = []
| *primitives* (*Duplicate* a) = []
| *primitives* (*Erase* a) = []
| *primitives* (*Apply* $a\ b$) = []
| *primitives* (*Repeat* $a\ b$) = []
| *primitives* (*Close* $a\ b$) = []
| *primitives* (*Once* $a\ b$) = []

| *primitives* (*Forget a*) = []

Primitives of mapped process are accordingly mapped primitives

lemma *map-process-primitives* [*simp*]:
primitives (*map-process f g h i x*)
= *map* (λ(*a*, *b*, *l*, *m*). (*map-resource f g a*, *map-resource f g b*, *h l*, *i m*)) (*primitives x*)
⟨*proof*⟩

8.3 Resource Refinement in Processes

We can apply *refine-resource* systematically throughout a process composition

primrec *process-refineRes* ::
('*a* ⇒ ('*x*, '*y*) *resource*) ⇒ ('*b* ⇒ '*y*) ⇒ ('*a*, '*b*, '*l*, '*m*) *process* ⇒ ('*x*, '*y*, '*l*, '*m*) *process*
where
process-refineRes f g (*Primitive ins outs l m*) =
Primitive (*refine-resource f g ins*) (*refine-resource f g outs*) *l m*
| *process-refineRes f g* (*Identity a*) = *Identity* (*refine-resource f g a*)
| *process-refineRes f g* (*Swap a b*) = *Swap* (*refine-resource f g a*) (*refine-resource f g b*)
| *process-refineRes f g* (*Seq p q*) = *Seq* (*process-refineRes f g p*) (*process-refineRes f g q*)
| *process-refineRes f g* (*Par p q*) = *Par* (*process-refineRes f g p*) (*process-refineRes f g q*)
| *process-refineRes f g* (*Opt p q*) = *Opt* (*process-refineRes f g p*) (*process-refineRes f g q*)
| *process-refineRes f g* (*InjectL a b*) = *InjectL* (*refine-resource f g a*) (*refine-resource f g b*)
| *process-refineRes f g* (*InjectR a b*) = *InjectR* (*refine-resource f g a*) (*refine-resource f g b*)
| *process-refineRes f g* (*OptDistrIn a b c*) =
OptDistrIn (*refine-resource f g a*) (*refine-resource f g b*) (*refine-resource f g c*)
| *process-refineRes f g* (*OptDistrOut a b c*) =
OptDistrOut (*refine-resource f g a*) (*refine-resource f g b*) (*refine-resource f g c*)
| *process-refineRes f g* (*Duplicate a*) = *Duplicate* (*g a*)
| *process-refineRes f g* (*Erase a*) = *Erase* (*g a*)
| *process-refineRes f g* (*Represent p*) = *Represent* (*process-refineRes f g p*)
| *process-refineRes f g* (*Apply a b*) = *Apply* (*refine-resource f g a*) (*refine-resource f g b*)
| *process-refineRes f g* (*Repeat a b*) = *Repeat* (*refine-resource f g a*) (*refine-resource f g b*)
| *process-refineRes f g* (*Close a b*) = *Close* (*refine-resource f g a*) (*refine-resource f g b*)
| *process-refineRes f g* (*Once a b*) = *Once* (*refine-resource f g a*) (*refine-resource f g b*)

| *process-refineRes f g* (*Forget a*) = *Forget* (*refine-resource f g a*)

This behaves well with the input, output and primitives, and preserves validity

lemma *process-refineRes-input* [*simp*]:
input (*process-refineRes f g x*) = *refine-resource f g* (*input x*)
⟨*proof*⟩
lemma *process-refineRes-output* [*simp*]:
output (*process-refineRes f g x*) = *refine-resource f g* (*output x*)
⟨*proof*⟩
lemma *process-refineRes-primitives*:
primitives (*process-refineRes f g x*)
= *map* (λ(*ins*, *outs*, *l*, *m*). (*refine-resource f g ins*, *refine-resource f g outs*, *l*, *m*))
(*primitives x*)
⟨*proof*⟩
lemma *process-refineRes-valid* [*simp*]:
valid x ⟹ *valid* (*process-refineRes f g x*)
⟨*proof*⟩

9 List-based Composition Actions

We define functions to compose a list of processes in sequence or in parallel. In both cases these associate the binary operation to the right, and for the empty list they both use the identity process on the *Resource.Empty* resource.

Compose a list of processes in sequence

primrec *seq-process-list* :: (′*a*, ′*b*, ′*l*, ′*m*) *process list* ⇒ (′*a*, ′*b*, ′*l*, ′*m*) *process*
where
seq-process-list [] = *Identity Empty*
| *seq-process-list* (*x* # *xs*) = (*if xs* = [] *then x else Seq x* (*seq-process-list xs*))

lemma *seq-process-list-input* [*simp*]:
xs ≠ [] ⟹ *input* (*seq-process-list xs*) = *input* (*hd xs*)
⟨*proof*⟩

lemma *seq-process-list-output* [*simp*]:
xs ≠ [] ⟹ *output* (*seq-process-list xs*) = *output* (*last xs*)
⟨*proof*⟩

lemma *seq-process-list-valid*:
valid (*seq-process-list xs*)
= (*list-all valid xs*
∧ (∀ *i* :: *nat*. *i* < *length xs* − *1* ⟶ *output* (*xs* ! *i*) = *input* (*xs* ! *Suc i*)))
⟨*proof*⟩

lemma *seq-process-list-primitives* [*simp*]:
primitives (*seq-process-list xs*) = *concat* (*map primitives xs*)

⟨*proof*⟩

We use list-based sequential composition to make generated code more readable

lemma *seq-process-list-code-unfold* [*code-unfold*]:
Seq x (*Seq y z*) = *seq-process-list* [*x*, *y*, *z*]
Seq x (*seq-process-list* (*y* # *ys*)) = *seq-process-list* (*x* # *y* # *ys*)
⟨*proof*⟩

Resource refinement can be distributed across the list being composed

lemma *seq-process-list-refine*:
process-refineRes f g (*seq-process-list xs*) = *seq-process-list* (*map* (*process-refineRes f g*) *xs*)
⟨*proof*⟩

Compose a list of processes in parallel

primrec *par-process-list* :: (*'a*, *'b*, *'l*, *'m*) *process list* ⇒ (*'a*, *'b*, *'l*, *'m*) *process*
where
par-process-list [] = *Identity Empty*
| *par-process-list* (*x* # *xs*) = (*if xs* = [] *then x else Par x* (*par-process-list xs*))

lemma *par-process-list-input* [*simp*]:
input (*par-process-list xs*) = *foldr* (⊙) (*map input xs*) *Empty*
⟨*proof*⟩

lemma *par-process-list-output* [*simp*]:
output (*par-process-list xs*) = *foldr* (⊙) (*map output xs*) *Empty*
⟨*proof*⟩

lemma *par-process-list-valid* [*simp*]:
valid (*par-process-list xs*) = *list-all valid xs*
⟨*proof*⟩

lemma *par-process-list-primitives* [*simp*]:
primitives (*par-process-list xs*) = *concat* (*map primitives xs*)
⟨*proof*⟩

We use list-based parallel composition to make generated code more readable

lemma *par-process-list-code-unfold* [*code-unfold*]:
Par x (*Par y z*) = *par-process-list* [*x*, *y*, *z*]
Par x (*par-process-list* (*y* # *ys*)) = *par-process-list* (*x* # *y* # *ys*)
⟨*proof*⟩

Resource refinement can be distributed across the list being composed

lemma *par-process-list-refine*:
process-refineRes f g (*par-process-list xs*) = *par-process-list* (*map* (*process-refineRes f g*) *xs*)
⟨*proof*⟩

9.1 Progressing Both Non-deterministic Branches

Note that validity of *Opt* requires that its children have equal outputs. However, we can define a composition template that allows us to optionally compose processes with different outputs, producing the non-deterministic combination of those outputs. This represents progressing both branches of a *Resource.NonD* resource without merging them.

fun *OptProgress* :: ('a, 'b, 'l, 'm) *process* ⇒ ('a, 'b, 'l, 'm) *process* ⇒ ('a, 'b, 'l, 'm) *process*
  where *OptProgress p q* =
  Opt (*Seq p* (*InjectL* (*output p*) (*output q*)))
    (*Seq q* (*InjectR* (*output p*) (*output q*)))

The result takes the non-deterministic combination of the children's inputs and produces the non-deterministic combination of their outputs, and it is valid whenever the two children are valid.

lemma [*simp*]:
  shows *OptProgress-input*: *input* (*OptProgress x y*) = *NonD* (*input x*) (*input y*)
    and *OptProgress-output*: *output* (*OptProgress x y*) = *NonD* (*output x*) (*output y*)
    and *OptProgress-valid*: *valid* (*OptProgress x y*) = (*valid x* ∧ *valid y*)
  ⟨*proof*⟩

10 Primitive Action Substitution

We define a function to substitute primitive actions within any process composition. The target actions are specified through a predicate on their parameters. The replacement composition is then a function of those primitives.

primrec *process-subst* ::
  (('a, 'b) *resource* ⇒ ('a, 'b) *resource* ⇒ 'l ⇒ 'm ⇒ *bool*) ⇒
   (('a, 'b) *resource* ⇒ ('a, 'b) *resource* ⇒ 'l ⇒ 'm ⇒ ('a, 'b, 'l, 'm) *process*) ⇒
   ('a, 'b, 'l, 'm) *process* ⇒ ('a, 'b, 'l, 'm) *process*
  where
   process-subst P f (*Primitive a b l m*) = (*if P a b l m then f a b l m else Primitive a b l m*)
  | *process-subst P f* (*Identity a*) = *Identity a*
  | *process-subst P f* (*Swap a b*) = *Swap a b*
  | *process-subst P f* (*Seq p q*) = *Seq* (*process-subst P f p*) (*process-subst P f q*)
  | *process-subst P f* (*Par p q*) = *Par* (*process-subst P f p*) (*process-subst P f q*)
  | *process-subst P f* (*Opt p q*) = *Opt* (*process-subst P f p*) (*process-subst P f q*)
  | *process-subst P f* (*InjectL a b*) = *InjectL a b*
  | *process-subst P f* (*InjectR a b*) = *InjectR a b*
  | *process-subst P f* (*OptDistrIn a b c*) = *OptDistrIn a b c*
  | *process-subst P f* (*OptDistrOut a b c*) = *OptDistrOut a b c*
  | *process-subst P f* (*Duplicate a*) = *Duplicate a*

| *process-subst* $P\ f$ (*Erase* a) = *Erase* a
| *process-subst* $P\ f$ (*Represent* p) = *Represent* (*process-subst* $P\ f\ p$)
| *process-subst* $P\ f$ (*Apply* $a\ b$) = *Apply* $a\ b$
| *process-subst* $P\ f$ (*Repeat* $a\ b$) = *Repeat* $a\ b$
| *process-subst* $P\ f$ (*Close* $a\ b$) = *Close* $a\ b$
| *process-subst* $P\ f$ (*Once* $a\ b$) = *Once* $a\ b$
| *process-subst* $P\ f$ (*Forget* a) = *Forget* a

If no matching target primitive is present, then the substitution does nothing

lemma *process-subst-no-target*:
($\bigwedge a\ b\ l\ m.\ (a, b, l, m) \in$ *set* (*primitives* x) $\Longrightarrow \neg\ P\ a\ b\ l\ m$) $\Longrightarrow$ *process-subst* $P\ f\ x = x$
⟨*proof*⟩

If a process has no primitives, then any substitution does nothing on it

lemma *process-subst-no-prims*:
primitives $x = [] \Longrightarrow$ *process-subst* $P\ f\ x = x$
⟨*proof*⟩

If the replacement process does not change the inputs, then input is preserved through the substitution

lemma *process-subst-input* [*simp*]:
($\bigwedge a\ b\ l\ m.\ P\ a\ b\ l\ m \Longrightarrow$ *input* ($f\ a\ b\ l\ m$) $= a$) $\Longrightarrow$ *input* (*process-subst* $P\ f\ x$) = *input* x
⟨*proof*⟩

If the replacement additionally does not change the outputs, then the output is also preserved through the substitution

lemma *process-subst-output* [*simp*]:
assumes $\bigwedge a\ b\ l\ m.\ P\ a\ b\ l\ m \Longrightarrow$ *input* ($f\ a\ b\ l\ m$) $= a$
and $\bigwedge a\ b\ l\ m.\ P\ a\ b\ l\ m \Longrightarrow$ *output* ($f\ a\ b\ l\ m$) $= b$
shows *output* (*process-subst* $P\ f\ x$) = *output* x
⟨*proof*⟩

If the replacement is additionally valid for every target, then validity is preserved through the substitution

lemma *process-subst-valid* [*simp*]:
assumes $\bigwedge a\ b\ l\ m.\ P\ a\ b\ l\ m \Longrightarrow$ *input* ($f\ a\ b\ l\ m$) $= a$
and $\bigwedge a\ b\ l\ m.\ P\ a\ b\ l\ m \Longrightarrow$ *output* ($f\ a\ b\ l\ m$) $= b$
and $\bigwedge a\ b\ l\ m.\ P\ a\ b\ l\ m \Longrightarrow$ *valid* ($f\ a\ b\ l\ m$)
shows *valid* (*process-subst* $P\ f\ x$) = *valid* x
⟨*proof*⟩

Primitives after substitution are those that didn't satisfy the predicate and anything that was introduced by the function applied on satisfying primitives' parameters.

lemma *process-subst-primitives*:

$primitives\ (process\text{-}subst\ P\ f\ x)$
$= concat\ (map$
$(\lambda(a,\ b,\ l,\ m).\ if\ P\ a\ b\ l\ m\ then\ primitives\ (f\ a\ b\ l\ m)\ else\ [(a,\ b,\ l,\ m)])$
$(primitives\ x))$
$\langle proof \rangle$

After substitution, no target action is left unless some replacement introduces one

lemma *process-subst-targets-removed*:
assumes $\bigwedge a\ b\ l\ m\ a'\ b'\ l'\ m'.$
$[\![(a,\ b,\ l,\ m) \in set\ (primitives\ x);\ P\ a\ b\ l\ m;\ (a',\ b',\ l',\ m') \in set\ (primitives\ (f\ a\ b\ l\ m))]\!]$
$\Longrightarrow \neg\ P\ a'\ b'\ l'\ m'$
— For any target primitive of the process, no primitive in its replacement is also a target
and $(a,\ b,\ l,\ m) \in set\ (primitives\ (process\text{-}subst\ P\ f\ x))$
shows $\neg\ P\ a\ b\ l\ m$
$\langle proof \rangle$

Process substitution distributes over list-based sequential and parallel composition

lemma *par-process-list-subst*:
$process\text{-}subst\ P\ f\ (par\text{-}process\text{-}list\ xs) = par\text{-}process\text{-}list\ (map\ (process\text{-}subst\ P\ f)\ xs)$
$\langle proof \rangle$

lemma *seq-process-list-subst*:
$process\text{-}subst\ P\ f\ (seq\text{-}process\text{-}list\ xs) = seq\text{-}process\text{-}list\ (map\ (process\text{-}subst\ P\ f)\ xs)$
$\langle proof \rangle$

11 Useful Notation

We set up notation to easily express the input and output of a process. We use two bundle: including one introduces the notation, while including the other removes it.

abbreviation $spec :: ('a,\ 'b,\ 'l,\ 'm)\ process \Rightarrow ('a,\ 'b)\ resource \Rightarrow ('a,\ 'b)\ resource \Rightarrow bool$
where $spec\ P\ a\ b \equiv input\ P = a \wedge output\ P = b$

bundle *spec-notation*
begin
notation *spec* ((-): (-) → (-) [*1000*, *60*] *60*)
end

bundle *spec-notation-undo*
begin

no-notation *spec* ((-): (-) → (-) [*1000*, *60*] *60*)
end

Set up notation bundles to be imported in a controlled way, along with inverses to undo them

We also set up infix notation for sequential and parallel process composition. Once again, we use two bundles to add and remove this notation. In this case that is even more useful, as out parallel composition notation overrides that of (‖).

bundle *process-notation*
begin
no-notation *Shuffle* (**infixr** ‖ *80*)
notation *Seq* (**infixr** ;; *55*)
notation *Par* (**infixr** ‖ *65*)
end

bundle *process-notation-undo*
begin
notation *Shuffle* (**infixr** ‖ *80*)
no-notation *Seq* (**infixr** ;; *55*)
no-notation *Par* (**infixr** ‖ *65*)
end

end
theory *CopyableElimination*
  imports *Process*
begin

12 Copyable Resource Elimination

We can show that copyable resources are not strictly necessary for the theory, being instead a convenience feature, by taking any valid process and transforming it into one that does not use any copyable resources. The cost is that we introduce new primitive actions, which represent the explicit assumptions that the resources that were copyable have actions that correspond to *Duplicate* and *Erase* in the domain. While an equivalent assumption (that such actions exist in the domain) is made by making an atom copyable instead of linear, that avenue fixes the form of those actions and as such lessens the risk of error in manually introducing them for this frequent pattern.
The concrete transformation takes a process of type $('a, 'b, 'l, 'm)$ *process* to one of type $('a + 'b, 'c, 'l + \mathit{String.literal}, 'm + \mathit{unit})$ *process*. Note the following:

- The two resource atom types are combined into one to form the new linear atoms.

- The new copyable atoms can be of any type, because the result makes no use of them.
- The old labels are combined with string literals to add label simple labels for the new actions.
- The old metadata is combined with *unit*, allowing the new actions to have no metadata.

12.1 Replacing Copyable Resource Actions

To remove the copyable resource actions *Duplicate* and *Erase* we replace them with *Primitive* actions with the corresponding input and output, string labels and no metadata.

```
primrec makeDuplEraToPrim
  :: ('a, 'b, 'l, 'm) process ⇒ ('a, 'b, 'l + String.literal, 'm + unit) process
  where
    makeDuplEraToPrim (Primitive a b l m) = Primitive a b (Inl l) (Inl m)
  | makeDuplEraToPrim (Identity a) = Identity a
  | makeDuplEraToPrim (Swap a b) = Swap a b
 | makeDuplEraToPrim (Seq p q) = Seq (makeDuplEraToPrim p) (makeDuplEraToPrim
q)
 | makeDuplEraToPrim (Par p q) = Par (makeDuplEraToPrim p) (makeDuplEraToPrim
q)
 | makeDuplEraToPrim (Opt p q) = Opt (makeDuplEraToPrim p) (makeDuplEraToPrim
q)
  | makeDuplEraToPrim (InjectL a b) = InjectL a b
  | makeDuplEraToPrim (InjectR a b) = InjectR a b
  | makeDuplEraToPrim (OptDistrIn a b c) = OptDistrIn a b c
  | makeDuplEraToPrim (OptDistrOut a b c) = OptDistrOut a b c
  | makeDuplEraToPrim (Duplicate a) =
      Primitive (Copyable a) (Copyable a ⊙ Copyable a) (Inr STR ''Duplicate'')
(Inr ())
  | makeDuplEraToPrim (Erase a) =
     Primitive (Copyable a) Empty (Inr STR ''Erase'') (Inr ())
  | makeDuplEraToPrim (Represent p) = Represent (makeDuplEraToPrim p)
  | makeDuplEraToPrim (Apply a b) = Apply a b
  | makeDuplEraToPrim (Repeat a b) = Repeat a b
  | makeDuplEraToPrim (Close a b) = Close a b
  | makeDuplEraToPrim (Once a b) = Once a b
  | makeDuplEraToPrim (Forget a) = Forget a
```

12.2 Making Copyable Resource Terms Linear

To eventually replace copyable resources, we first define how resource terms are replaced. Linear atoms are injected into the left side of the sum while copyable ones are injected into the right side, but both are turned into linear atoms in the result.

primrec *copyableToRes-term* :: $('a, 'b)$ *res-term* $\Rightarrow$ $('a + 'b, 'c)$ *res-term*
where
copyableToRes-term res-term.Empty = *res-term.Empty*
| *copyableToRes-term res-term.Anything* = *res-term.Anything*
| *copyableToRes-term* (*res-term.Res a*) = *res-term.Res* (*Inl a*)
| *copyableToRes-term* (*res-term.Copyable a*) = *res-term.Res* (*Inr a*)
| *copyableToRes-term* (*res-term.Parallel xs*) =
res-term.Parallel (*map copyableToRes-term xs*)
| *copyableToRes-term* (*res-term.NonD a b*) =
res-term.NonD (*copyableToRes-term a*) (*copyableToRes-term b*)
| *copyableToRes-term* (*res-term.Executable a b*) =
res-term.Executable (*copyableToRes-term a*) (*copyableToRes-term b*)
| *copyableToRes-term* (*res-term.Repeatable a b*) =
res-term.Repeatable (*copyableToRes-term a*) (*copyableToRes-term b*)

Replacing copyable resource terms preserves term equivalence

lemma *copyableToRes-term-equiv*:
$x \sim y \Longrightarrow$ *copyableToRes-term x* $\sim$ *copyableToRes-term y*
⟨*proof*⟩

Replacing copyable resource terms does not affect the nature of non-atoms

lemma *copyableToRes-term-is-Empty* [*simp*]:
is-Empty (*copyableToRes-term x*) = *is-Empty x*
⟨*proof*⟩
lemma *copyableToRes-term-has-Empty* [*simp*]:
list-ex is-Empty (*map copyableToRes-term xs*) = *list-ex is-Empty xs*
⟨*proof*⟩
lemma *copyableToRes-term-has-no-Empty* [*simp*]:
list-all ($\lambda x.$ $\neg$ *is-Empty x*) (*map copyableToRes-term xs*) = *list-all* ($\lambda x.$ $\neg$ *is-Empty x*) *xs*
⟨*proof*⟩
lemma *copyableToRes-term-is-Parallel* [*simp*]:
is-Parallel (*copyableToRes-term x*) = *is-Parallel x*
⟨*proof*⟩
lemma *copyableToRes-term-has-Parallel* [*simp*]:
list-ex is-Parallel (*map copyableToRes-term xs*) = *list-ex is-Parallel xs*
⟨*proof*⟩
lemma *copyableToRes-term-has-no-Parallel* [*simp*]:
list-all ($\lambda x.$ $\neg$ *is-Parallel x*) (*map copyableToRes-term xs*) = *list-all* ($\lambda x.$ $\neg$ *is-Parallel x*) *xs*
⟨*proof*⟩

Replacing copyable resource terms does not affect whether they are normalised

lemma *normalised-copyableToRes-term* [*simp*]:

normalised (*copyableToRes-term* x) = *normalised* x (**is** *normalised* (*?f* x) = *normalised* x)
— Note the pattern matching, which is needed to later refer to *copyableToRes-term* with the right type variable for copyable resources in its output
⟨*proof*⟩

Term rewriting step commutes with the copyable term replacement

lemma *remove-one-empty-copyableToRes-term-commute*:
remove-one-empty (*map copyableToRes-term xs*) = *map copyableToRes-term* (*remove-one-empty xs*)
⟨*proof*⟩

lemma *merge-one-parallel-copyableToRes-term-commute*:
merge-one-parallel (*map copyableToRes-term xs*) = *map copyableToRes-term* (*merge-one-parallel xs*)
⟨*proof*⟩

lemma *step-copyableToRes-term*:
step (*copyableToRes-term* x) = *copyableToRes-term* (*step* x) (**is** *step* (*?f* x) = *?f* (*step* x))
⟨*proof*⟩

By induction, the replacement of copyable terms also passes through term normalisation

lemma *normal-rewr-copyableToRes-term*:
normal-rewr (*copyableToRes-term* x) = *copyableToRes-term* (*normal-rewr* x)
⟨*proof*⟩

Copyable term replacement is injective

lemma *copyableToRes-term-inj*:
copyableToRes-term x = *copyableToRes-term* $y \Longrightarrow x = y$
⟨*proof*⟩

Making Copyable Resources Linear

We then lift the term-level replacement to resources

lift-definition *copyableToRes* :: ($'a$, $'b$) *resource* ⇒ ($'a$ + $'b$, $'c$) *resource*
is *copyableToRes-term* ⟨*proof*⟩

lemma *copyableToRes-simps* [*simp*]:
copyableToRes Empty = *Empty*
copyableToRes Anything = *Anything*
copyableToRes (*Res a*) = *Res* (*Inl a*)
copyableToRes (*Copyable a*) = *Res* (*Inr a*)
copyableToRes (*Parallel xs*) = *Parallel* (*map copyableToRes xs*)
copyableToRes (*NonD x y*) = *NonD* (*copyableToRes x*) (*copyableToRes y*)
copyableToRes (*Executable x y*) = *Executable* (*copyableToRes x*) (*copyableToRes y*)

copyableToRes (*Repeatable* *x* *y*) = *Repeatable* (*copyableToRes* *x*) (*copyableToRes* *y*)
⟨*proof*⟩

Resource-level replacement is injective, which is vital for preserving composition validity

lemma *copyableToRes-inj*:
fixes *x* *y* :: ('*a*, '*b*) *resource*
shows (*copyableToRes* *x* :: ('*a* + '*b*, '*c*) *resource*) = *copyableToRes* *y* ⟹ *x* = *y*
⟨*proof*⟩

lemma *copyableToRes-eq-conv* [*simp*]:
(*copyableToRes* *x* = *copyableToRes* *y*) = (*x* = *y*)
⟨*proof*⟩

Resource-level replacement can then be applied over a process

primrec *process-copyableToRes* :: ('*a*, '*b*, '*l*, '*m*) *process* ⇒ ('*a* + '*b*, '*c*, '*l*, '*m*) *process*
where
process-copyableToRes (*Primitive* *ins* *outs* *l* *m*) =
Primitive (*copyableToRes* *ins*) (*copyableToRes* *outs*) *l* *m*
| *process-copyableToRes* (*Identity* *a*) = *Identity* (*copyableToRes* *a*)
| *process-copyableToRes* (*Swap* *a* *b*) = *Swap* (*copyableToRes* *a*) (*copyableToRes* *b*)
| *process-copyableToRes* (*Seq* *p* *q*) = *Seq* (*process-copyableToRes* *p*) (*process-copyableToRes* *q*)
| *process-copyableToRes* (*Par* *p* *q*) = *Par* (*process-copyableToRes* *p*) (*process-copyableToRes* *q*)
| *process-copyableToRes* (*Opt* *p* *q*) = *Opt* (*process-copyableToRes* *p*) (*process-copyableToRes* *q*)
| *process-copyableToRes* (*InjectL* *a* *b*) = *InjectL* (*copyableToRes* *a*) (*copyableToRes* *b*)
| *process-copyableToRes* (*InjectR* *a* *b*) = *InjectR* (*copyableToRes* *a*) (*copyableToRes* *b*)
| *process-copyableToRes* (*OptDistrIn* *a* *b* *c*) =
OptDistrIn (*copyableToRes* *a*) (*copyableToRes* *b*) (*copyableToRes* *c*)
| *process-copyableToRes* (*OptDistrOut* *a* *b* *c*) =
OptDistrOut (*copyableToRes* *a*) (*copyableToRes* *b*) (*copyableToRes* *c*)
| *process-copyableToRes* (*Duplicate* *a*) = *undefined*
— There is no sensible definition for *Duplicate*, but we will not need one
| *process-copyableToRes* (*Erase* *a*) = *undefined*
— There is no sensible definition for *Erase*, but we will not need one
| *process-copyableToRes* (*Represent* *p*) = *Represent* (*process-copyableToRes* *p*)
| *process-copyableToRes* (*Apply* *a* *b*) = *Apply* (*copyableToRes* *a*) (*copyableToRes* *b*)
| *process-copyableToRes* (*Repeat* *a* *b*) = *Repeat* (*copyableToRes* *a*) (*copyableToRes* *b*)
| *process-copyableToRes* (*Close* *a* *b*) = *Close* (*copyableToRes* *a*) (*copyableToRes* *b*)

| *process-copyableToRes* (*Once a b*) = *Once* (*copyableToRes a*) (*copyableToRes b*)
| *process-copyableToRes* (*Forget a*) = *Forget* (*copyableToRes a*)

12.3 Final Properties

The final transformation proceeds by first *makeDuplEraToPrim* to remove the resource actions that depend on their copyable nature and then *process-copyableToRes* to make all copyable resources into linear ones. We verify that the result:

- Has the expected type,
- Has as input the original input made linear,
- Has as output the original output made linear,
- Is valid iff the original is valid.
- Contains no copyable atoms

notepad begin
⟨*proof*⟩
end

lemma *eliminateCopyable-input*:
input (*process-copyableToRes* (*makeDuplEraToPrim x*)) = *copyableToRes* (*input x*)
⟨*proof*⟩

lemma *eliminateCopyable-output*:
output (*process-copyableToRes* (*makeDuplEraToPrim x*)) = *copyableToRes* (*output x*)
⟨*proof*⟩

lemma *eliminateCopyable-valid*:
valid (*process-copyableToRes* (*makeDuplEraToPrim x*)) = *valid x*
⟨*proof*⟩

lemma *set2-process-eliminateCopyable*:
fixes *x* :: ('*a*, '*b*, '*l*, '*m*) *process*
shows *set2-process* (*process-copyableToRes* (*makeDuplEraToPrim x*)) = {}
⟨*proof*⟩

end

References

[1] B. Fürer, A. Lochbihler, J. Schneider, and D. Traytel. Quotients of bounded natural functors. In N. Peltier and V. Sofronie-Stokkermans,

editors, *Automated Reasoning*, pages 58–78, Cham, 2020. Springer International Publishing.