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*:
  $(\neg\ \mathit{list\text{-}all}\ P\ xs) = \mathit{list\text{-}ex}\ (\lambda x.\ \neg\ P\ x)\ xs$
  by (*metis Ball-set Bex-set*)
lemma *not-list-ex*:
  $(\neg\ \mathit{list\text{-}ex}\ P\ xs) = \mathit{list\text{-}all}\ (\lambda x.\ \neg\ P\ x)\ xs$
  by (*metis Ball-set Bex-set*)

A list of length more than one starts with two elements

lemma *list-obtain-2*:
assumes $1 < \mathit{length}\ xs$
obtains $v\ vb\ vc$ **where** $xs = v \# vb \# vc$
using *assms* **by** (*cases xs rule*: *remdups-adj.cases*) *simp-all*

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-monoid-add*, *ordered-cancel-ab-semigroup-add*, *linorder*$\}$
assumes $k < l$
and $m + l = k + n$
shows $m < n$
using *assms* **by** (*metis add.commute add-strict-left-mono linorder-not-less nless-le*)

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-monoid-add*, *ordered-cancel-ab-semigroup-add*, *linorder*$\}$
assumes $\bigwedge x.\ x \in \mathit{set}\ xs \Longrightarrow f\ x \le g\ x$
and $x \in \mathit{set}\ xs$
and $f\ x < g\ x$
shows *sum-list* (*map f xs*) $<$ *sum-list* (*map g xs*)
proof −
have *sum-list* (*map f xs*) $\le$ *sum-list* (*map g xs*)
using *assms sum-list-mono* **by** *blast*
moreover have *sum-list* (*map f xs*) $\ne$ *sum-list* (*map g xs*)
proof
assume *sum-list* (*map f xs*) $=$ *sum-list* (*map g xs*)
then have *sum-list* (*map f* (*remove1 x xs*)) $>$ *sum-list* (*map g* (*remove1 x xs*))
by (*metis add.commute assms*(*2*,*3*) *less-add-eq-less-general sum-list-map-remove1*)
then show *False*
by (*metis assms*(*1*) *leD notin-set-remove1 sum-list-mono*)
qed
ultimately show *?thesis*
by *simp*
qed

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

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

proof (*induct xs arbitrary*: *ys*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a as*)
  moreover obtain *b bs* **where** *ys* = *b* # *bs*
    using *Cons* **by** (*meson list-all2-Cons1*)
  ultimately show *?case*
    by (*simp add*: *add-mono*)
qed

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 −
  note *len* = *list-all2-lengthD*[*OF assms*(*1*)]

  have $sum\text{-}list\ (map\ f\ xs) = (\sum x \leftarrow zip\ xs\ ys.\ f\ (fst\ x))$
  proof −
    have $map\ f\ xs = map\ f\ (map\ fst\ (zip\ xs\ ys))$
      using *len* **by** *simp*
    then have $map\ f\ xs = map\ (\lambda x.\ f\ (fst\ x))\ (zip\ xs\ ys)$
      by *simp*
    then show *?thesis*
      by *metis*
  qed
  moreover have $sum\text{-}list\ (map\ g\ ys) = (\sum x \leftarrow zip\ xs\ ys.\ g\ (snd\ x))$
  proof −
    have $map\ g\ ys = map\ g\ (map\ snd\ (zip\ xs\ ys))$
      using *len* **by** *simp*
    then have $map\ g\ ys = map\ (\lambda x.\ g\ (snd\ x))\ (zip\ xs\ ys)$
      by *simp*
    then show *?thesis*
      by *metis*
  qed
  moreover have $x \in set\ (zip\ xs\ ys) \Longrightarrow f\ (fst\ x) \leq g\ (snd\ x)$ **for** x
    using *assms*(*1*) **by** (*fastforce simp add*: *in-set-zip list-all2-conv-all-nth*)
  ultimately show *?thesis*
    using *assms*(*2*,*3*) **by** (*simp add*: *sum-list-mono-one-strict*)
qed

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-if P [] = *0*
  | *count-if P* ($x\#xs$) = (*if P x then Suc* (*count-if P xs*) *else count-if P xs*)

lemma *count-if-append* [*simp*]:
  count-if P (*xs* @ *ys*) = *count-if P xs* + *count-if P ys*
  by (*induct xs*) *simp-all*

lemma *count-if-0-conv*:
  (*count-if P xs* = *0*) = ($\neg$ *list-ex P xs*)
  by (*induct xs*) *simp-all*

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 $(\bigcap x \in A.\ f\ x) = f\ x$
  using *assms* **by** *blast*

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*:
$x \in set\ xs \Longrightarrow size\text{-}res\text{-}term\ f\ g\ x < size\text{-}res\text{-}term\ f\ g\ (Parallel\ xs)$
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
by *simp* (*metis add-Suc-right less-SucI less-add-Suc1 trans-less-add2*)
qed

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

lemma *parallel-neq-single* [*simp*]:
$Parallel\ [a] \neq a$
proof −
have $\bigwedge f\ g.\ size\text{-}res\text{-}term\ f\ g\ a < size\text{-}res\text{-}term\ f\ g\ (Parallel\ [a])$
using *parallel-child-smaller* **by** *simp*
then show *?thesis*
by *fastforce*
qed

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* ⇒ ('*a*, '*b*) *res-term* ⇒ *bool* (**infix** ∼ *100*)
  where
    nil: *Parallel* [] ∼ *Empty*
  | *singleton*: *Parallel* [*a*] ∼ *a*
  | *merge*: *Parallel* (*x* @ [*Parallel y*] @ *z*) ∼ *Parallel* (*x* @ *y* @ *z*)
  | *empty*: *Empty* ∼ *Empty*
  | *anything*: *Anything* ∼ *Anything*
  | *res*: *Res x* ∼ *Res x*
  | *copyable*: *Copyable x* ∼ *Copyable x*
  | *parallel*: *list-all2* (∼) *xs ys* ⟹ *Parallel xs* ∼ *Parallel ys*
  | *nondet*: ⟦*x* ∼ *y*; *u* ∼ *v*⟧ ⟹ *NonD x u* ∼ *NonD y v*
  | *executable*: ⟦*x* ∼ *y*; *u* ∼ *v*⟧ ⟹ *Executable x u* ∼ *Executable y v*
  | *repeatable*: ⟦*x* ∼ *y*; *u* ∼ *v*⟧ ⟹ *Repeatable x u* ∼ *Repeatable y v*
  | *sym* [*sym*]: *x* ∼ *y* ⟹ *y* ∼ *x*
  | *trans* [*trans*]: ⟦*x* ∼ *y*; *y* ∼ *z*⟧ ⟹ *x* ∼ *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

setup ‹*Sign.mandatory-path res-term-equiv*›

It can be shown to be reflexive

lemma *refl* [*simp*]:
  a ∼ *a*
  by (*induct a* ; *rule res-term-equiv.intros* ; *simp add*: *list-all2-same*)

lemma *reflI*:
  a = *b* ⟹ *a* ∼ *b*
  by *simp*

lemma *equivp* [*simp*]:

equivp res-term-equiv
by (*simp add*: *equivpI reflpI res-term-equiv.sym res-term-equiv.trans sympI transpI*)

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 –
have *Parallel* [*Parallel x1*, *Parallel x2*] $\sim$ *Parallel* [*Parallel y1*, *Parallel y2*]
by (*simp add*: *assms res-term-equiv.parallel*)
then have *Parallel* (*Parallel x1* # *x2*) $\sim$ *Parallel* (*Parallel y1* # *y2*)
using *res-term-equiv.merge*[*of* [*Parallel x1*] *x2 Nil*, *simplified*]
res-term-equiv.merge[*of* [*Parallel y1*] *y2 Nil*, *simplified*]
by (*meson res-term-equiv.sym res-term-equiv.trans*)
then show *Parallel* (*x1* @ *x2*) $\sim$ *Parallel* (*y1* @ *y2*)
using *res-term-equiv.merge*[*of Nil y1 y2*, *simplified*]
res-term-equiv.merge[*of Nil x1 x2*, *simplified*]
by (*meson res-term-equiv.sym res-term-equiv.trans*)
qed

We can drop a unit from any parallel resource term

lemma *drop*:
Parallel (*x* @ [*Empty*] @ *y*) $\sim$ *Parallel* (*x* @ *y*)
proof –
have *Parallel* [*Empty*] $\sim$ *Parallel* [*Parallel* []]
using *res-term-equiv.nil res-term-equiv.sym res-term-equiv.trans res-term-equiv.singleton*
by *blast*
then have *Parallel* (*x* @ [*Empty*] @ *y*) $\sim$ *Parallel* (*x* @ [*Parallel* []] @ *y*)
using *res-term-equiv.decompose*[*OF res-term-equiv.refl*, *of* [*Empty*] @ *y* [*Parallel* []] @ *y x*]
res-term-equiv.decompose[*OF - res-term-equiv.refl*, *of* [*Empty*] [*Parallel* []] *y*]
by *blast*
then show *?thesis*
using *res-term-equiv.merge res-term-equiv.trans* **by** *fastforce*
qed

Equivalent resource terms remain equivalent wrapped in a parallel

lemma *singleton-both*:
$x \sim y \Longrightarrow$ *Parallel* [*x*] $\sim$ *Parallel* [*y*]
by (*simp add*: *res-term-equiv.parallel*)

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

lemma *trans-both*:
$[\![a \sim x;\ y \sim b;\ x \sim y]\!] \Longrightarrow a \sim b$
by (*rule res-term-equiv.trans*[*OF res-term-equiv.trans*])

setup ‹*Sign.parent-path*›

experiment begin
lemma *Parallel* [*Parallel* [], *Empty*] ∼ *Empty*
proof −
  have *Parallel* [*Parallel* [], *Empty*] ∼ *Parallel* [*Parallel* []]
    using *res-term-equiv.drop*[*of* [*Parallel* []]] **by** *simp*
  also have ... ∼ *Parallel* [] **by** *simp*
  also have ... ∼ *Empty* **by** *simp*
  finally show *?thesis* **.**
qed
end

Inserting equivalent terms anywhere in equivalent parallel terms preserves the equivalence

lemma *res-term-parallel-insert*:
  assumes *Parallel x* ∼ *Parallel y*
      and *Parallel u* ∼ *Parallel v*
      and *a* ∼ *b*
    shows *Parallel* (*x* @ [*a*] @ *u*) ∼ *Parallel* (*y* @ [*b*] @ *v*)
  by (*meson assms res-term-equiv.decompose res-term-equiv.singleton-both*)

With inserting at the start being just a special case

lemma *res-term-parallel-cons*:
  assumes *Parallel x* ∼ *Parallel y*
      and *a* ∼ *b*
    shows *Parallel* (*a* # *x*) ∼ *Parallel* (*b* # *y*)
  using *res-term-parallel-insert*[*OF res-term-equiv.refl assms*, *of Nil*] **by** *simp*

Empty is a unit for binary *Parallel*

lemma *res-term-parallel-emptyR* [*simp*]: *Parallel* [*x*, *Empty*] ∼ *x*
  using *res-term-equiv.drop*[*of* [*x*] *Nil*] **by** (*simp add*: *res-term-equiv.trans*)
lemma *res-term-parallel-emptyL* [*simp*]: *Parallel* [*Empty*, *x*] ∼ *x*
  using *res-term-equiv.drop*[*of Nil* [*x*]] **by** (*simp add*: *res-term-equiv.trans*)

Term equivalence is preserved by parallel on either side

lemma *res-term-equiv-parallel* [*simp*]:
  x ∼ *y* ⟹ *x* ∼ *Parallel* [*y*]
  using *res-term-equiv.singleton res-term-equiv.sym res-term-equiv.trans* **by** *blast*
lemmas [*simp*] = *res-term-equiv-parallel*[*symmetric*]

Resource term map preserves equivalence:

lemma *map-res-term-preserves-equiv* [*simp*]:
  x ∼ *y* ⟹ *map-res-term f g x* ∼ *map-res-term f g y*
proof (*induct rule*: *res-term-equiv.induct*)
    case *empty* **then show** *?case* **by** *simp*
next case *anything* **then show** *?case* **by** *simp*

next case (*res x*) **then show** *?case* **by** *simp*
next case (*copyable x*) **then show** *?case* **by** *simp*
next case *nil* **then show** *?case* **by** *simp*
next case (*singleton a*) **then show** *?case* **by** *simp*
next case (*merge x y z*) **then show** *?case* **using** *res-term-equiv.merge* **by** *fastforce*
next
  case (*parallel xs ys*)
  then show *?case*
    by (*simp add*: *list-all2-conv-all-nth res-term-equiv.parallel*)
next case (*nondet x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.nondet*)
next case (*executable x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.executable*)
next case (*repeatable x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.repeatable*)
next case (*sym x y*) **then show** *?case* **by** (*simp add*: *res-term-equiv.sym*)
next case (*trans x y z*) **then show** *?case* **using** *res-term-equiv.trans* **by** *blast*
qed

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 ∼ *Parallel* (*parallel-parts x*)
proof (*induct x*)
    case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*

next case (*Res x*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next
  case (*Parallel xs*)
  then show *?case*
  proof (*induct xs*)
    case *Nil* **then show** *?case* **by** *simp*
  next
    case (*Cons a x*)
    then have *a1*: *a* ∼ *Parallel* (*parallel-parts a*)
        and *a2*: *Parallel x* ∼ *Parallel* (*parallel-parts* (*Parallel x*))
      by *simp-all*

    have *Parallel* [*a*] ∼ *Parallel* (*parallel-parts a*)
      using *a1 res-term-equiv.trans res-term-equiv.singleton* **by** *blast*
   then have *Parallel* (*a* # *x*) ∼ *Parallel* (*parallel-parts a* @ *parallel-parts* (*Parallel x*))
      using *res-term-equiv.decompose*[*OF* - *a2*, *of* [*a*]] **by** *simp*
    then show *?case*
      by *simp*
  qed
next case (*NonD x1 x2*) **then show** *?case* **by** *simp*
next case (*Executable x1 x2*) **then show** *?case* **by** *simp*
next case (*Repeatable x1 x2*) **then show** *?case* **by** *simp*
qed

Equivalent parallel parts is the same as equivalent resource terms

lemma *equiv-parallel-parts*:
  list-all2 (∼) (*parallel-parts a*) (*parallel-parts b*) = *a* ∼ *b*
proof
  show *list-all2* (∼) (*parallel-parts a*) (*parallel-parts b*) ⟹ *a* ∼ *b*
   by (*meson res-term-equiv.parallel parallel-parts-eq res-term-equiv.sym res-term-equiv.trans*)
  show *a* ∼ *b* ⟹ *list-all2* (∼) (*parallel-parts a*) (*parallel-parts b*)
  proof (*induct rule*: *res-term-equiv.induct*)
      case *empty* **then show** *?case* **by** *simp*
  next case *anything* **then show** *?case* **by** *simp*
  next case (*res x*) **then show** *?case* **by** *simp*
  next case (*copyable x*) **then show** *?case* **by** *simp*
  next case *nil* **then show** *?case* **by** *simp*
  next case (*singleton a*) **then show** *?case* **by** (*simp add*: *list-all2-refl*)
  next case (*merge x y z*) **then show** *?case* **by** (*simp add*: *list-all2-refl*)
  next
    case (*parallel xs ys*)
    then show *?case*
      by (*induct rule*: *list-all2-induct* ; *simp add*: *list-all2-appendI*)
  next case (*nondet x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.nondet*)
  next case (*executable x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.executable*)
  next case (*repeatable x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.repeatable*)
   next case (*sym x y*) **then show** *?case* **by** (*simp add*: *list-all2-conv-all-nth*

res-term-equiv.sym)
next case (*trans x y z*) **then show** *?case* **using** *res-term-equiv.trans list-all2-trans*
by *blast*
qed
qed

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
let $?x$ = *NonD* (*Parallel* [*Anything*, *Empty*]) (*Parallel* [])
let $?y$ = *NonD Anything Empty*

show $?x \sim ?y$
by (*simp add*: *res-term-equiv.nondet*)
show *parallel-parts* $?x \neq$ *parallel-parts* $?y$
by *simp*
qed

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

lemma *parallel-parts-length-eq*:
$x \sim y \Longrightarrow$ *length* (*parallel-parts* x) = *length* (*parallel-parts* y)
using *equiv-parallel-parts list-all2-lengthD* **by** *blast*

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

lemma *parallel-parts-nil-equiv-empty*:
(*parallel-parts* a = []) = $a \sim$ *Empty*
using *equiv-parallel-parts list.rel-sel parallel-parts.simps*(*1*) **by** *blast*

Singleton parallel parts imply equivalence to the one element

lemma *parallel-parts-single-equiv-element*:
parallel-parts $a = [x] \Longrightarrow a \sim x$
using *parallel-parts-eq res-term-equiv.trans* **by** *force*

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

lemma *parallel-parts-have-no-empty*:
$x \in$ *set* (*parallel-parts* a) $\Longrightarrow \neg$ *is-Empty* x
by (*induct* a) *fastforce*+
lemma *parallel-parts-have-no-par*:
$x \in$ *set* (*parallel-parts* a) $\Longrightarrow \neg$ *is-Parallel* x
by (*induct* a) *fastforce*+

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

lemma *parallel-parts-not-bigger*:
$x \in$ *set* (*parallel-parts* a) $\Longrightarrow$ *size-res-term* $f\ g\ x \leq$ (*size-res-term* $f\ g\ a$)
proof (*induct* a)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*

next case (*Res x*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next
  case (*Parallel x*)
  then show *?case*
    by (*clarsimp simp add*: *le-SucI size-list-estimation′*)
next case (*NonD a1 a2*) **then show** *?case* **by** *simp*
next case (*Executable a1 a2*) **then show** *?case* **by** *simp*
next case (*Repeatable a1 a2*) **then show** *?case* **by** *simp*
qed

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

lemma *parallel-parts-self* [*simp*]:
  $[\![\neg$ *is-Empty x*; $\neg$ *is-Parallel x*$]\!] \Longrightarrow$ *parallel-parts x* = [*x*]
  by (*cases x*) *simp-all*

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 $\neg$ *list-ex is-Empty xs*
    and $\neg$ *list-ex is-Parallel xs*
   shows *parallel-parts* (*Parallel xs*) = *xs*
  using *assms*
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases a* ; *simp*)
qed

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* ⇒ (*′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 ∼ *Parallel xs*

by (*cases xs rule*: *parallelise.cases*) *simp-all*

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*
by (*elim parallelise.elims*) *simp-all*

Parallelisation and naive parallel combination have the same parallel parts

lemma *parallel-parts-parallelise-eq*:
parallel-parts (*parallelise xs*) = *parallel-parts* (*Parallel xs*)
by (*cases xs rule*: *parallelise.cases*) *simp-all*

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
show *parallelise xs* = *Parallel ys* $\Longrightarrow$ *xs* = [*Parallel ys*] $\vee$ *1* < *length xs* $\wedge$ *xs* = *ys*
by (*fastforce elim*: *parallelise.elims*)
have *xs* = [*Parallel ys*] $\Longrightarrow$ *parallelise xs* = *Parallel ys*
by *simp*
moreover have *1* < *length xs* $\wedge$ *xs* = *ys* $\Longrightarrow$ *parallelise xs* = *Parallel ys*
by *simp* (*metis Suc-lessD length-Cons list.size*(*3*) *nat-neq-iff parallelise.elims*)
ultimately show *xs* = [*Parallel ys*] $\vee$ *1* < *length xs* $\wedge$ *xs* = *ys* $\Longrightarrow$ *parallelise xs* = *Parallel ys*
by *blast*
qed

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*)
by (*simp add*: *parallelise-to-parallel-conv*) (*metis parallel-neq-single*)

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*
using *assms* **by** (*induct xs rule*: *parallelise.induct*) *simp-all*

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*
using *assms parallelise-to-parallel-conv* **by** *blast*

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*]
using *assms parallelise.elims* **by** *blast*

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*]
using *assms* **by** (*cases xs rule*: *parallelise.cases* ; *fastforce*)

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*) = $\bigcup$(*set1-res-term* ‘ *set xs*)
by (*induct xs rule*: *parallelise.induct*) *simp-all*
lemma *set2-res-term-parallelise* [*simp*]:
set2-res-term (*ResTerm.parallelise xs*) = $\bigcup$(*set2-res-term* ‘ *set xs*)
by (*induct xs rule*: *parallelise.induct*) *simp-all*

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* ::
$('a \Rightarrow ('x, 'y)\ \mathit{res\text{-}term}) \Rightarrow ('b \Rightarrow 'y) \Rightarrow ('a, 'b)\ \mathit{res\text{-}term} \Rightarrow ('x, 'y)\ \mathit{res\text{-}term}$
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 −
have *reflexivity*: *refine-res-term f g a* $\sim$ *refine-res-term f' g' a* **for** *a*
— First we prove the simpler case where the two resources are equal, so we can use it later
proof (*induct a*)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res x*) **then show** *?case* **using** *assms*(*2*) **by** *simp*
next case (*Copyable x*) **then show** *?case* **using** *assms*(*3*) **by** *simp*
next
case (*Parallel x*)
then show *?case*
by (*clarsimp intro*!: *res-term-equiv.parallel*)
(*metis* (*mono-tags*, *lifting*) *length-map list-all2-all-nthI nth-map nth-mem*)
next case (*NonD a1 a2*) **then show** *?case* **by** (*simp add*: *res-term-equiv.nondet*)
next case (*Executable a1 a2*) **then show** *?case* **by** (*simp add*: *res-term-equiv.executable*)
next case (*Repeatable a1 a2*) **then show** *?case* **by** (*simp add*: *res-term-equiv.repeatable*)
qed

from *assms* **show** *?thesis*
— Then we prove the general statement by induction on assumed equivalence
proof (*induct rule*: *res-term-equiv.induct*)
case *empty* **then show** *?case* **by** *simp*
next case *anything* **then show** *?case* **by** *simp*
next case (*res x*) **then show** *?case* **by** *simp*
next case (*copyable x*) **then show** *?case* **by** *simp*
next case *nil* **then show** *?case* **by** *simp*
next

case (*singleton a*)
then have *refine-res-term f g* (*Parallel* [*a*]) $\sim$ *refine-res-term f g a*
by *simp*
then show *?case*
using *reflexivity res-term-equiv.trans* **by** *metis*
next
case (*merge x y z*)
have
length (*map* (*refine-res-term f g′*) *x* @
map (*refine-res-term f g′*) *y* @ *map* (*refine-res-term f g′*) *z*)
= *length* (*map* (*refine-res-term f′ g′*) *x* @
map (*refine-res-term f′ g′*) *y* @ *map* (*refine-res-term f′ g′*) *z*)
by *simp*
moreover have
((*map* (*refine-res-term f g*) *x* @
map (*refine-res-term f g*) *y* @ *map* (*refine-res-term f g*) *z*) ! *i*)
$\sim$ ((*map* (*refine-res-term f′ g′*) *x* @
map (*refine-res-term f′ g′*) *y* @ *map* (*refine-res-term f′ g′*) *z*) ! *i*)
if $i <$ *length* x + *length* y + *length* z **for** *i*
by (*metis append.assoc length-append map-append nth-map reflexivity that*)
ultimately have
list-all2 ($\sim$)
(*map* (*refine-res-term f g*) *x*
@ *map* (*refine-res-term f g*) *y*
@ *map* (*refine-res-term f g*) *z*)
(*map* (*refine-res-term f′ g′*) *x*
@ *map* (*refine-res-term f′ g′*) *y*
@ *map* (*refine-res-term f′ g′*) *z*)
by (*smt* (*verit*, *del-insts*) *append-assoc length-append length-map list-all2-all-nthI*)
then have
Parallel (*map* (*refine-res-term f g*) *x* @
[*Parallel* (*map* (*refine-res-term f g*) *y*)] @ *map* (*refine-res-term f g*) *z*)
$\sim$ *Parallel* (*map* (*refine-res-term f′ g′*) *x* @
map (*refine-res-term f′ g′*) *y* @ *map* (*refine-res-term f′ g′*) *z*)
using *res-term-equiv.merge res-term-equiv.parallel res-term-equiv.trans* **by**
blast
then show *?case*
by *simp*
next
case (*parallel xs ys*)
then show *?case*
by (*simp add*: *res-term-equiv.parallel list-all2-conv-all-nth*)
next case (*nondet x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.nondet*)
next case (*executable x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.executable*)
next case (*repeatable x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.repeatable*)
next
case (*sym x y*)
then show *?case*
by (*metis res-term-equiv.sym res-term-equiv.trans reflexivity*)

  next
    case (*trans x y z*)
    then show *?case*
      by (*metis res-term-equiv.sym res-term-equiv.trans reflexivity*)
  qed
qed

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*:
  $x \in set\ (remove\text{-}all\text{-}empty\ xs) \Longrightarrow x \in set\ xs$
proof (*induct xs*)
  case *Nil* **then show** *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case*
    by *simp* (*metis* (*full-types*) *set-ConsD*)
qed

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

lemma *remove-all-empty-none*:
  $\neg$ *list-ex is-Empty xs* $\Longrightarrow$ *remove-all-empty xs* = *xs*
  by (*induct xs* ; *force*)

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

lemma *remove-all-empty-result*:
  $\neg$ *list-ex is-Empty* (*remove-all-empty xs*)
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases a* ; *simp*)
qed

Removing *Empty* terms distributes over appending lists

lemma *remove-all-empty-append*:

remove-all-empty (*xs* @ *ys*) = *remove-all-empty xs* @ *remove-all-empty ys*
proof (*induct xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case* **by** (*cases a* ; *simp*)
qed

Removing *Empty* terms distributes over constructing lists

lemma *remove-all-empty-Cons*:
remove-all-empty (*x* # *xs*) = *remove-all-empty* [*x*] @ *remove-all-empty xs*
using *remove-all-empty-append* **by** (*metis append.left-neutral append-Cons*)

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

lemma *remove-all-empty-equiv*:
Parallel xs ∼ *Parallel* (*remove-all-empty xs*)
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
by (*metis append.left-neutral append-Cons remove-all-empty.simps*(*2*) *res-term-equiv.drop res-term-equiv.refl res-term-equiv.trans res-term-parallel-cons is-Empty-def*)
qed

Removing *Empty* terms does not affect the atom sets

lemma *set1-res-term-remove-all-empty* [*simp*]:
$\bigcup$(*set1-res-term* ‘ *set* (*remove-all-empty xs*)) = $\bigcup$(*set1-res-term* ‘ *set xs*)
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
by (*cases a*) *simp-all*
qed
lemma *set2-res-term-remove-all-empty* [*simp*]:
$\bigcup$(*set2-res-term* ‘ *set* (*remove-all-empty xs*)) = $\bigcup$(*set2-res-term* ‘ *set xs*)
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
by (*cases a*) *simp-all*
qed

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

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

lemma *merge-all-parallel-none*:
  ¬ *list-ex is-Parallel xs* ⟹ *merge-all-parallel xs* = *xs*
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases a* ; *simp*)
qed

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 ⋀*ys*. *Parallel ys* ∈ *set xs* ⟹ ¬ *list-ex is-Parallel ys*
    shows ¬ *list-ex is-Parallel* (*merge-all-parallel xs*)
  using *assms*
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases a* ; *fastforce*)
qed

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 (*induct xs arbitrary*: *ys*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases a* ; *simp*)
qed

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*
  using *merge-all-parallel-append* **by** (*metis append.left-neutral append-Cons*)

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

lemma *merge-all-parallel-equiv*:
  Parallel xs $\sim$ *Parallel* (*merge-all-parallel xs*)
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  have *?case* **if** *a* = *Parallel as* **for** *as*
    using *Cons*
    by (*simp add*: *that*)
     (*metis append.left-neutral append-Cons res-term-equiv.decompose res-term-equiv.singleton*)
  moreover have *?case* **if** $\bigwedge$*as. a* $\neq$ *Parallel as*
    using *Cons* **by** (*cases a*) (*simp-all add*: *that res-term-parallel-cons*)
  ultimately show *?case*
    by *metis*
qed

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* $\in$ *set xs* **and** *list-ex is-Empty ys*
      | *list-ex is-Empty xs*
  using *assms*
proof (*induct xs*)
  case *Nil* **then show** *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases a*) *fastforce+*
qed

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 (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next

case (*Cons a xs*)
then show *?case*
by (*cases a*) *simp-all*
qed
lemma *set2-res-term-merge-all-parallel* [*simp*]:
$\bigcup$(*set2-res-term* ‘ *set* (*merge-all-parallel xs*)) = $\bigcup$(*set2-res-term* ‘ *set xs*)
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
by (*cases a*) *simp-all*
qed

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* ⇒ *bool*
where
normalised Empty = *True*
| *normalised Anything* = *True*
| *normalised* (*Res x*) = *True*
| *normalised* (*Copyable x*) = *True*
| *normalised* (*Parallel xs*) =
(*list-all normalised xs* ∧
list-all (λ*x*. ¬ *is-Empty x*) *xs* ∧ *list-all* (λ*x*. ¬ *is-Parallel x*) *xs* ∧
1 < *length xs*)
| *normalised* (*NonD x y*) = (*normalised x* ∧ *normalised y*)
| *normalised* (*Executable x y*) = (*normalised x* ∧ *normalised y*)
| *normalised* (*Repeatable x y*) = (*normalised x* ∧ *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*
proof (*cases x*)
   case *Empty* **then show** *?thesis* **using** *assms* **by** *simp*
next case *Anything* **then show** *?thesis* **using** *assms* **by** *simp*
next case (*Res x*) **then show** *?thesis* **using** *assms* **by** *simp*
next case (*Copyable x*) **then show** *?thesis* **using** *assms* **by** *simp*
next
  case (*Parallel xs*)
  then consider
      list-ex ($\lambda x.$ $\neg$ *normalised x*) *xs*
    | *list-ex is-Empty xs*
    | *list-ex is-Parallel xs*
    | *length xs* $\leq$ *Suc 0*
    using *assms not-list-ex* **by** *fastforce*
  then show *?thesis*
    using *that*(*1*−*5*) *Parallel*
    by (*metis* (*no-types*, *lifting*) *le-Suc-eq le-zero-eq length-0-conv length-Suc-conv*)
next
  case (*NonD x y*)
  then show *?thesis*
    using *assms that*(*6*,*7*) **by** (*cases normalised x*) *simp-all*
next
  case (*Executable x y*)
  then show *?thesis*
    using *assms that*(*8*,*9*) **by** (*cases normalised x*) *simp-all*
next
  case (*Repeatable x y*)
  then show *?thesis*
    using *assms that*(*10*,*11*) **by** (*cases normalised x*) *simp-all*
qed

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
  using *assms*
proof (*induct xs*)
  case *Nil* **then show** *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case*
   by *simp* (*metis* (*mono-tags*, *lifting*) *append-eq-Cons-conv is-Parallel-def list.pred-inject*)
qed
lemma *obtain-first-empty*:
  assumes *list-ex is-Empty xs*
  obtains *a b c* **where** $xs = a$ @ [*Empty*] @ *c* **and** *list-all* ($\lambda x.$ ¬ *is-Empty x*) *a*
  using *assms*
proof (*induct xs*)
  case *Nil* **then show** *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case*
   by *simp* (*metis* (*mono-tags*, *lifting*) *append-eq-Cons-conv is-Empty-def list.pred-inject*)
qed
lemma *obtain-first-unnormalised*:
  assumes *list-ex* ($\lambda x.$ ¬ *normalised x*) *xs*
 obtains *a b c* **where** $xs = a$ @ [*b*] @ *c* **and** *list-all normalised a* **and** ¬ *normalised*
b
  using *assms*
proof (*induct xs*)
  case *Nil* **then show** *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case*
    by *simp* (*metis* (*mono-tags*, *lifting*) *append-eq-Cons-conv list.pred-inject*)
qed

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

lemma *normalised-map*:
  normalised (*map-res-term f g x*) = *normalised x*
  by (*induct x*) (*simp-all add*: *Ball-set*[*symmetric*])

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*
  by (*induct xs rule*: *remdups-adj.induct* ; *fastforce*)

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

lemma *normalised-parallel-parts-eq*:
  normalised (*Parallel xs*) $\Longrightarrow$ *parallel-parts* (*Parallel xs*) = *xs*
  by (*induct xs rule*: *induct-list012* ; *fastforce*)

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*)
proof (*cases xs rule*: *parallelise.cases*)
  case *1*
  then show *?thesis*
   by *simp*
next
  case (*2 x*)
  then show *?thesis*
   using *assms*(*1*) **by** *simp*
next
  case (*3 v vb vc*)
  then show *?thesis*
   using *assms* **by** (*simp add*: *not-list-ex Ball-set*[*symmetric*])
qed

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:

- *Parallel* [] $\rightarrow$ *Empty*
- *Parallel* [*a*] $\rightarrow$ *a*
- *Parallel* (*x* @ [*Parallel y*] @ *z*) $\rightarrow$ *Parallel* (*x* @ *y* @ *z*)
- *Parallel* (*x* @ [*Empty*] @ *y*) $\rightarrow$ *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 *Parallel* (*?x* @ [*Empty*] @ *?y*) $\sim$ *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 *Parallel* (*?x* @ [*Empty*] @ *?y*) $\sim$ *Parallel* (*?x* @ *?y*).

inductive *res-term-rewrite* :: ('*a*, '*b*) *res-term* ⇒ ('*a*, '*b*) *res-term* ⇒ *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* ⟹ *res-term-rewrite* (*Parallel xs*) (*Parallel ys*)
| *nondet*: ⟦*res-term-rewrite x y*; *res-term-rewrite u v*⟧ ⟹ *res-term-rewrite* (*NonD x u*) (*NonD y v*)
| *executable*: ⟦*res-term-rewrite x y*; *res-term-rewrite u v*⟧ ⟹
    res-term-rewrite (*Executable x u*) (*Executable y v*)
| *repeatable*: ⟦*res-term-rewrite x y*; *res-term-rewrite u v*⟧ ⟹
    res-term-rewrite (*Repeatable x u*) (*Repeatable y v*)

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

setup ‹*Sign.mandatory-path res-term-rewrite*›

The rewrite relation is reflexive

lemma *refl* [*simp*]:
  res-term-rewrite x x
proof (*induct x*)
    case *Empty* **then show** *?case* **by** (*rule res-term-rewrite.empty*)
next case *Anything* **then show** *?case* **by** (*rule res-term-rewrite.anything*)
next case (*Res x*) **then show** *?case* **by** (*rule res-term-rewrite.res*)
next case (*Copyable x*) **then show** *?case* **by** (*rule res-term-rewrite.copyable*)
next
  case (*Parallel x*)
  then show *?case*
    by (*simp add*: *res-term-rewrite.parallel list.rel-refl-strong*)
next case (*NonD x1 x2*) **then show** *?case* **by** (*rule res-term-rewrite.nondet*)
next case (*Executable x1 x2*) **then show** *?case* **by** (*rule res-term-rewrite.executable*)
next case (*Repeatable x1 x2*) **then show** *?case* **by** (*rule res-term-rewrite.repeatable*)
qed

lemma *parallel-one*:
  res-term-rewrite a b ⟹ *res-term-rewrite* (*Parallel* (*xs* @ [*a*] @ *ys*)) (*Parallel* (*xs* @ [*b*] @ *ys*))

using *res-term-rewrite.refl res-term-rewrite.parallel*
by (*metis list.rel-refl list-all2-Cons2 list-all2-appendI*)

setup ‹*Sign.parent-path*›

Every term rewrites to an equivalent term

lemma *res-term-rewrite-imp-equiv*:
res-term-rewrite x y $\Longrightarrow$ *x* $\sim$ *y*
proof (*induct x y rule*: *res-term-rewrite.induct*)
case *empty* **then show** *?case* **by** (*rule res-term-equiv.empty*)
next case *anything* **then show** *?case* **by** (*rule res-term-equiv.anything*)
next case (*res x*) **then show** *?case* **by** (*rule res-term-equiv.res*)
next case (*copyable x*) **then show** *?case* **by** (*intro res-term-equiv.copyable*)
next case *nil* **then show** *?case* **by** (*rule res-term-equiv.nil*)
next case (*singleton a*) **then show** *?case* **by** (*rule res-term-equiv.singleton*)
next case (*merge x y z*) **then show** *?case* **by** (*rule res-term-equiv.merge*)
next case (*drop x z*) **then show** *?case* **by** (*rule res-term-equiv.drop*)
next
case (*parallel xs ys*)
then show *?case*
using *res-term-equiv.parallel list-all2-mono* **by** *blast*
next case (*nondet x y u v*) **then show** *?case* **by** (*intro res-term-equiv.nondet*)
next case (*executable x y u v*) **then show** *?case* **by** (*intro res-term-equiv.executable*)
next case (*repeatable x y u v*) **then show** *?case* **by** (*intro res-term-equiv.repeatable*)
qed

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

lemma *res-term-rewrite-trancl-imp-equiv*:
res-term-rewrite^{++} *x y* $\Longrightarrow$ *x* $\sim$ *y*
proof (*induct rule*: *tranclp-induct*)
case (*base y*)
then show *?case* **using** *res-term-rewrite-imp-equiv* **by** *blast*
next
case (*step y z*)
then show *?case* **using** *res-term-rewrite-imp-equiv res-term-equiv.trans* **by** *blast*
qed

Normalised terms have no distinct term to which they transition

lemma *res-term-rewrite-normalised*:
assumes *normalised x*
shows $\nexists$ *y*. *res-term-rewrite x y* $\wedge$ *x* $\neq$ *y*
proof *safe*
fix *y*
assume *res-term-rewrite x y*
then have *x* = *y*
using *assms*
proof (*induct x y rule*: *res-term-rewrite.induct*)
case *empty* **then show** *?case* **by** *simp*

next case *anything* **then show** *?case* **by** *simp*
next case (*res x*) **then show** *?case* **by** *simp*
next case (*copyable x*) **then show** *?case* **by** *simp*
next case *nil* **then show** *?case* **by** *simp*
next case (*singleton a*) **then show** *?case* **by** *simp*
next case (*merge x y z*) **then show** *?case* **by** *simp*
next case (*drop x z*) **then show** *?case* **by** *simp*
next
case (*parallel xs ys*)
then show *?case*
by *simp* (*smt* (*z3*) *Ball-set list.rel-eq list.rel-mono-strong*)
next case (*nondet x y u v*) **then show** *?case* **by** *simp*
next case (*executable x y u v*) **then show** *?case* **by** *simp*
next case (*repeatable x y u v*) **then show** *?case* **by** *simp*
qed
moreover assume $x \neq y$
ultimately show *False*
by *metis*
qed

lemma *res-term-rewrite-normalisedD*:
$[\![$*res-term-rewrite x y*; *normalised x*$]\!] \Longrightarrow x = y$
by (*drule res-term-rewrite-normalised*) *clarsimp*

Whereas other terms have a distinct term to which they transition

lemma *res-term-rewrite-not-normalised*:
assumes ¬ *normalised x*
shows ∃ *y*. *res-term-rewrite x y* ∧ $x \neq y$
using *assms*
proof (*induct x*)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res x*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next
case (*Parallel xs*)
then show *?case*
proof (*cases list-ex is-Parallel xs*)
case *True*
then obtain *a b c* **where** *xs* = *a* @ [*Parallel b*] @ *c* **and** *list-all* (λ*x*. ¬ *is-Parallel x*) *a*
using *obtain-first-parallel* **by** *metis*
then show *?thesis*
using *Parallel res-term-rewrite.merge*
by (*metis append-eq-append-conv parallel-neq-single res-term.sel*(*3*))
next
case *no-par*: *False*
then show *?thesis*
proof (*cases list-ex is-Empty xs*)

case *True*
then obtain *a c* **where** $xs = a$ @ [*Empty*] @ *c* **and** *list-all* (λx. $\neg$ *is-Empty* x) *a*
using *obtain-first-empty* **by** *metis*
then show *?thesis*
using *no-par Parallel res-term-rewrite.drop* **by** *blast*
next
case *no-empty*: *False*
then show *?thesis*
proof (*cases list-ex* (λx. $\neg$ *normalised* x) *xs*)
case *True*
then obtain *a b c*
where *xs*: $xs = a$ @ [b] @ *c* **and** *list-all normalised a* **and** $\neg$ *normalised b*
using *obtain-first-unnormalised* **by** *metis*
then obtain b' **where** *res-term-rewrite* b b' **and** $b \neq b'$
using *Parallel* **by** (*metis append-eq-Cons-conv in-set-conv-decomp*)
then have *res-term-rewrite* (*Parallel* (*a* @ [b] @ *c*)) (*Parallel* (*a* @ [b'] @ *c*))
and *Parallel* (*a* @ [b] @ *c*) $\neq$ *Parallel* (*a* @ [b'] @ *c*)
using *res-term-rewrite.parallel-one* **by** *blast+*
then show *?thesis*
using *xs* **by** *metis*
next
case *all-normal*: *False*
then consider $xs = []$ | *a* **where** $xs = [a]$
using *no-par no-empty Parallel* **by** (*metis Bex-set normalised-parallelise parallelise.elims*)
then show *?thesis*
using *res-term-rewrite.nil res-term-rewrite.singleton*
by (*metis parallel-neq-single res-term.distinct*(*29*))
qed
qed
qed
next
case (*NonD x1 x2*)
then show *?case*
by (*metis normalised.simps*(*6*) *res-term.inject*(*4*) *res-term-rewrite.nondet res-term-rewrite.refl*)
next
case (*Executable x1 x2*)
then show *?case*
by (*metis normalised.simps*(*7*) *res-term.inject*(*5*) *res-term-rewrite.executable res-term-rewrite.refl*)
next
case (*Repeatable x1 x2*)
then show *?case*
by (*metis normalised.simps*(*8*) *res-term.inject*(*6*) *res-term-rewrite.repeatable res-term-rewrite.refl*)
qed

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

lemma *normalised-is-rewrite-refl*:
  normalised x = (∀ *y*. *res-term-rewrite x y* ⟶ *x* = *y*)
  using *res-term-rewrite-normalised res-term-rewrite-not-normalised* **by** *metis*

Every term rewrites to one of at most equal size

lemma *res-term-rewrite-not-increase-size*:
  res-term-rewrite x y ⟹ *size-res-term f g y* ≤ *size-res-term f g x*
  by (*induct x y rule*: *res-term-rewrite.induct*)
   (*simp-all add*: *list-all2-conv-all-nth size-list-conv-sum-list sum-list-mono-list-all2*)

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* ⇒ *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*:
  ¬ *normalised x* ⟹ *res-term-rewrite-bound x* ≠ *0*
  by (*induct x* ; *fastforce*)

Rewriting relation does not increase this bound

lemma *res-term-rewrite-non-increase-bound*:
  res-term-rewrite x y ⟹ *res-term-rewrite-bound y* ≤ *res-term-rewrite-bound x*
  by (*induct x y rule*: *res-term-rewrite.induct*)
   (*simp-all add*: *sum-list-mono-list-all2 list-all2-conv-all-nth*)

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* ⇒ ('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 ⟹ *remove-one-empty* (*x* # *xs*) = *xs*
  ¬ *is-Empty x* ⟹ *remove-one-empty* (*x* # *xs*) = *x* # *remove-one-empty xs*
  by (*cases x* ; *simp*)+

lemma *remove-one-empty-append*:
  list-all (λ*x*. ¬ *is-Empty x*) *a* ⟹ *remove-one-empty* (*a* @ *d*) = *a* @ *remove-one-empty d*
  by (*induct a* ; *simp*)

lemma *remove-one-empty-distinct*:
  list-ex is-Empty xs ⟹ *remove-one-empty xs* ≠ *xs*
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases a* ; *simp*)
qed

This is identity when there are no *Empty* terms

lemma *remove-one-empty-none* [*simp*]:
  ¬ *list-ex is-Empty xs* ⟹ *remove-one-empty xs* = *xs*
  by (*induct xs rule*: *remove-one-empty.induct* ; *simp*)

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

lemma *length-remove-one-empty* [*simp*]:
  list-ex is-Empty xs ⟹ *length* (*remove-one-empty xs*) + *1* = *length xs*
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case*
    by (*cases is-Empty a* ; *simp*)

qed

Removing an *Empty* term does not increase the size

lemma *remove-one-empty-not-increase-size*:
size-res-term f g (*Parallel* (*remove-one-empty xs*)) $\leq$ *size-res-term f g* (*Parallel xs*)
by (*induct xs rule*: *remove-one-empty.induct* ; *simp*)

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

lemma *remove-one-empty-equiv*:
Parallel xs $\sim$ *Parallel* (*remove-one-empty xs*)
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
proof (*cases is-Empty a*)
case *True*
then show *?thesis*
using *res-term-equiv.drop*[*of Nil*] *Cons* **by** (*fastforce simp add*: *is-Empty-def*)
next
case *False*
then show *?thesis*
using *Cons* **by** (*simp add*: *res-term-parallel-cons*)
qed
qed

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

lemma *remove-one-empty-map*:
map (*map-res-term f g*) (*remove-one-empty xs*) = *remove-one-empty* (*map* (*map-res-term f g*) *xs*)
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case* **by** (*cases is-Empty a* ; *simp*)
qed

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-one-empty xs*) $\Longrightarrow$ $x \in$ *set xs*
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons a xs*)

then show *?case*
by (*cases is-Empty a* ; *simp*) *blast*
qed

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*
by (*cases x* ; *simp*)

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*
by (*induct a* ; *simp*)

lemma *merge-one-parallel-distinct*:
list-ex is-Parallel xs ⟹ *merge-one-parallel xs* ≠ *xs*
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case* **by** (*cases a* ; *simp*) (*metis parallel-neq-single*)
qed

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*
by (*induct xs rule*: *merge-one-parallel.induct* ; *simp*)

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 (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case* **by** (*cases a* ; *simp*)

qed

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 (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case*
  proof (*cases is-Parallel a*)
    case *True*
    then show *?thesis*
     using *Cons res-term-equiv.merge*[*of Nil*] **by** (*fastforce simp add*: *is-Parallel-def*)
  next
    case *False*
    then show *?thesis*
      using *Cons* **by** (*simp add*: *res-term-parallel-cons*)
  qed
qed

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 (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases a* ; *simp*)
qed

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* ⟹ *P*
    and *x* = *Anything* ⟹ *P*
    and ⋀*a*. *x* = *Res a* ⟹ *P*
    and ⋀*u*. *x* = *Copyable u* ⟹ *P*
    and ⋀*u v*. ⟦¬ *normalised u*; *x* = *NonD u v*⟧ ⟹ *P*
    and ⋀*u v*. ⟦*normalised u*; ¬ *normalised v*; *x* = *NonD u v*⟧ ⟹ *P*
    and ⋀*u v*. ⟦*normalised u*; *normalised v*; *x* = *NonD u v*⟧ ⟹ *P*
    and ⋀*u v*. ⟦¬ *normalised u*; *x* = *Executable u v*⟧ ⟹ *P*
    and ⋀*u v*. ⟦*normalised u*; ¬ *normalised v*; *x* = *Executable u v*⟧ ⟹ *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 (*cases x*)
case *Empty* **then show** *?thesis* **using** *assms* **by** *simp*
next case *Anything* **then show** *?thesis* **using** *assms* **by** *simp*
next case (*Res x3*) **then show** *?thesis* **using** *assms* **by** *simp*
next case (*Copyable x4*) **then show** *?thesis* **using** *assms* **by** *simp*
next
case (*Parallel xs*)
then show *?thesis*
proof (*cases list-ex* ($\lambda x.\ \neg$ *normalised x*) *xs*)
case *True*
then show *?thesis*
using *assms*(*14*) **by** (*meson Bex-set Parallel*)
next
case *not-norm*: *False*
then show *?thesis*
proof (*cases list-ex is-Parallel xs*)
case *True*
then show *?thesis*
using *Parallel assms*(*14*,*15*) **by** *blast*
next
case *not-par*: *False*
then show *?thesis*
proof (*cases list-ex is-Empty xs*)
case *True*
then show *?thesis*
by (*metis not-par Parallel assms*(*14*,*16*) *not-list-ex*)
next
case *not-empty*: *False*
then show *?thesis*
proof (*cases xs rule*: *remdups-adj.cases*)
case *1*

then show *?thesis*
by (*simp add*: *Parallel assms*(*17*))
next
case (*2 x*)
then show *?thesis*
using *Parallel assms*(*14*,*18*) *not-empty not-par* **by** *fastforce*
next
case (*3 x y xs*)
then show *?thesis*
by (*metis Parallel assms*(*14*,*19*) *not-empty not-list-ex not-par*)
qed
qed
qed
qed
next case (*NonD x61 x62*) **then show** *?thesis* **using** *assms*(*5*−*7*) **by** *blast*
next case (*Executable x71 x72*) **then show** *?thesis* **using** *assms*(*8*−*10*) **by** *blast*
next case (*Repeatable x71 x72*) **then show** *?thesis* **using** *assms*(*11*−*13*) **by** *blast*
qed

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 (*induct x*)
case *Empty* **then show** *?case* **using** *assms* **by** *simp*
next case *Anything* **then show** *?case* **using** *assms* **by** *simp*
next case (*Res x*) **then show** *?case* **using** *assms* **by** *simp*
next case (*Copyable x*) **then show** *?case* **using** *assms* **by** *simp*
next
case (*Parallel xs*)
then show *?case*
proof (*cases list-ex* (λ*x*. ¬ *normalised x*) *xs*)
case *True*
then show *?thesis*
using *assms*(*14*) **by** (*metis Bex-set Parallel*)
next
case *not-norm*: *False*
then show *?thesis*
proof (*cases list-ex is-Parallel xs*)
case *True*
then show *?thesis*
using *Parallel assms*(*14*,*15*) **by** *blast*
next
case *not-par*: *False*
then show *?thesis*
proof (*cases list-ex is-Empty xs*)
case *True*
then show *?thesis*
by (*metis not-par Parallel assms*(*14*,*16*) *not-list-ex*)
next
case *not-empty*: *False*
then show *?thesis*
proof (*cases xs rule*: *remdups-adj.cases*)
case *1*
then show *?thesis*
by (*simp add*: *Parallel assms*(*17*))
next
case (*2 x*)
then show *?thesis*
using *Parallel assms*(*14*,*18*) *not-empty not-par* **by** *fastforce*
next
case (*3 x y xs*)
then show *?thesis*
by (*metis Parallel assms*(*14*,*19*) *not-empty not-list-ex not-par*)
qed

qed
qed
qed
next case (*NonD x61 x62*) **then show** *?case* **using** *assms*(*5*−*7*) **by** *blast*
next case (*Executable x71 x72*) **then show** *?case* **using** *assms*(*8*−*10*) **by** *blast*
next case (*Repeatable x71 x72*) **then show** *?case* **using** *assms*(*11*−*13*) **by** *blast*
qed

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-Parallel xs;
step (*Parallel xs*) = *Parallel* (*merge-one-parallel xs*)⟧
$\Longrightarrow$ *P* (*Parallel xs*)
and $\bigwedge$*xs*. ⟦$\bigwedge$*x*. *x* ∈ *set xs* $\Longrightarrow$ *P x*; ∀ *a*. *a* ∈ *set xs* ⟶ *normalised a*
; *list-all* (λ*x*. ¬ *is-Parallel x*) *xs*; *list-ex is-Empty xs*
; *step* (*Parallel xs*) = *Parallel* (*remove-one-empty xs*)⟧
$\Longrightarrow$ *P* (*Parallel xs*)
and *P* (*Parallel* [])
and $\bigwedge$*u*. ⟦*P u*; *normalised u*; ¬ *is-Parallel u*; ¬ *is-Empty u*; *step* (*Parallel* [*u*]) = *u*⟧
$\Longrightarrow$ *P* (*Parallel* [*u*])
and $\bigwedge$*v vb vc*.
⟦ $\bigwedge$*x*. *x* ∈ *set* (*v* # *vb* # *vc*) $\Longrightarrow$ *P x*; ∀ *a*. *a* ∈ *set* (*v* # *vb* # *vc*) ⟶ *normalised a*
; *list-all* (λ*x*. ¬ *is-Parallel x*) (*v* # *vb* # *vc*)
; *list-all* (λ*x*. ¬ *is-Empty x*) (*v* # *vb* # *vc*)
; *step* (*Parallel* (*v* # *vb* # *vc*)) = *Parallel* (*v* # *vb* # *vc*)⟧
$\Longrightarrow$ *P* (*Parallel* (*v* # *vb* # *vc*))
shows *P x*
proof (*induct x rule*: *step-induct*)
case *Empty* **then show** *?case* **using** *assms*(*1*) **by** *simp*
next case *Anything* **then show** *?case* **using** *assms*(*2*) **by** *simp*
next case (*Res a*) **then show** *?case* **using** *assms*(*3*) **by** *simp*
next case (*Copyable x*) **then show** *?case* **using** *assms*(*4*) **by** *simp*
next case (*NonD-L x y*) **then show** *?case* **using** *assms*(*5*) **by** *simp*
next case (*NonD-R x y*) **then show** *?case* **using** *assms*(*6*) **by** *simp*
next case (*NonD x y*) **then show** *?case* **using** *assms*(*7*) **by** *simp*
next case (*Executable-L x y*) **then show** *?case* **using** *assms*(*8*) **by** *simp*
next case (*Executable-R x y*) **then show** *?case* **using** *assms*(*9*) **by** *simp*
next case (*Executable x y*) **then show** *?case* **using** *assms*(*10*) **by** *simp*
next case (*Repeatable-L x y*) **then show** *?case* **using** *assms*(*11*) **by** *simp*
next case (*Repeatable-R x y*) **then show** *?case* **using** *assms*(*12*) **by** *simp*
next case (*Repeatable x y*) **then show** *?case* **using** *assms*(*13*) **by** *simp*
next case (*Par-Norm xs*) **then show** *?case* **using** *assms*(*14*) **by** (*simp add*: *Bex-set*[*symmetric*] *Bex-def*)
next case (*Par-Par xs*) **then show** *?case* **using** *assms*(*15*) **by** (*simp add*: *Bex-set*[*symmetric*] *Bex-def*)
next
case (*Par-Empty xs*)
then show *?case*
using *assms*(*15*,*16*) **by** (*metis* (*mono-tags*, *lifting*) *list-ex-iff step.simps*(*8*))
next case *Par-Nil* **then show** *?case* **using** *assms*(*17*) **by** *simp*
next case (*Par-Single u*) **then show** *?case* **using** *assms*(*18*) **by** *simp*
next
case (*Par v vb vc*)
then show *?case*
proof (*rule assms*(*19*))
show $\bigwedge$*x*. *x* ∈ *set* (*v* # *vb* # *vc*) $\Longrightarrow$ *x* ∈ *set* (*v* # *vb* # *vc*)
by *simp*

show *step* (*Parallel* (*v* # *vb* # *vc*)) = *Parallel* (*v* # *vb* # *vc*)
using *Par* **by** (*simp add*: *Ball-set*[*symmetric*] *Bex-set*[*symmetric*])
qed
qed

Set of atoms remains unchanged by rewriting step

lemma *set1-res-term-step* [*simp*]:
set1-res-term (*step x*) = *set1-res-term x*
proof (*induct x rule*: *step-induct'*)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res a*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next case (*NonD-L x y*) **then show** *?case* **by** *simp*
next case (*NonD-R x y*) **then show** *?case* **by** *simp*
next case (*NonD x y*) **then show** *?case* **by** *simp*
next case (*Executable-L x y*) **then show** *?case* **by** *simp*
next case (*Executable-R x y*) **then show** *?case* **by** *simp*
next case (*Executable x y*) **then show** *?case* **by** *simp*
next case (*Repeatable-L x y*) **then show** *?case* **by** *simp*
next case (*Repeatable-R x y*) **then show** *?case* **by** *simp*
next case (*Repeatable x y*) **then show** *?case* **by** *simp*
next case (*Par-Norm xs*) **then show** *?case* **by** *simp*
next
case (*Par-Par xs*)
then show *?case*
by (*fastforce elim*!: *obtain-first-parallel simp add*: *merge-one-parallel-append*)
next
case (*Par-Empty xs*)
then show *?case*
by (*fastforce elim*!: *obtain-first-empty simp add*: *remove-one-empty-append*)
next case *Par-Nil* **then show** *?case* **by** *simp*
next case (*Par-Single u*) **then show** *?case* **by** *simp*
next case (*Par v vb vc*) **then show** *?case* **by** *simp*
qed

lemma *set2-res-term-step* [*simp*]:
set2-res-term (*step x*) = *set2-res-term x*
proof (*induct x rule*: *step-induct'*)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res a*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next case (*NonD-L x y*) **then show** *?case* **by** *simp*
next case (*NonD-R x y*) **then show** *?case* **by** *simp*
next case (*NonD x y*) **then show** *?case* **by** *simp*
next case (*Executable-L x y*) **then show** *?case* **by** *simp*
next case (*Executable-R x y*) **then show** *?case* **by** *simp*
next case (*Executable x y*) **then show** *?case* **by** *simp*

next case (*Repeatable-L x y*) **then show** *?case* **by** *simp*
next case (*Repeatable-R x y*) **then show** *?case* **by** *simp*
next case (*Repeatable x y*) **then show** *?case* **by** *simp*
next case (*Par-Norm xs*) **then show** *?case* **by** *simp*
next
 case (*Par-Par xs*)
 then show *?case*
  by (*fastforce elim*!: *obtain-first-parallel simp add*: *merge-one-parallel-append*)
next
 case (*Par-Empty xs*)
 then show *?case*
  by (*fastforce elim*!: *obtain-first-empty simp add*: *remove-one-empty-append*)
next case *Par-Nil* **then show** *?case* **by** *simp*
next case (*Par-Single u*) **then show** *?case* **by** *simp*
next case (*Par v vb vc*) **then show** *?case* **by** *simp*
qed

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 (*induct x rule*: *step-induct′*)
  case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res a*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next case (*NonD-L x y*) **then show** *?case* **by** (*simp add*: *res-term-rewrite.nondet*)
next case (*NonD-R x y*) **then show** *?case* **by** (*simp add*: *res-term-rewrite.nondet*)
next case (*NonD x y*) **then show** *?case* **by** *simp*
next case (*Executable-L x y*) **then show** *?case* **by** (*simp add*: *res-term-rewrite.executable*)
next case (*Executable-R x y*) **then show** *?case* **by** (*simp add*: *res-term-rewrite.executable*)
next case (*Executable x y*) **then show** *?case* **by** *simp*
next case (*Repeatable-L x y*) **then show** *?case* **by** (*simp add*: *res-term-rewrite.repeatable*)
next case (*Repeatable-R x y*) **then show** *?case* **by** (*simp add*: *res-term-rewrite.repeatable*)
next case (*Repeatable x y*) **then show** *?case* **by** *simp*
next
 case (*Par-Norm xs*)
 then show *?case*
  by (*simp add*: *Bex-set*[*symmetric*] *res-term-rewrite.intros*(*9*) *list.rel-map*(*2*) *list-all2-same*)
next
 case (*Par-Par xs*)
 moreover obtain *a b c* **where** *xs* = *a* @ [*Parallel b*] @ *c* **and** *list-all* (*λx*. ¬ *is-Parallel x*) *a*
  using *Par-Par*(*3*) *obtain-first-parallel* **by** *blast*
 moreover have *res-term-rewrite* (*Parallel* (*a* @ [*Parallel b*] @ *c*)) (*Parallel* (*a* @ *b* @ *c*))
  using *res-term-rewrite.intros*(*7*) **.**
 ultimately show *?case*

by (*simp add*: *Bex-set*[*symmetric*] *merge-one-parallel-append*)
next
case (*Par-Empty xs*)
moreover obtain *a c* **where** *xs* = *a* @ [*Empty*] @ *c* **and** *list-all* (λx. ¬ *is-Empty x*) *a*
using *Par-Empty*(*4*) *obtain-first-empty* **by** *blast*
moreover have *res-term-rewrite* (*Parallel* (*a* @ [*Empty*] @ *c*)) (*Parallel* (*a* @ *c*))
using *res-term-rewrite.intros*(*8*) **.**
ultimately show *?case*
by (*simp add*: *Bex-set*[*symmetric*] *remove-one-empty-append*)
next case *Par-Nil* **then show** *?case* **by** (*simp add*: *res-term-rewrite.intros*(*5*))
next case (*Par-Single u*) **then show** *?case* **by** (*simp add*: *res-term-rewrite.intros*(*6*))
next case (*Par v vb vc*) **then show** *?case* **by** *simp*
qed

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

lemma *normalised-is-step-id*:
normalised x = (*step x* = *x*)
proof
show *normalised x* $\Longrightarrow$ *step x* = *x*
by (*metis res-term-rewrite-contains-step res-term-rewrite-normalised*)
show *step x* = *x* $\Longrightarrow$ *normalised x*
proof (*induct x rule*: *step-induct'*)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res a*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next case (*NonD-L x y*) **then show** *?case* **by** *simp*
next case (*NonD-R x y*) **then show** *?case* **by** *simp*
next case (*NonD x y*) **then show** *?case* **by** *simp*
next case (*Executable-L x y*) **then show** *?case* **by** *simp*
next case (*Executable-R x y*) **then show** *?case* **by** *simp*
next case (*Executable x y*) **then show** *?case* **by** *simp*
next case (*Repeatable-L x y*) **then show** *?case* **by** *simp*
next case (*Repeatable-R x y*) **then show** *?case* **by** *simp*
next case (*Repeatable x y*) **then show** *?case* **by** *simp*
next case (*Par-Norm xs*) **then show** *?case* **by** *simp* (*metis map-eq-conv map-ident*)
next case (*Par-Par xs*) **then show** *?case* **by** (*simp add*: *merge-one-parallel-distinct*)
next case (*Par-Empty xs*) **then show** *?case* **by** (*simp add*: *remove-one-empty-distinct*)
next case *Par-Nil* **then show** *?case* **by** *simp*
next case (*Par-Single u*) **then show** *?case* **by** *simp*
next case (*Par v vb vc*) **then show** *?case* **by** (*simp add*: *Ball-set*[*symmetric*])
qed
qed

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

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

using *normalised-is-step-id* **by** (*rule iffD1*)

Rewriting step never increases the term size

lemma *step-not-increase-size*:
size-res-term f g (*step x*) $\leq$ *size-res-term f g x*
using *res-term-rewrite-not-increase-size res-term-rewrite-contains-step* **by** *blast*

Every resource is equivalent to itself after the step

lemma *res-term-equiv-step*:
$x \sim$ *step x*
using *res-term-rewrite-contains-step res-term-rewrite-imp-equiv* **by** *blast*

Normalisation step commutes with the resource term map

lemma *step-map*:
map-res-term f g (*step x*) = *step* (*map-res-term f g x*)
proof (*induct x rule*: *step-induct′*)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res a*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next case (*NonD-L x y*) **then show** *?case* **by** (*simp add*: *normalised-map*)
next case (*NonD-R x y*) **then show** *?case* **by** (*simp add*: *normalised-map*)
next case (*NonD x y*) **then show** *?case* **by** *simp*
next case (*Executable-L x y*) **then show** *?case* **by** (*simp add*: *normalised-map*)
next case (*Executable-R x y*) **then show** *?case* **by** (*simp add*: *normalised-map*)
next case (*Executable x y*) **then show** *?case* **by** *simp*
next case (*Repeatable-L x y*) **then show** *?case* **by** (*simp add*: *normalised-map*)
next case (*Repeatable-R x y*) **then show** *?case* **by** (*simp add*: *normalised-map*)
next case (*Repeatable x y*) **then show** *?case* **by** *simp*
next
case (*Par-Norm xs*)
then show *?case*
by (*fastforce simp add*: *Bex-set*[*symmetric*] *normalised-map*)
next
case (*Par-Par xs*)
then show *?case*
by (*fastforce simp add*: *Bex-set*[*symmetric*] *normalised-map merge-one-parallel-map*)
next
case (*Par-Empty xs*)
then show *?case*
by (*simp add*: *Bex-set*[*symmetric*] *normalised-map remove-one-empty-map*)
(*metis Ball-set*)
next case *Par-Nil* **then show** *?case* **by** *simp*
next case (*Par-Single u*) **then show** *?case* **by** (*simp add*: *normalised-map*)
next
case (*Par v vb vc*)
then show *?case*
by (*fastforce simp add*: *Bex-set*[*symmetric*] *Ball-set normalised-map*)
qed

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-term-rewrite-bound* (*step* x) < *res-term-rewrite-bound* x
proof (*induct x rule*: *step-induct'*)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res a*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next case (*NonD-L x y*) **then show** *?case* **by** *simp*
next case (*NonD-R x y*) **then show** *?case* **by** *simp*
next case (*NonD x y*) **then show** *?case* **by** *simp*
next case (*Executable-L x y*) **then show** *?case* **by** *simp*
next case (*Executable-R x y*) **then show** *?case* **by** *simp*
next case (*Executable x y*) **then show** *?case* **by** *simp*
next case (*Repeatable-L x y*) **then show** *?case* **by** *simp*
next case (*Repeatable-R x y*) **then show** *?case* **by** *simp*
next case (*Repeatable x y*) **then show** *?case* **by** *simp*
next
case (*Par-Norm xs*)
then have ($\sum x \leftarrow xs$. *res-term-rewrite-bound* (*step x*)) < *sum-list* (*map res-term-rewrite-bound xs*)
by (*meson res-term-rewrite-bound-step-non-increase sum-list-mono-one-strict*)
then show *?case*
using *Par-Norm.hyps* **by** (*simp add*: *comp-def*)
next
case (*Par-Par xs*)
then show *?case*
by (*fastforce elim*: *obtain-first-parallel simp add*: *merge-one-parallel-append*)
next
case (*Par-Empty xs*)
then show *?case*
by (*fastforce elim*: *obtain-first-empty simp add*: *remove-one-empty-append*)
next case *Par-Nil* **then show** *?case* **by** *simp*
next case (*Par-Single u*) **then show** *?case* **by** *simp*
next case (*Par v vb vc*) **then show** *?case* **using** *normalised-is-step-id* **by** *blast*
qed

4.4 Normalisation Procedure

Rewrite a resource term until normalised

function *normal-rewr* :: ('*a*, '*b*) *res-term* ⇒ ('*a*, '*b*) *res-term*

where *normal-rewr x* = (*if normalised x then x else normal-rewr* (*step x*))
by *pat-completeness auto*

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

termination *normal-rewr*
using *res-term-rewrite-bound-step-decrease*
by (*relation Wellfounded.measure res-term-rewrite-bound*, *auto*)

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*
by (*simp add*: *normal-rewr.simps*)

Normalisation produces actually normalised terms

lemma *normal-rewr-normalised*:
normalised (*normal-rewr x*)
by (*induct x rule*: *normal-rewr.induct*, *simp add*: *normal-rewr.simps*)

Normalisation is idempotent

lemma *normal-rewr-idempotent* [*simp*]:
normal-rewr (*normal-rewr x*) = *normal-rewr x*
using *normal-rewr-normalised normalised-normal-rewr* **by** *blast*

Normalisation absorbs rewriting step

lemma *normal-rewr-step*:
normal-rewr x = *normal-rewr* (*step x*)
by (*cases normalised x*) (*simp-all add*: *normal-rewr.simps*)

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*
by *simp-all*

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

lemma *normal-rewr-nondet*:
normal-rewr (*NonD x y*) = *NonD* (*normal-rewr x*) (*normal-rewr y*)
proof (*induct x rule*: *normal-rewr.induct*)
case *x*: (*1 x*)
then show *?case*

proof (*induct y rule*: *normal-rewr.induct*)
case *y*: (*1 y*)
then show *?case*
by (*metis normal-rewr-step normalised.simps*(*6*) *normalised-normal-rewr step.simps*(*5*))
qed
qed
lemma *normal-rewr-executable*:
normal-rewr (*Executable x y*) = *Executable* (*normal-rewr x*) (*normal-rewr y*)
proof (*induct x rule*: *normal-rewr.induct*)
case *x*: (*1 x*)
then show *?case*
proof (*induct y rule*: *normal-rewr.induct*)
case *y*: (*1 y*)
then show *?case*
by (*metis normal-rewr-step normalised.simps*(*7*) *normalised-normal-rewr step.simps*(*6*))
qed
qed
lemma *normal-rewr-repeatable*:
normal-rewr (*Repeatable x y*) = *Repeatable* (*normal-rewr x*) (*normal-rewr y*)
proof (*induct x rule*: *normal-rewr.induct*)
case *x*: (*1 x*)
then show *?case*
proof (*induct y rule*: *normal-rewr.induct*)
case *y*: (*1 y*)
then show *?case*
by (*metis normal-rewr-step normalised.simps*(*8*) *normalised-normal-rewr step.simps*(*7*))
qed
qed

Normalisation simplifies empty *Parallel* terms

lemma *normal-rewr-parallel-empty*:
normal-rewr (*Parallel* []) = *Empty*
by (*simp add*: *normal-rewr.simps*)

Every resource is equivalent to its normalisation

lemma *res-term-equiv-normal-rewr*:
$x \sim$ *normal-rewr x*
proof (*induct x rule*: *normal-rewr.induct*)
case (*1 x*)
then show *?case*
proof (*cases normalised x*)
case *True*
then show *?thesis* **by** (*simp add*: *normal-rewr.simps*)
next
case *False*
then have *step x* $\sim$ *normal-rewr* (*step x*)

using *1* **by** *simp*
then have $x \sim$ *normal-rewr* (*step x*)
using *res-term-equiv.trans res-term-equiv-step* **by** *blast*
then show *?thesis*
by (*simp add*: *normal-rewr.simps*)
qed
qed

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

lemma *normal-rewr-imp-equiv*:
normal-rewr x = *normal-rewr y* $\Longrightarrow x \sim y$
using *res-term-equiv-normal-rewr*[*of x*] *res-term-equiv-normal-rewr*[*of y*, *symmetric*]
by (*metis res-term-equiv.trans*)

Resource normalisation commutes with the resource map

lemma *normal-rewr-map*:
map-res-term f g (*normal-rewr x*) = *normal-rewr* (*map-res-term f g x*)
proof (*induct x rule*: *normal-rewr.induct*)
case (*1 x*)
then show *?case*
proof (*cases normalised x*)
case *True*
then show *?thesis*
by (*simp add*: *normalised-map normal-rewr.simps*)
next
case *False*
have *map-res-term f g* (*normal-rewr x*) = *map-res-term f g* (*normal-rewr* (*step x*))
using *False* **by** (*simp add*: *normal-rewr.simps*)
also have ... = *normal-rewr* (*map-res-term f g* (*step x*))
using *1 False* **by** *simp*
also have ... = *normal-rewr* (*step* (*map-res-term f g x*))
using *step-map*[*of f g x*] **by** *simp*
also have ... = *normal-rewr* (*map-res-term f g x*)
using *False* **by** (*simp add*: *normalised-map normal-rewr.simps*)
finally show *?thesis* **.**
qed
qed

Normalisation is contained in transitive closure of the rewriting

lemma *res-term-rewrite-tranclp-normal-rewr*:
res-term-rewrite^{++} *x* (*normal-rewr x*)
proof (*induct x rule*: *normal-rewr.induct*)
case (*1 x*)
then show *?case*
proof (*cases normalised x*)
case *True*
then show *?thesis*

by (*simp add*: *tranclp.r-into-trancl*)
next
case *False*
then show *?thesis*
using *1 res-term-rewrite-contains-step tranclp-into-tranclp2 normal-rewr-step* **by** *metis*
qed
qed

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

lemma *step-irr-inI*:
$x \neq$ *step* $x \Longrightarrow (x,$ *step* $x) \in$ *step-irr*
by (*simp add*: *step-irr-def*)

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

lemma *normal-rewr-in-step-rtrancl*:
$(x,$ *normal-rewr* $x) \in$ *step-irr**
proof (*induct x rule*: *normal-rewr.induct*)
case (*1 x*)
then show *?case*
proof (*cases normalised x*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
moreover have $(x,$ *step* $x) \in$ *step-irr*
using *False normalised-is-step-id* **by** (*fastforce simp add*: *step-irr-def*)
ultimately show *?thesis*
by (*metis 1 converse-rtrancl-into-rtrancl normal-rewr.elims*)
qed
qed

Normal forms of irreflexive step are exactly the normalised terms

lemma *step-nf-is-normalised*:
  NF step-irr = {*x*. *normalised x*}
proof *safe*
  fix *x* :: (′*a*, ′*b*) *res-term*
  show *x* ∈ *NF step-irr* ⟹ *normalised x*
    by (*metis NF-not-suc normal-rewr-in-step-rtrancl normal-rewr-normalised*)
  show *normalised x* ⟹ *x* ∈ *NF step-irr*
    by (*simp add*: *NF-I step-irr-def*)
qed

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*
  by (*simp add*: *normal-rewr-normalised step-nf-is-normalised*)

Terms related by reflexive-transitive step are equivalent

lemma *step-rtrancl-equivalent*:
  $(a,b) \in$ *step-irr** ⟹ $a \sim b$
proof (*induct rule*: *rtrancl-induct*)
  case *base*
  then show *?case* **by** *simp*
next
  case (*step y z*)
  then show *?case*
   by (*metis* (*mono-tags*, *lifting*) *Product-Type.Collect-case-prodD fst-conv res-term-equiv.refl res-term-equiv.trans-both snd-conv res-term-equiv-step step-irr-def*)
qed

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

lemma *step-irr-locally-confluent*:
  WCR step-irr
  unfolding *step-irr-def* **by** *standard fastforce*

lemma *step-irr-strongly-confluent*:
  strongly-confluent step-irr
  unfolding *step-irr-def* **by** *standard fastforce*

Therefore it is Church-Rosser and has unique normal forms

lemma *step-CR*: *CR step-irr*
 using *step-irr-strongly-confluent strong-confluence-imp-CR CR-imp-UNC CR-imp-UNF*
by *blast*
lemma *step-UNC*: *UNC step-irr*
  using *step-CR CR-imp-UNC* **by** *blast*
lemma *step-UNF*: *UNF step-irr*
  using *step-CR CR-imp-UNF* **by** *blast*

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

lemma *step-SN*:
SN step-irr
unfolding *SN-def*
using *SN-onI*
proof
fix x :: $('a, 'b)$ *res-term* **and** f
show $[\![f\ 0 \in \{x\}; \forall i.\ (f\ i,\ f\ (Suc\ i)) \in step\text{-}irr]\!] \Longrightarrow False$
— Irreflexivity of step is essential here to get the needed contradiction
— Strong induction is needed because bound may decrease by more than 1
proof (*induct res-term-rewrite-bound x arbitrary*: *f x rule*: *less-induct*)
case *less*
then show *?case*
using *less*(*1*)[**where** $x = step\ x$ **and** $f = \lambda x.\ f\ (Suc\ x)$]
by (*metis* (*mono-tags*, *lifting*) *case-prodD mem-Collect-eq normalised-is-step-id res-term-rewrite-bound-step-decrease singleton-iff step-irr-def*)
qed
qed

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

lemma *step-normalizability-normal-rewr*:
$step\text{-}irr^! = \{(x,\ y).\ y = normal\text{-}rewr\ x\}$
proof *safe*
fix $a\ b$ :: $('a, 'b)$ *res-term*
assume $(a,\ b) \in step\text{-}irr^!$
then show $b = normal\text{-}rewr\ a$
by (*meson UNF-onE UNIV-I normal-rewr-NF normal-rewr-in-step-rtrancl normalizability-I step-UNF*)
next
fix a :: $('a, 'b)$ *res-term*
show $(a,\ normal\text{-}rewr\ a) \in step\text{-}irr^!$
using *normal-rewr-NF normal-rewr-in-step-rtrancl* **by** *blast*
qed

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*
by (*meson UNF-onE UNIV-I normal-rewr-NF normal-rewr-in-step-rtrancl normalizability-I step-CR step-SN step-UNF the-NF*)

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

lemma *step-rtrancl-eq-normal*:
$(x,y) \in step\text{-}irr^* \Longrightarrow normal\text{-}rewr\ x = normal\text{-}rewr\ y$
by (*metis normal-rewr-NF normal-rewr-in-step-rtrancl rtrancl-trans some-NF-UNF step-UNF*)

4.5.2 *NonD* Joinability

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

lemma *step-rtrancl-nondL*:
$(x,u) \in$ *step-irr*$^* \Longrightarrow$ (*NonD x y*, *NonD u y*) $\in$ *step-irr**
proof (*induct rule*: *rtrancl-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step y z*)
then show *?case*
by (*fastforce intro*: *rtrancl-into-rtrancl simp add*: *step-irr-def*)
qed

lemma *step-rtrancl-nondR*:
$[\![$(*y*,*v*) $\in$ *step-irr**; *normalised x*$]\!] \Longrightarrow$ (*NonD x y*, *NonD x v*) $\in$ *step-irr**
proof (*induct rule*: *rtrancl-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step y z*)
then show *?case*
by (*fastforce intro*: *rtrancl-into-rtrancl simp add*: *step-irr-def*)
qed

lemma *step-rtrancl-nond*:
$[\![$(*x*,*u*) $\in$ *step-irr**; *normalised u*; (*y*,*v*) $\in$ *step-irr*$^*]\!] \Longrightarrow$ (*NonD x y*, *NonD u v*) $\in$ *step-irr**
using *step-rtrancl-nondL step-rtrancl-nondR* **by** (*metis rtrancl-trans*)

lemma *step-join-apply-nondet*:
assumes (*x*,*u*) $\in$ *step-irr*$^\downarrow$ **and** (*y*,*v*) $\in$ *step-irr*$^\downarrow$ **shows** (*NonD x y*, *NonD u v*) $\in$ *step-irr*$^\downarrow$
proof (*rule joinI*)
have (*NonD x y*, *NonD* (*normal-rewr x*) *y*) $\in$ *step-irr**
using *step-rtrancl-nondL normal-rewr-in-step-rtrancl* **by** *metis*
also have (*NonD* (*normal-rewr x*) *y*, *NonD* (*normal-rewr x*) (*normal-rewr y*)) $\in$ *step-irr**
using *step-rtrancl-nondR normal-rewr-in-step-rtrancl normal-rewr-normalised* **by** *metis*
finally show (*NonD x y*, *NonD* (*normal-rewr x*) (*normal-rewr y*)) $\in$ *step-irr** **.**

have (*NonD u v*, *NonD* (*normal-rewr u*) *v*) $\in$ *step-irr**
using *step-rtrancl-nondL normal-rewr-in-step-rtrancl* **by** *metis*
also have (*NonD* (*normal-rewr u*) *v*, *NonD* (*normal-rewr u*) (*normal-rewr v*)) $\in$ *step-irr**
using *step-rtrancl-nondR normal-rewr-in-step-rtrancl normal-rewr-normalised* **by** *metis*
also have
(*NonD* (*normal-rewr u*) (*normal-rewr v*), *NonD* (*normal-rewr x*) (*normal-rewr y*)) $\in$ *step-irr**

using *assms joinD step-rtrancl-eq-normal rtrancl.rtrancl-refl* **by** *metis*
finally show $(NonD\ u\ v,\ NonD\ (normal\text{-}rewr\ x)\ (normal\text{-}rewr\ y)) \in step\text{-}irr^*$ **.**
qed

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*:
$[\![(x,u) \in step\text{-}irr^{\downarrow};\ (y,v) \in step\text{-}irr^{\downarrow}]\!] \Longrightarrow (Executable\ x\ y,\ Executable\ u\ v) \in step\text{-}irr^{\downarrow}$
using *joinI*[**where** *c* = *Executable* (*normal-rewr x*) (*normal-rewr y*)] *normal-rewr-executable*
by (*metis* (*mono-tags*, *lifting*) *joinD normal-rewr-in-step-rtrancl step-rtrancl-eq-normal*)

lemma *step-join-apply-repeatable*:
$[\![(x,u) \in step\text{-}irr^{\downarrow};\ (y,v) \in step\text{-}irr^{\downarrow}]\!] \Longrightarrow (Repeatable\ x\ y,\ Repeatable\ u\ v) \in step\text{-}irr^{\downarrow}$
using *joinI*[**where** *c* = *Repeatable* (*normal-rewr x*) (*normal-rewr y*)] *normal-rewr-repeatable*
by (*metis* (*mono-tags*, *lifting*) *joinD normal-rewr-in-step-rtrancl step-rtrancl-eq-normal*)

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 *list-all2* $(\lambda x\ y.\ (x,\ y) \in R^{\downarrow})$ *xs 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*
using *assms* **by** (*induct rule*: *list-all2-induct* ; *blast*)

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*:
$(Parallel\ xs,\ Parallel\ (map\ normal\text{-}rewr\ xs)) \in step\text{-}irr^*$
proof (*induct sum-list* (*map res-term-rewrite-bound xs*) *arbitrary*: *xs rule*: *less-induct*)
case *less*
then show *?case*
proof (*cases list-all normalised xs*)
case *True*
then show *?thesis*
by (*metis Ball-set map-idI normalised-normal-rewr rtrancl.rtrancl-refl*)
next
case *False*
then have *unnorm*: ¬ *normalised* (*Parallel xs*)
by *simp*

have *step*: *step* (*Parallel xs*) = *Parallel* (*map step xs*)

using *False* **by** (*simp add*: *not-list-all*)
moreover have *Parallel xs* $\neq$ *Parallel* (*map step xs*)
using *unnorm* **by** (*metis calculation normalised-is-step-id*)
ultimately have (*Parallel xs*, *Parallel* (*map step xs*)) $\in$ *step-irr*
using *step-irr-inI* **by** *metis*
moreover have (*Parallel* (*map step xs*), *Parallel* (*map normal-rewr* (*map step xs*))) $\in$ *step-irr**
using *less*[*of map step xs*] *False step unnorm*
by (*smt* (*verit*, *ccfv-threshold*) *ab-semigroup-add-class.add-ac*(*1*)
add-mono-thms-linordered-field(*3*) *dual-order.refl length-map not-less-eq plus-1-eq-Suc*
res-term-rewrite-bound.simps(*5*) *res-term-rewrite-bound-step-decrease*)
moreover have *map normal-rewr* (*map step xs*) = *map normal-rewr xs*
by (*simp* ; *safe* ; *rule normal-rewr-step*[*symmetric*])
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*) *converse-rtrancl-into-rtrancl*)
qed
qed

Two lists of joinable terms have the same normal forms

lemma *list-all2-join-normal-eq*:
list-all2 (λ*u v.* (*u*, *v*) $\in$ *step-irr*$^\downarrow$) *xs ys* $\Longrightarrow$ *map normal-rewr xs* = *map normal-rewr ys*
proof (*induct rule*: *list-all2-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs y ys*)
then show *?case* **by** *simp* (*metis* (*no-types*, *lifting*) *joinD step-rtrancl-eq-normal*)
qed

Parallel resource terms whose contents are joinable are themselves joinable

lemma *step-join-apply-parallel*:
assumes *list-all2* (λ*u v.* (*u*,*v*) $\in$ *step-irr*$^\downarrow$) *xs ys*
shows (*Parallel xs*, *Parallel ys*) $\in$ *step-irr*$^\downarrow$
by (*metis assms joinI list-all2-join-normal-eq step-rtrancl-map-normal*)

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 (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
by (*cases a* ; *fastforce*)
qed

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 (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case*
    by (*cases a* ; *simp*)
qed

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**
  using *assms*
proof (*induct count-if is-Empty xs arbitrary*: *xs rule*: *less-induct*)
  case *less*
  then show *?case*
  proof (*cases list-ex is-Empty xs*)
    case *True*
    then have *a*: *step* (*Parallel xs*) = *Parallel* (*remove-one-empty xs*)
      using *less* **by** (*metis Bex-set step.simps*(*8*))
    moreover have *b*: *count-if is-Empty* (*remove-one-empty xs*) < *count-if is-Empty xs*
      using *True* **by** (*rule remove-one-empty-count-if-decrease*)
    moreover have *c*: $\bigwedge$*x*. *x* $\in$ *set* (*remove-one-empty xs*) $\Longrightarrow$ *normalised x*
      using *remove-one-empty-subset less*(*2*) **by** *fast*
    moreover have $\neg$ *list-ex is-Parallel* (*remove-one-empty xs*)
      using *remove-one-empty-subset less*(*3*) *not-list-ex*
      by (*metis* (*mono-tags*, *lifting*) *Ball-set*)
    ultimately show *?thesis*
      using *less remove-all-empty-subsumes-remove-one*
      by (*metis converse-rtrancl-into-rtrancl step-irr-inI*)
  next
    case *False*
    then show *?thesis*
      by (*simp add*: *joinI-right remove-all-empty-none*)
  qed
qed

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*)
using *assms merge-all-parallel-result normalised.simps*(*5*) *not-list-ex* **by** *blast*

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-if is-Parallel* (*merge-one-parallel xs*) < *count-if is-Parallel xs*
using *assms*
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case* **by** (*cases a*) (*simp-all add*: *count-if-0-conv*)
qed

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-all-parallel* (*merge-one-parallel xs*) = *merge-all-parallel xs*
using *assms*
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
proof (*cases a*)
case *Empty* **then show** *?thesis* **using** *Cons* **by** *simp*
next case *Anything* **then show** *?thesis* **using** *Cons* **by** *simp*
next case (*Res x3*) **then show** *?thesis* **using** *Cons* **by** *simp*
next case (*Copyable x4*) **then show** *?thesis* **using** *Cons* **by** *simp*
next
case (*Parallel x5*)
then show *?thesis*
using *Cons* **by** (*simp add*: *merge-all-parallel-append merge-all-parallel-none*)
next case (*NonD x61 x62*) **then show** *?thesis* **using** *Cons* **by** *simp*
next case (*Executable x71 x72*) **then show** *?thesis* **using** *Cons* **by** *simp*
next case (*Repeatable x81 x82*) **then show** *?thesis* **using** *Cons* **by** *simp*
qed
qed

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

lemma *merge-one-parallel-preserve-normalised*:

$\llbracket\bigwedge x.\ x \in set\ xs \Longrightarrow normalised\ x;\ a \in set\ (merge\text{-}one\text{-}parallel\ xs)\rrbracket \Longrightarrow normalised\ a$
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases a* ; *simp* ; (*presburger* | *metis normalised-parallel-children*))
qed

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

lemma *merge-all-parallel-preserve-normalised*:
  $\llbracket\bigwedge x.\ x \in set\ xs \Longrightarrow normalised\ x;\ a \in set\ (merge\text{-}all\text{-}parallel\ xs)\rrbracket \Longrightarrow normalised\ a$
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case* **by** (*cases a* ; *simp* ; (*presburger* | *metis normalised-parallel-children*))
qed

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 $(Parallel\ xs,\ Parallel\ (merge\text{-}all\text{-}parallel\ xs)) \in step\text{-}irr^*$
  using *assms*
proof (*induct count-if is-Parallel xs arbitrary*: *xs rule*: *less-induct*)
  case *less*
  then show *?case*
  proof (*cases list-ex is-Parallel xs*)
    case *False*
    then show *?thesis*
      using *merge-all-parallel-none* **by** (*metis rtrancl.rtrancl-refl*)
  next
    case *True*
    then have $step\ (Parallel\ xs) = Parallel\ (merge\text{-}one\text{-}parallel\ xs)$
      using *less* **by** (*metis Bex-set step.simps*(*8*))
    moreover have $\bigwedge x.\ x \in set\ (merge\text{-}one\text{-}parallel\ xs) \Longrightarrow normalised\ x$
      using *merge-one-parallel-preserve-normalised less*(*2*) **by** *blast*
    moreover have $count\text{-}if\ is\text{-}Parallel\ (merge\text{-}one\text{-}parallel\ xs) < count\text{-}if\ is\text{-}Parallel\ xs$
      using *less*(*2*) *True merge-one-parallel-count-if-decrease normalised.simps*(*5*) *not-list-ex*
      by *blast*
    ultimately show *?thesis*

using *less merge-all-parallel-subsumes-merge-one*
by (*metis converse-rtrancl-into-rtrancl normalised.simps*(*5*) *not-list-ex step-irr-inI*)
qed
qed

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

lemma *step-rtrancl-parallel*:
(*Parallel xs*, *Parallel* (*remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*)))) $\in$ *step-irr**
proof −
have (*Parallel xs*, *Parallel* (*map normal-rewr xs*)) $\in$ *step-irr**
by (*rule step-rtrancl-map-normal*)
also have
(*Parallel* (*map normal-rewr xs*), *Parallel* (*merge-all-parallel* (*map normal-rewr xs*)))
$\in$ *step-irr**
by (*metis ex-map-conv normal-rewr-normalised step-rtrancl-merge-all-parallel*)
also have (*Parallel* (*merge-all-parallel* (*map normal-rewr xs*)),
Parallel (*remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*))))
$\in$ *step-irr**
using *merge-all-parallel-map-normal-result merge-all-parallel-preserve-normalised normal-rewr-normalised step-rtrancl-remove-all-empty*
by (*metis* (*mono-tags*, *lifting*) *imageE list.set-map*)
finally show *?thesis* **.**
qed

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 −
obtain a **where** a: (x, a) $\in R^*$ (y, a) $\in R^*$
using *assms*(*3*) *joinE* **by** *metis*
then have *the-NF R y* = *the-NF R a*
using *assms*(*1*,*2*) *the-NF-steps* **by** *metis*
moreover obtain b **where** b: (y, b) $\in R^*$ (z, b) $\in R^*$
using *assms*(*4*) *joinE* **by** *metis*
then have *the-NF R y* = *the-NF R b*
using *assms*(*1*,*2*) *the-NF-steps* **by** *metis*
ultimately show *?thesis*
using *assms*(*1*,*2*) a b **by** (*meson CR-join-right-I joinI join-rtrancl-join*)
qed

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

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

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 (*cases xs rule*: *remdups-adj.cases*)
  case *1*
  then show *?thesis* **using** *that normal-rewr-parallel-empty* **by** *fastforce*
next
  case (*2 x*)
  then have *normal-rewr* (*Parallel* [*x*]) = *step* (*Parallel* [*x*])
    using *assms* **by** (*subst normal-rewr.simps*) *simp*
  then show *?thesis*
    using *that assms 2* **by** *simp*
next
  case (*3 x y xs*)
  then show *?thesis*
    using *assms that*
    by (*metis normal-rewr.simps normalised-parallelise parallelise.simps*(*3*))
qed

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 −
  show *?thesis*
    using *assms*
  proof (*cases rule*: *normal-rewr-parallel-cases*)

case *Parallel*
then show *?thesis*
using *parallel-parts-no-empty-parallel assms*
by (*metis list-obtain-2 normalised.simps*(*5*) *parallelise.simps*(*3*))
next case *Empty* **then show** *?thesis* **by** *simp*
next case (*Single a*) **then show** *?thesis* **using** *assms* **by** (*cases a* ; *simp*)
qed
qed

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*
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next case (*Cons a xs*) **then show** *?case* **by** (*cases a*) *simp-all*
qed

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

lemma *remove-all-empty-idempotent*:
shows *remove-all-empty* (*remove-all-empty xs*) = *remove-all-empty xs*
by (*induct xs*) *simp-all*

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*:
normal-rewr (*Parallel xs*)
= *parallelise* (*remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*)))
proof −
have
normal-rewr (*Parallel xs*)
= *normal-rewr* (*Parallel* (*remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*))))
using *step-rtrancl-parallel step-rtrancl-eq-normal* **by** *metis*
also have ...
= *parallelise* (*parallel-parts* (*Parallel*
(*remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*)))))
using *merge-all-parallel-preserve-normalised normal-rewr-parallelise parallel-remove-all-empty*
using *merge-all-parallel-map-normal-result remove-all-empty-result normal-rewr-normalised*
by (*smt* (*verit*, *ccfv-threshold*) *imageE list.set-map remove-all-empty-subset*)
also have ... = *parallelise* (*remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*)))
using *parallel-parts-no-empty-parallel parallel-remove-all-empty*
using *merge-all-parallel-map-normal-result remove-all-empty-result normal-rewr-normalised*
by (*metis* (*mono-tags*, *lifting*) *imageE list.set-map*)
finally show *?thesis* **.**
qed

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 *normal-rewr* (*Parallel xs*) = *Empty*
    shows *remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*)) = []
  using *assms normal-rewr-to-parallelise parallelise-to-empty-eq remove-all-empty-result*
  by (*metis list-ex-simps*(*1*) *res-term.disc*(*19*))

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 *normal-rewr* (*Parallel xs*) = *Parallel ys*
   shows *remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*)) = *remove-all-empty ys*
proof −
  have ¬ *list-ex is-Parallel* (*remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*)))
    using *merge-all-parallel-map-normal-result normal-rewr-normalised parallel-remove-all-empty*
    by (*metis* (*mono-tags*, *lifting*) *imageE list.set-map*)
  then have *remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*)) = *ys*
    by (*metis assms normal-rewr-to-parallelise normal-rewr-normalised normalised-parallel-parts-eq parallel-parts-no-empty-parallel parallel-parts-parallelise-eq remove-all-empty-result*)
  then show *?thesis*
    using *assms remove-all-empty-idempotent* **by** *metis*
qed

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 *normal-rewr* (*Parallel xs*) = *a*
      and ¬ *is-Empty a*
      and ¬ *is-Parallel a*
    shows *remove-all-empty* (*merge-all-parallel* (*map normal-rewr xs*)) = [*a*]
  using *assms* **by** (*simp add*: *normal-rewr-to-parallelise parallelise-to-single-eq*)

4.5.6 Equivalent Term Joinability

Equivalent resource terms are joinable by irreflexive step

lemma *res-term-equiv-joinable*:
  $x \sim y \Longrightarrow (x,\ y) \in \mathit{step\text{-}irr}^{\downarrow}$
proof (*induct rule*: *res-term-equiv.induct*)
    case *empty* **then show** *?case* **by** *blast*
next case *anything* **then show** *?case* **by** *blast*
next case (*res x*) **then show** *?case* **by** *blast*
next case (*copyable x*) **then show** *?case* **by** *blast*
next

```
  case nil
  then show ?case
    by (metis joinI-left normal-rewr-in-step-rtrancl normal-rewr-parallel-empty)
next
  case (singleton a)
  then show ?case
  proof (induct res-term-rewrite-bound a arbitrary: a rule: less-induct)
    case less
      then show ?case
    proof (cases normalised a)
      case True
      then show ?thesis
      proof (cases a)
        case Empty
        moreover have (Parallel [Empty], Empty) ∈ step-irr*
        proof –
          have step (Parallel [Empty]) = Parallel []
            by simp
          then show ?thesis
            using normal-rewr-in-step-rtrancl normal-rewr-parallel-empty
            by (metis converse-rtrancl-into-rtrancl step-irr-inI)
        qed
        ultimately show ?thesis
          using joinI-left by simp
      next
        case Anything
        then have step (Parallel [a]) = a
          by simp
        then show ?thesis
          using step-irr-inI parallel-neq-single r-into-rtrancl joinI-left by metis
      next
        case (Res x3)
        then have step (Parallel [a]) = a
          by simp
        then show ?thesis
          using step-irr-inI parallel-neq-single r-into-rtrancl joinI-left by metis
      next
        case (Copyable x4)
        then have step (Parallel [a]) = a
          using True by simp
        then show ?thesis
          using step-irr-inI parallel-neq-single r-into-rtrancl joinI-left by metis
      next
        case (Parallel x5)
        then have step (Parallel [Parallel x5]) = Parallel x5
          using True by simp
        then show ?thesis
           using step-irr-inI parallel-neq-single r-into-rtrancl joinI-left Parallel by
metis
```

```
      next
        case (NonD x61 x62)
        then have step (Parallel [a]) = a
          using True by simp
        then show ?thesis
          using step-irr-inI parallel-neq-single r-into-rtrancl joinI-left by metis
      next
        case (Executable x71 x72)
        then have step (Parallel [a]) = a
          using True by simp
        then show ?thesis
          using step-irr-inI parallel-neq-single r-into-rtrancl joinI-left by metis
      next
        case (Repeatable x71 x72)
        then have step (Parallel [a]) = a
          using True by simp
        then show ?thesis
          using step-irr-inI parallel-neq-single r-into-rtrancl joinI-left by metis
      qed
    next
      case False
      then have step (Parallel [a]) = Parallel [step a]
        by simp
      moreover have res-term-rewrite-bound (step a) < res-term-rewrite-bound a
        using res-term-rewrite-bound-step-decrease False by blast
      ultimately show ?thesis
        using less normal-rewr-in-step-rtrancl step-irr-join-trans step-normalised
        by (metis joinI normal-rewr.elims)
    qed
  qed
next
  case (merge x y z)
  have
    ( Parallel (x @ y @ z)
     , Parallel (remove-all-empty (merge-all-parallel (map normal-rewr (x @ y @
z))))
     ) ∈ step-irr*
    using step-rtrancl-parallel .
  also have
     ( Parallel (remove-all-empty (merge-all-parallel (map normal-rewr (x @ y @
z))))
        , Parallel (remove-all-empty (merge-all-parallel (map normal-rewr (x @
[Parallel y] @ z))))
     ) ∈ step-irr*
  proof (cases normal-rewr (Parallel y))
    case Empty
    then show ?thesis
     by (simp add: merge-all-parallel-append remove-all-empty-append normal-rewr-to-empty)
  next
```

case *Anything*
then show *?thesis*
by (*simp add*: *merge-all-parallel-append remove-all-empty-append normal-rewr-to-other*)
next
case (*Res x3*)
then show *?thesis*
by (*simp add*: *merge-all-parallel-append remove-all-empty-append normal-rewr-to-other*)
next
case (*Copyable x4*)
then show *?thesis*
by (*simp add*: *merge-all-parallel-append remove-all-empty-append normal-rewr-to-other*)
next
case (*Parallel x5*)
then show *?thesis*
by (*simp add*: *merge-all-parallel-append remove-all-empty-append normal-rewr-to-parallel*)
next
case (*NonD x61 x62*)
then show *?thesis*
by (*simp add*: *merge-all-parallel-append remove-all-empty-append normal-rewr-to-other*)
next
case (*Executable x71 x72*)
then show *?thesis*
by (*simp add*: *merge-all-parallel-append remove-all-empty-append normal-rewr-to-other*)
next
case (*Repeatable x81 x82*)
then show *?thesis*
by (*simp add*: *merge-all-parallel-append remove-all-empty-append normal-rewr-to-other*)
qed
finally show *?case*
using *step-rtrancl-parallel* **by** *blast*
next
case (*parallel xs ys*)
then show *?case*
by (*simp add*: *list-all2-mono step-join-apply-parallel*)
next
case (*nondet x y u v*)
then show *?case* **using** *step-join-apply-nondet* **by** *blast*
next
case (*executable x y u v*)
then show *?case* **using** *step-join-apply-executable* **by** *blast*
next
case (*repeatable x y u v*)
then show *?case* **using** *step-join-apply-repeatable* **by** *blast*
next
case (*sym x y*)
then show *?case* **by** (*simp add*: *join-sym*)
next
case (*trans x y z*)
then show *?case* **by** (*meson joinE CR-join-right-I joinI join-rtrancl-join step-CR*)

qed

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-rewr* x = *normal-rewr* y
proof (*rule join-NF-imp-eq*)
  have *normal-rewr* $x \sim x$
    using *res-term-equiv-normal-rewr res-term-equiv.sym* **by** *blast*
  moreover have $y \sim$ *normal-rewr* y
    by (*rule res-term-equiv-normal-rewr*)
  ultimately have *normal-rewr* $x \sim$ *normal-rewr* y
    using *assms* **by** (*rule res-term-equiv.trans-both*)
  then show (*normal-rewr* x, *normal-rewr* y) $\in$ *step-irr*$^{\downarrow}$
    by (*rule res-term-equiv-joinable*)

  show *normal-rewr* $x \in$ *NF step-irr*
   and *normal-rewr* $y \in$ *NF step-irr*
    by (*rule normal-rewr-NF*)+
qed

And resource term equivalence is equal to having equal normal forms

theorem *res-term-equiv-is-normal-rewr*:
  $x \sim y$ = (*normal-rewr* x = *normal-rewr* y)
  using *res-term-equiv-imp-normal-rewr normal-rewr-imp-equiv* **by** *standard*

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-term-rewrite*)$^{++}$ x y

One direction is simpler, because rewriting implies equivalence

lemma *res-term-rewrite-equivclp-imp-equiv*:
  (*symclp res-term-rewrite*)$^{++}$ x $y \Longrightarrow x \sim y$
proof (*induct rule*: *tranclp.induct*)
  case (*r-into-trancl a b*)
  then show *?case*
    by (*metis symclp-def res-term-rewrite-imp-equiv res-term-equiv.sym*)
next
  case (*trancl-into-trancl a b c*)
  then have $b \sim c$
    by (*metis symclp-def res-term-rewrite-imp-equiv res-term-equiv.sym*)
  then show *?case*

by (*metis trancl-into-trancl*(*2*) *res-term-equiv.trans*)
qed

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

lemma
$x \sim y \Longrightarrow$ (*symclp res-term-rewrite*)$^{++}$ x y
proof (*induct x y rule*: *res-term-equiv.induct*)
case *empty* **then show** *?case* **by** (*simp add*: *tranclp.r-into-trancl*)
next case *anything* **then show** *?case* **by** (*simp add*: *tranclp.r-into-trancl*)
next case (*res x*) **then show** *?case* **by** (*simp add*: *tranclp.r-into-trancl*)
next case (*copyable x*) **then show** *?case* **by** (*simp add*: *tranclp.r-into-trancl*)
next
case *nil*
then show *?case*
by (*simp add*: *res-term-rewrite.nil tranclp.r-into-trancl*)
next
case (*singleton a*)
then show *?case*
by (*simp add*: *res-term-rewrite.singleton tranclp.r-into-trancl*)
next
case (*merge x y z*)
then show *?case*
by (*meson res-term-rewrite.merge symclp-def tranclp.r-into-trancl*)
next
case (*sym x y*)
then show *?case*
by (*metis rtranclpD rtranclp-symclp-sym tranclp-into-rtranclp*)
next case (*trans x y z*) **then show** *?case* **by** *simp*
next
case (*parallel xs ys*)
then show *?case*
— While we do know that corresponding parallel terms are related, the rewrite rule *list-all2 res-term-rewrite ?xs ?ys* $\Longrightarrow$ *res-term-rewrite* (*Parallel ?xs*) (*Parallel ?ys*) needs all rewrites to be in a uniform direction. Such an issue arises with all remaining cases.
oops

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-term-rewrite*)$^{++}$ x y
proof −
have *normal-rewr x* = *normal-rewr y*
using *assms res-term-equiv-is-normal-rewr* **by** *metis*
then have (*symclp res-term-rewrite*)$^{++}$ (*normal-rewr x*) (*normal-rewr y*)
by (*simp add*: *tranclp.r-into-trancl*)
moreover have (*symclp res-term-rewrite*)$^{++}$ x (*normal-rewr x*)
using *res-term-rewrite-tranclp-normal-rewr symclp-def res-term-rewrite.refl*

by (*metis equivclp-def rev-predicate2D rtranclp-into-tranclp2 rtranlcp-le-equivclp tranclp-into-rtranclp*)
moreover have (*symclp res-term-rewrite*)$^{++}$ (*normal-rewr y*) *y*
using *res-term-rewrite-tranclp-normal-rewr symclp-def res-term-rewrite.refl*
by (*metis conversepD equivclp-def rev-predicate2D rtranclpD rtranlcp-le-equivclp symp-conv-conversep-eq symp-rtranclp-symclp tranclp.r-into-trancl tranclp-into-rtranclp*)
ultimately show *?thesis*
by *simp*
qed

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

lemma *res-term-equiv-is-rewrite-closure*:
($\sim$) = *equivclp res-term-rewrite*
proof –
have *equivclp res-term-rewrite x y* = (*symclp res-term-rewrite*)$^{++}$ *x y*
for *x y* :: (*'a*, *'b*) *res-term*
by (*metis equivclp-def res-term-equiv.refl res-term-rewrite-equiv-imp-equivclp rtranclpD tranclp-into-rtranclp*)
then have *x* $\sim$ *y* = *equivclp res-term-rewrite x y*
for *x y* :: (*'a*, *'b*) *res-term*
using *res-term-rewrite-equivclp-imp-equiv res-term-rewrite-equiv-imp-equivclp* **by** *metis*
then show *?thesis*
by *blast*
qed

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-ex is-Empty ys*
  shows *merge-all-parallel* (*remove-all-empty xs*) = *remove-all-empty* (*merge-all-parallel xs*)
  using *assms*
proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case*
    by (*cases a*) (*simp-all add*: *remove-all-empty-append remove-all-empty-none*)
qed

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

Any resource term is equivalent to its direct normalisation

lemma *normal-dir-equiv*:
  $a \sim$ *normal-dir a*
proof (*induct a*)
    case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res x*) **then show** *?case* **by** *simp*
next case (*Copyable a*) **then show** *?case* **by** *simp*
next
  case (*Parallel xs*)
  then have *Parallel xs* $\sim$ *Parallel* (*map normal-dir xs*)
    by (*intro res-term-equiv.parallel*) (*simp add*: *list-all2-conv-all-nth*)
  also have ... $\sim$ *Parallel* (*remove-all-empty* (*map normal-dir xs*))
    by (*rule remove-all-empty-equiv*)
  also have ... $\sim$ *Parallel* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir xs*)))

by (*rule merge-all-parallel-equiv*)
finally show *?case*
using *parallelise-equiv res-term-equiv.trans res-term-equiv.sym* **by** *fastforce*
next case (*NonD a1 a2*) **then show** *?case* **by** (*simp add*: *res-term-equiv.nondet*)
next case (*Executable a1 a2*) **then show** *?case* **by** (*simp add*: *res-term-equiv.executable*)
next case (*Repeatable a1 a2*) **then show** *?case* **by** (*simp add*: *res-term-equiv.repeatable*)
qed

Thus terms with equal normalisation are equivalent

lemma *normal-dir-eq-imp-equiv*:
normal-dir a = normal-dir b $\Longrightarrow$ *a* $\sim$ *b*
using *normal-dir-equiv res-term-equiv.sym res-term-equiv.trans* **by** *metis*

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*
using *assms*
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
using *merge-all-parallel-result* **by** *blast*
qed

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*
using *assms*
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
using *remove-all-empty-subset* **by** *blast*
qed

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

lemma *normal-dir-no-nested-Parallel*:
normal-dir a = Parallel xs $\Longrightarrow$ $\neg$ *list-ex is-Parallel xs*
proof (*rule notI*, *induct a arbitrary*: *xs*)

case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res x*) **then show** *?case* **by** *simp*
next case (*Copyable a*) **then show** *?case* **by** *simp*
next
case (*Parallel x*)
then have *parallelise* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir x*))) = *Parallel xs*
by *simp*
then have *list-ex is-Parallel* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir x*)))
using *Parallel*(*3*) *ResTerm.parallelise-to-parallel-has-paralell* **by** *blast*
then obtain *ys*
where *Parallel ys* ∈ *set* (*remove-all-empty* (*map normal-dir x*))
and *ex-ys*: *list-ex is-Parallel ys*
by (*erule merge-all-parallel-has-Parallel*)
then have *Parallel ys* ∈ *set* (*map normal-dir x*)
using *remove-all-empty-has-Parallel* **by** *blast*
then show *?case*
using *Parallel*(*1*) *ex-ys* **by** *fastforce*
next case (*NonD a1 a2*) **then show** *?case* **by** *simp*
next case (*Executable a1 a2*) **then show** *?case* **by** *simp*
next case (*Repeatable a1 a2*) **then show** *?case* **by** *simp*
qed

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

lemma *normal-dir-no-nested-Empty*:
normal-dir a = *Parallel xs* $\Longrightarrow$ ¬ *list-ex is-Empty xs*
proof (*rule notI*, *induct a arbitrary*: *xs*)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res x*) **then show** *?case* **by** *simp*
next case (*Copyable a*) **then show** *?case* **by** *simp*
next
case (*Parallel x*)
then have *parallelise* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir x*))) = *Parallel xs*
by *simp*
then have *merge-all-parallel* (*remove-all-empty* (*map normal-dir x*)) = *xs*
proof (*elim parallelise-to-parallel-has-empty*)
assume *merge-all-parallel* (*remove-all-empty* (*map normal-dir x*)) = [*Parallel xs*]
then show *?thesis*
using *Parallel*(*3*) *merge-all-parallel-has-Parallel normal-dir-no-nested-Parallel remove-all-empty-has-Parallel*
by (*smt* (*verit*, *best*) *image-iff list.set-map list-ex-simps*(*1*) *res-term.discI*(*5*))
next
assume *merge-all-parallel* (*remove-all-empty* (*map normal-dir x*)) = *xs*

then show *?thesis* **.**
qed
then have *list-ex is-Empty* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir x*)))
using *Parallel*(*3*) **by** *blast*
then have *list-ex is-Empty* (*remove-all-empty* (*map normal-dir x*))
proof (*elim merge-all-parallel-has-empty*)
fix *ys*
assume *Parallel ys* $\in$ *set* (*remove-all-empty* (*map normal-dir x*)) **and** *list-ex is-Empty ys*
then show *?thesis*
using *Parallel*(*1*) *remove-all-empty-has-Parallel*
by (*metis* (*mono-tags*, *lifting*) *image-iff list.set-map*)
next
assume *list-ex is-Empty* (*remove-all-empty* (*map normal-dir x*))
then show *?thesis* **.**
qed
then show *?case*
using *remove-all-empty-result* **by** *blast*
next case (*NonD a1 a2*) **then show** *?case* **by** *simp*
next case (*Executable a1 a2*) **then show** *?case* **by** *simp*
next case (*Repeatable a1 a2*) **then show** *?case* **by** *simp*
qed

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 $\bigwedge x.\ x \in$ *set xs* $\Longrightarrow$ *normalised x*
shows *normalised x*
using *assms*
proof (*induct xs arbitrary*: *x*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons a xs*)
then show *?case*
proof (*cases a*)
case *Empty* **then show** *?thesis* **using** *Cons* **by** *simp metis*
next case *Anything* **then show** *?thesis* **using** *Cons* **by** *simp metis*
next case (*Res x3*) **then show** *?thesis* **using** *Cons* **by** *simp metis*
next case (*Copyable x4*) **then show** *?thesis* **using** *Cons* **by** *simp metis*
next
case (*Parallel x5*)
then show *?thesis*
using *Cons* **by** *simp* (*metis Ball-set normalised.simps*(*5*))
next case (*NonD x61 x62*) **then show** *?thesis* **using** *Cons* **by** *simp metis*
next case (*Executable x71 x72*) **then show** *?thesis* **using** *Cons* **by** *simp metis*
next case (*Repeatable x71 x72*) **then show** *?thesis* **using** *Cons* **by** *simp metis*
qed

qed

Normalisation produces resources in normal form

lemma *normalised-normal-dir*:
normalised (*normal-dir a*)
proof (*induct a*)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res x*) **then show** *?case* **by** *simp*
next case (*Copyable a*) **then show** *?case* **by** *simp*
next
case (*Parallel xs*)
have *normalised* (*parallelise* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir xs*))))
proof (*intro normalised-parallelise*)
fix *x*
assume $x \in$ *set* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir xs*)))
then show *normalised x*
using *Parallel*(*1*) *normalised-merge-all-parallel remove-all-empty-subset*
by (*metis* (*mono-tags*, *lifting*) *imageE list.set-map*)
next
show ¬ *list-ex is-Empty* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir xs*)))
using *merge-all-parallel-has-empty remove-all-empty-has-Parallel remove-all-empty-result normal-dir-no-nested-Empty*
by (*metis imageE list.set-map*)
next
show ¬ *list-ex is-Parallel* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir xs*)))
using *merge-all-parallel-has-Parallel remove-all-empty-has-Parallel normal-dir-no-nested-Parallel*
by (*metis imageE list.set-map*)
qed
then show *?case*
by *simp*
next case (*NonD a1 a2*) **then show** *?case* **by** *simp*
next case (*Executable a1 a2*) **then show** *?case* **by** *simp*
next case (*Repeatable a1 a2*) **then show** *?case* **by** *simp*
qed

Normalisation does nothing to resource terms in normal form

lemma *normal-dir-normalised*:
normalised x $\Longrightarrow$ *normal-dir x* = *x*
proof (*induct x*)
case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next case (*Res x*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next

case (*Parallel x*)
then show *?case*
by (*simp add*: *map-idI merge-all-parallel-none normalised-parallel-children not-list-ex parallelise-to-parallel-conv remove-all-empty-none*)
next case (*NonD x1 x2*) **then show** *?case* **by** *simp*
next case (*Executable x1 x2*) **then show** *?case* **by** *simp*
next case (*Repeatable a1 a2*) **then show** *?case* **by** *simp*
qed

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*])
using *parallelise.elims parallelise.simps*(*2*) **by** *blast+*

Equivalent resource terms normalise to equal results

lemma *res-term-equiv-normal-dir*:
$a \sim b \Longrightarrow$ *normal-dir a* = *normal-dir b*
proof (*induct a b rule*: *res-term-equiv.induct*)
case *empty* **then show** *?case* **by** *simp*
next case *anything* **then show** *?case* **by** *simp*
next case (*res x*) **then show** *?case* **by** *simp*
next case (*copyable x*) **then show** *?case* **by** *simp*
next case *nil* **then show** *?case* **by** *simp*
next
case (*singleton a*)
have $\bigwedge$*xs. normal-dir a* = *Parallel xs* $\Longrightarrow$ *parallelise xs* = *Parallel xs*
using *normalised-normal-dir normalised.simps*(*5*) *parallelise-to-parallel-same-length* **by** *metis*
then show *?case*
by (*cases normal-dir a* ; *simp add*: *is-Parallel-def*)
next
case (*merge x y z*)
then show *?case*
proof (*cases normal-dir* (*Parallel y*) = *Empty*)
case *True*
then consider
merge-all-parallel (*remove-all-empty* (*map normal-dir y*)) = []
| *merge-all-parallel* (*remove-all-empty* (*map normal-dir y*)) = [*Empty*]
using *parallelise-to-empty-eq* **by** *fastforce*
then show *?thesis*

proof *cases*
case *1*
then show *?thesis* **by** (*simp add*: *remove-all-empty-append merge-all-parallel-append*)
next
case *2*
have *list-ex is-Empty* (*remove-all-empty* (*map normal-dir y*))
proof (*rule merge-all-parallel-has-empty*)
show *list-ex is-Empty* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir y*)))
using *2* **by** *simp*
show *list-ex is-Empty* (*remove-all-empty* (*map normal-dir y*))
if *Parallel ys* $\in$ *set* (*remove-all-empty* (*map normal-dir y*)) **and** *list-ex is-Empty ys*
for *ys*
using *that remove-all-empty-has-Parallel normal-dir-no-nested-Empty*
by (*metis ex-map-conv*)
qed
then show *?thesis*
using *remove-all-empty-result* **by** *blast*
qed
next
case *False*

have *?thesis* **if** *y*: *normal-dir* (*Parallel y*) = *Parallel ys* **for** *ys*
proof −
consider
merge-all-parallel (*remove-all-empty* (*map normal-dir y*)) = [*Parallel ys*]
| *1* < *length* (*merge-all-parallel* (*remove-all-empty* (*map normal-dir y*)))
and *merge-all-parallel* (*remove-all-empty* (*map normal-dir y*)) = *ys*
using *y parallelise-to-parallel-conv*
by (*fastforce simp add*: *remove-all-empty-append merge-all-parallel-append*)
then show *?thesis*
proof *cases*
case *1*
then show *?thesis*
by (*simp add*: *remove-all-empty-append merge-all-parallel-append*)
(*smt* (*z3*) *image-iff list.set-map list-ex-simps*(*1*) *merge-all-parallel-has-Parallel remove-all-empty-has-Parallel res-term.discI*(*5*) *normal-dir-no-nested-Parallel*)
next
case *2*
then show *?thesis*
using *False y* **by** (*simp add*: *remove-all-empty-append merge-all-parallel-append*)
qed
qed
then show *?thesis*
using *False*
by (*cases normal-dir* (*Parallel y*))
(*simp-all add*: *remove-all-empty-append merge-all-parallel-append*)
qed

next
  case (*parallel xs ys*)
  then have *map normal-dir xs* = *map normal-dir ys*
    by (*clarsimp simp add*: *list-all2-conv-all-nth list-eq-iff-nth-eq*)
  then show *?case*
    by *simp*
next case (*nondet x y u v*) **then show** *?case* **by** *simp*
next case (*executable x y u v*) **then show** *?case* **by** *simp*
next case (*repeatable x y u v*) **then show** *?case* **by** *simp*
next case (*sym x y*) **then show** *?case* **by** *simp*
next case (*trans x y z*) **then show** *?case* **by** *simp*
qed

Equivalence of resource term is equality of their normal forms

lemma *res-term-equiv-is-normal-dir*:
  $a \sim b = (normal\text{-}dir\ a = normal\text{-}dir\ b)$
  using *res-term-equiv-normal-dir normal-dir-eq-imp-equiv* **by** *standard*

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*:
  $[\![normal\text{-}dir\ x = x;\ normal\text{-}dir\ y = y;\ x \sim y]\!] \Longrightarrow x = y$
  using *res-term-equiv-normal-dir* **by** *metis*

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-rewr = *normal-dir*
proof
  fix $x :: ('a, 'b)$ *res-term*
  show *normal-rewr x* = *normal-dir x*
    using *normal-dir-normalised res-term-equiv-normal-dir*
        normal-rewr-normalised res-term-equiv-normal-rewr
    by *metis*
qed

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** $\textit{map-res-term}\ f\ g\ x \sim \textit{map-res-term}\ f\ g\ y$ **and** $\neg\ x \sim y$
proof
show $\textit{map-res-term}\ (\lambda x.\ c)\ g\ (\textit{Res}\ a) \sim \textit{map-res-term}\ (\lambda x.\ c)\ g\ (\textit{Res}\ b)$
by *simp*
show $\neg\ \textit{Res}\ a \sim \textit{Res}\ b$
using *assms* **by** (*simp add*: *res-term-equiv-is-normal-rewr*)
qed

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
fix $f\ g$ **and** $f' :: 'u \Rightarrow 'x$ **and** $g' :: 'v \Rightarrow 'y$ **and** $x :: ('a, 'b)\ \textit{res-term}$
show $(\textit{map-res-term}\ f'\ g' \circ \textit{map-res-term}\ f\ g)\ x = \textit{map-res-term}\ (f' \circ f)\ (g' \circ g)\ x$
by (*induct x* ; *simp add*: *comp-def*)
next
show $\textit{map-res-term}\ \textit{id}\ \textit{id} = \textit{id}$
by (*standard*, *simp add*: *id-def res-term.map-ident*)
qed

Resources are resource terms modulo their equivalence

quotient-type $('a, 'b)\ \textit{resource} = ('a, 'b)\ \textit{res-term}\ /\ \textit{res-term-equiv}$
using *res-term-equiv.equivp* **.**

lemma *abs-resource-eqI* [*intro*]:
  $x \sim y \Longrightarrow$ *abs-resource* x = *abs-resource* y
  using *resource.abs-eq-iff* **by** *blast*
lemma *abs-resource-eqE* [*elim*]:
  $[\![$*abs-resource* x = *abs-resource* y; $x \sim y \Longrightarrow P]\!] \Longrightarrow P$
  using *resource.abs-eq-iff* **by** *blast*

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* ⇒ ('*a*, '*b*) *res-term* **is** *normal-rewr*
  by (*rule res-term-equiv-imp-normal-rewr*)

lemma *of-resource-absorb-normal-rewr* [*simp*]:
  normal-rewr (*of-resource* x) = *of-resource* x
  by (*simp add*: *of-resource.rep-eq*)

lemma *of-resource-absorb-normal-dir* [*simp*]:
  normal-dir (*of-resource* x) = *of-resource* x
  by (*simp add*: *normal-rewr-is-normal-dir*[*symmetric*] *of-resource.rep-eq*)

Equality of resources can be characterised by equality of representative terms

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

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

instance
proof
  fix x y :: ('*a* ,'*b*) *resource*
  have (*of-resource* x = *of-resource* y) = (x = y)
    by *transfer* (*metis res-term-equiv-is-normal-rewr*)
  then show *equal-class.equal* x y = (x = y)
    unfolding *equal-resource-def* **.**
qed
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 (*induct rule*: *res-term-equiv.induct*)
    case *empty* **then show** *?case* **by** *simp*
next case *anything* **then show** *?case* **by** *simp*
next case (*res* x) **then show** *?case* **by** *simp*

next case (*copyable x*) **then show** *?case* **by** *simp*
next case *nil* **then show** *?case* **by** *simp*
next case (*singleton a*) **then show** *?case* **by** *simp*
next case (*merge x y z*) **then show** *?case* **by** (*simp add*: *Un-left-commute*)
next case (*parallel xs ys*) **then show** *?case* **by** (*induct rule*: *list-all2-induct* ; *simp*)
next case (*nondet x y u v*) **then show** *?case* **by** *simp*
next case (*executable x y u v*) **then show** *?case* **by** *simp*
next case (*repeatable x y u v*) **then show** *?case* **by** *simp*
next case (*sym x y*) **then show** *?case* **by** *simp*
next case (*trans x y z*) **then show** *?case* **by** *simp*
qed

lemma *res-term-equiv-set2* [*simp*]:
  $x \sim y \Longrightarrow$ *set2-res-term* $x =$ *set2-res-term* y
proof (*induct rule*: *res-term-equiv.induct*)
  case *empty* **then show** *?case* **by** *simp*
next case *anything* **then show** *?case* **by** *simp*
next case (*res x*) **then show** *?case* **by** *simp*
next case (*copyable x*) **then show** *?case* **by** *simp*
next case *nil* **then show** *?case* **by** *simp*
next case (*singleton a*) **then show** *?case* **by** *simp*
next case (*merge x y z*) **then show** *?case* **by** (*simp add*: *Un-left-commute*)
next case (*parallel xs ys*) **then show** *?case* **by** (*induct rule*: *list-all2-induct* ; *simp*)
next case (*nondet x y u v*) **then show** *?case* **by** *simp*
next case (*executable x y u v*) **then show** *?case* **by** *simp*
next case (*repeatable x y u v*) **then show** *?case* **by** *simp*
next case (*sym x y*) **then show** *?case* **by** *simp*
next case (*trans x y z*) **then show** *?case* **by** *simp*
qed

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

lift-bnf ($'a$, $'b$) *resource*
proof *safe*
  fix *R1* :: $'a \Rightarrow 'u \Rightarrow$ *bool*
  and *R2* :: $'b \Rightarrow 'v \Rightarrow$ *bool*
  and *S1* :: $'u \Rightarrow 'x \Rightarrow$ *bool*
  and *S2* :: $'v \Rightarrow 'y \Rightarrow$ *bool*
  and x :: ($'a$, $'b$) *res-term*
  and $y\ y'$ :: ($'u$, $'v$) *res-term*
  and z :: ($'x$, $'y$) *res-term*

  assume *assms*:
    R1 OO S1 $\neq$ *bot*
    R2 OO S2 $\neq$ *bot*
    rel-res-term R1 R2 x y
    $y \sim y'$
    rel-res-term S1 S2 y' z

  obtain u **where** *ux*: $x =$ *map-res-term fst fst u* **and** *uy*: $y =$ *map-res-term snd*

snd u
and *u-set*: *set1-res-term u* $\subseteq$ {(*x*, *y*). *R1 x y*} *set2-res-term u* $\subseteq$ {(*x*, *y*). *R2 x y*}
using *res-term.in-rel*[*THEN iffD1*, *OF assms*(*3*)] **by** *blast*
obtain *v* **where** *vy'*: *y'* = *map-res-term fst fst v* **and** *vz*: *z* = *map-res-term snd snd v*
and *v-set*: *set1-res-term v* $\subseteq$ {(*x*, *y*). *S1 x y*} *set2-res-term v* $\subseteq$ {(*x*, *y*). *S2 x y*}
using *res-term.in-rel*[*THEN iffD1*, *OF assms*(*5*)] **by** *blast*

obtain *w* **where** *wy*: *w* = *normal-rewr y* **and** *wy'*: *w* = *normal-rewr y'*
using *assms*(*4*) *res-term-equiv-imp-normal-rewr* **by** *blast*

obtain *u'* **where** *uu'*: *u* ∼ *u'* **and** *u'w*: *w* = *map-res-term snd snd u'*
by (*metis res-term-equiv-normal-rewr normal-rewr-map uy wy*)
obtain *v'* **where** *vv'*: *v* ∼ *v'* **and** *v'w*: *w* = *map-res-term fst fst v'*
by (*metis res-term-equiv-normal-rewr normal-rewr-map vy' wy'*)

obtain *x'* **where** *xx'*: *x* ∼ *x'* **and** *u'x'*: *x'* = *map-res-term fst fst u'*
using *map-res-term-preserves-equiv uu' ux* **by** *blast*
obtain *z'* **where** *zz'*: *z* ∼ *z'* **and** *v'z'*: *z'* = *map-res-term snd snd v'*
using *map-res-term-preserves-equiv vv' vz* **by** *blast*

have *rel-res-term R1 R2 x' w*
using *res-term.in-rel u'x' u'w uu' u-set* **by** *force*
moreover have *rel-res-term S1 S2 w z'*
using *res-term.in-rel v'z' v'w vv' v-set* **by** *force*
ultimately have *rel-res-term* (*R1 OO S1*) (*R2 OO S2*) *x' z'*
using *res-term.rel-compp relcompp.relcompI* **by** *metis*
then show ((∼) *OO rel-res-term* (*R1 OO S1*) (*R2 OO S2*) *OO* (∼)) *x z*
using *xx' zz'*[*symmetric*] **by** (*meson relcompp.relcompI*)
next
show $\bigwedge$*Ss1 x xa xb*.
$[\![$*x* ∈ ($\bigcap$*As1*∈*Ss1*. {(*x*, *x'*). *x* ∼ *x'*} ‘‘ {*x*. *set1-res-term x* $\subseteq$ *As1*});
x ∉ {(*x*, *x'*). *x* ∼ *x'*} ‘‘ {*x*. *set1-res-term x* $\subseteq$ $\bigcap$ *Ss1*}; *xa* ∈ *Ss1*; *xa* ∉ {};
xb ∈ $\bigcap$ *Ss1*$]\!]$
$\Longrightarrow$ *xb* ∈ {}
by *simp* (*metis Inf-greatest res-term-equiv-set1*)
next
show $\bigwedge$*Ss2 x xa xb*.
$[\![$*x* ∈ ($\bigcap$*As2*∈*Ss2*. {(*x*, *x'*). *x* ∼ *x'*} ‘‘ {*x*. *set2-res-term x* $\subseteq$ *As2*});
x ∉ {(*x*, *x'*). *x* ∼ *x'*} ‘‘ {*x*. *set2-res-term x* $\subseteq$ $\bigcap$ *Ss2*}; *xa* ∈ *Ss2*; *xa* ∉ {};
xb ∈ $\bigcap$ *Ss2*$]\!]$
$\Longrightarrow$ *xb* ∈ {}
by *simp* (*metis Inf-greatest res-term-equiv-set2*)
qed

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*)
by *transfer simp*

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 *transfer*
fix x :: ('*a*, '*b*) *res-term*

let *?InrL* = *Inr* :: '*a* ⇒ *unit* + '*a*
let *?InrC* = *Inr* :: '*b* ⇒ *unit* + '*b*

have
(⋂ *mx* ∈ *Collect* ((∼) (*map-res-term ?InrL ?InrC x*)). ⋃ (*Basic-BNFs.setr* ‘ *set1-res-term mx*))
= (⋃ *x* :: *unit* + '*a* ∈ *set1-res-term* (*map-res-term ?InrL ?InrC* (*normal-rewr x*)). {*xa*. *x* = *Inr xa*})
proof (*subst Inter-all-same*)
show ⋃ (*Basic-BNFs.setr* ‘ *set1-res-term u*) = ⋃ (*Basic-BNFs.setr* ‘ *set1-res-term v*)
if *u* ∈ *Collect* ((∼) (*map-res-term ?InrL ?InrC x*))
and *v* ∈ *Collect* ((∼) (*map-res-term ?InrL ?InrC x*))
for *u v*
using *that* **by** (*metis mem-Collect-eq res-term-equiv-set1*)
show *map-res-term ?InrL ?InrC* (*normal-rewr x*) ∈ *Collect* ((∼) (*map-res-term ?InrL ?InrC x*))
by (*metis CollectI map-res-term-preserves-equiv res-term-equiv-normal-rewr*)
show
⋃ (*Basic-BNFs.setr* ‘ *set1-res-term* (*map-res-term ?InrL ?InrC* (*normal-rewr x*)))
= (⋃ *x*∈*set1-res-term* (*map-res-term ?InrL ?InrC* (*normal-rewr x*)). {*xa*. *x* = *Inr xa*})
by (*simp add*: *setr-def setrp.simps*)
qed
then show
(⋂ *mx*∈*Collect* ((∼) (*map-res-term ?InrL ?InrC x*)). ⋃ (*Basic-BNFs.setr* ‘ *set1-res-term mx*))
= *set1-res-term* (*normal-rewr x*)
by (*simp add*: *res-term.set-map setr-def setrp.simps*)
qed
lemma *set2-resource*:
fixes x :: ('*a*, '*b*) *resource*
shows *set2-resource x* = *set2-res-term* (*of-resource x*)
proof *transfer*
fix x :: ('*a*, '*b*) *res-term*

let *?InrL* = *Inr* :: '*a* ⇒ *unit* + '*a*
let *?InrC* = *Inr* :: '*b* ⇒ *unit* + '*b*

have
$(\bigcap mx \in Collect\ ((\sim)\ (map\text{-}res\text{-}term\ ?InrL\ ?InrC\ x)).\ \bigcup\ (Basic\text{-}BNFs.setr\ `\ set2\text{-}res\text{-}term\ mx))$
$= (\bigcup x :: unit + 'b \in set2\text{-}res\text{-}term\ (map\text{-}res\text{-}term\ ?InrL\ ?InrC\ (normal\text{-}rewr\ x)).\ \{xa.\ x = Inr\ xa\})$
proof (*subst Inter-all-same*)
show $\bigcup\ (Basic\text{-}BNFs.setr\ `\ set2\text{-}res\text{-}term\ u) = \bigcup\ (Basic\text{-}BNFs.setr\ `\ set2\text{-}res\text{-}term\ v)$
if $u \in Collect\ ((\sim)\ (map\text{-}res\text{-}term\ ?InrL\ ?InrC\ x))$
and $v \in Collect\ ((\sim)\ (map\text{-}res\text{-}term\ ?InrL\ ?InrC\ x))$
for *u v*
using *that* **by** (*metis mem-Collect-eq res-term-equiv-set2*)
show $map\text{-}res\text{-}term\ ?InrL\ ?InrC\ (normal\text{-}rewr\ x) \in Collect\ ((\sim)\ (map\text{-}res\text{-}term\ ?InrL\ ?InrC\ x))$
by (*metis CollectI map-res-term-preserves-equiv res-term-equiv-normal-rewr*)
show
$\bigcup\ (Basic\text{-}BNFs.setr\ `\ set2\text{-}res\text{-}term\ (map\text{-}res\text{-}term\ ?InrL\ ?InrC\ (normal\text{-}rewr\ x)))$
$= (\bigcup x \in set2\text{-}res\text{-}term\ (map\text{-}res\text{-}term\ ?InrL\ ?InrC\ (normal\text{-}rewr\ x)).\ \{xa.\ x = Inr\ xa\})$
by (*simp add*: *setr-def setrp.simps*)
qed
then show
$(\bigcap mx \in Collect\ ((\sim)\ (map\text{-}res\text{-}term\ ?InrL\ ?InrC\ x)).\ \bigcup\ (Basic\text{-}BNFs.setr\ `\ set2\text{-}res\text{-}term\ mx))$
$= set2\text{-}res\text{-}term\ (normal\text{-}rewr\ x)$
by (*simp add*: *res-term.set-map setr-def setrp.simps*)
qed

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* **.**
lift-definition *Anything* :: $('a, 'b)$ *resource*
is *res-term.Anything* **.**
lift-definition *Res* :: $'a \Rightarrow ('a, 'b)$ *resource*
is *res-term.Res* **.**
lift-definition *Copyable* :: $'b \Rightarrow ('a, 'b)$ *resource*
is *res-term.Copyable* **.**
lift-definition *Parallel* :: $('a, 'b)$ *resource list* $\Rightarrow ('a, 'b)$ *resource*
is *res-term.Parallel* **using** *res-term-equiv.parallel* **.**
lift-definition *NonD* :: $('a, 'b)$ *resource* $\Rightarrow ('a, 'b)$ *resource* $\Rightarrow ('a, 'b)$ *resource*
is *res-term.NonD* **using** *res-term-equiv.nondet* **.**
lift-definition *Executable* :: $('a, 'b)$ *resource* $\Rightarrow ('a, 'b)$ *resource* $\Rightarrow ('a, 'b)$ *resource*
is *res-term.Executable* **using** *res-term-equiv.executable* **.**
lift-definition *Repeatable* :: $('a, 'b)$ *resource* $\Rightarrow ('a, 'b)$ *resource* $\Rightarrow ('a, 'b)$ *resource*

is *res-term.Repeatable* **using** *res-term-equiv.repeatable* **.**

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 *transfer*
fix r :: $('a, 'b)$ *res-term* **and** *thesis*
assume $r \sim$ *res-term.Empty* $\Longrightarrow$ *thesis*
and $r \sim$ *res-term.Anything* $\Longrightarrow$ *thesis*
and $\bigwedge a.\ r \sim$ *res-term.Res* $a \Longrightarrow$ *thesis*
and $\bigwedge x.\ r \sim$ *res-term.Copyable* $x \Longrightarrow$ *thesis*
and $\bigwedge xs.\ r \sim$ *res-term.Parallel xs* $\Longrightarrow$ *thesis*
and $\bigwedge x\ y.\ r \sim$ *res-term.NonD* $x\ y \Longrightarrow$ *thesis*
and $\bigwedge x\ y.\ r \sim$ *res-term.Executable* $x\ y \Longrightarrow$ *thesis*
and $\bigwedge x\ y.\ r \sim$ *res-term.Repeatable* $x\ y \Longrightarrow$ *thesis*
note a = *this*

show *thesis*
using a **by** (*cases* r) (*blast intro*: *res-term-equiv.refl*)+
qed

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 $P\ x$
using *res-term.induct*[*of* $\lambda x.\ P$ (*abs-resource* x) *rep-resource* x, *unfolded resource-abs-of-rep*]
using *assms*

by (*smt* (*verit*, *del-insts*) *resource-constr-abs-eq imageE list.set-map*)

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*)
by (*transfer*, *simp add*: *normal-rewr-nondet normal-rewr-executable normal-rewr-repeatable*)+

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*)
by (*transfer*
, *intro res-term-equiv.nil res-term-equiv.singleton res-term-equiv.merge res-term-equiv.drop*)+

lemma *resource-parallel-nested* [*simp*]:
Parallel (*Parallel xs* # *ys*) = *Parallel* (*xs* @ *ys*)
using *resource-merge*[*of Nil*] **by** *simp*

lemma *resource-decompose*:
assumes *Parallel xs* = *Parallel ys*
and *Parallel us* = *Parallel vs*
shows *Parallel* (*xs* @ *us*) = *Parallel* (*ys* @ *vs*)
using *assms* **by** (*metis append-Nil append-Nil2 resource-merge*)

lemma *resource-drop-list*:
($\bigwedge y.\ y \in set\ ys \Longrightarrow y = Empty$) $\Longrightarrow$ *Parallel* (*xs* @ *ys* @ *zs*) = *Parallel* (*xs* @ *zs*)
proof (*induct ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a ys*)
then show *?case*
by *simp* (*metis Cons-eq-appendI resource-drop self-append-conv2*)
qed

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*: $Copyable\ x = Copyable\ y \Longrightarrow x = y$
by (*transfer*, *simp add*: *res-term-equiv-is-normal-rewr*)+

lemma *resource-nondet-eq*:
$NonD\ a\ b = NonD\ x\ y \Longrightarrow a = x$
$NonD\ a\ b = NonD\ x\ y \Longrightarrow b = y$
by (*transfer*, *simp add*: *normal-rewr-nondet res-term-equiv-is-normal-rewr*)+

lemma *resource-executable-eq*:
$Executable\ a\ b = Executable\ x\ y \Longrightarrow a = x$
$Executable\ a\ b = Executable\ x\ y \Longrightarrow b = y$
by (*transfer*, *simp add*: *normal-rewr-executable res-term-equiv-is-normal-rewr*)+

lemma *resource-repeatable-eq*:
$Repeatable\ a\ b = Repeatable\ x\ y \Longrightarrow a = x$
$Repeatable\ a\ b = Repeatable\ x\ y \Longrightarrow b = y$
by (*transfer*, *simp add*: *normal-rewr-repeatable res-term-equiv-is-normal-rewr*)+

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

lemma *resource-neq* [*simp*]:
$Empty \neq Anything$
$Empty \neq Res\ a$
$Empty \neq Copyable\ b$
$Empty \neq NonD\ x\ y$
$Empty \neq Executable\ x\ y$
$Empty \neq Repeatable\ x\ y$
$Anything \neq Res\ a$
$Anything \neq Copyable\ b$
$Anything \neq NonD\ x\ y$
$Anything \neq Executable\ x\ y$
$Anything \neq Repeatable\ x\ y$
$Res\ a \neq Copyable\ b$
$Res\ a \neq NonD\ x\ y$
$Res\ a \neq Executable\ x\ y$
$Res\ a \neq Repeatable\ x\ y$
$Copyable\ b \neq NonD\ x\ y$
$Copyable\ b \neq Executable\ x\ y$
$Copyable\ b \neq Repeatable\ x\ y$
$NonD\ x\ y \neq Executable\ u\ v$
$NonD\ x\ y \neq Repeatable\ u\ v$
$Executable\ x\ y \neq Repeatable\ u\ v$
by (*transfer*, *simp add*: *res-term-equiv-is-normal-dir*)+

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*)
by (*transfer*, *simp*)+

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*])
unfolding *resource-par-def* **by** *transfer simp*

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*])
using *resource-merge*[*of* [*x*] *xs Nil*] **by** (*simp-all add*: *resource-par-def*)

lemma *resource-par-nested-start* [*simp*]:
Parallel (*x* ⊙ *y* # *zs*) = *Parallel* (*x* # *y* # *zs*)
by (*metis append-Cons append-Nil resource-merge resource-par-is-parallel*(*1*) *resource-singleton*)

lemma *resource-par-nested* [*simp*]:
Parallel (*xs* @ *a* ⊙ *b* # *ys*) = *Parallel* (*xs* @ *a* # *b* # *ys*)
using *resource-decompose resource-par-nested-start* **by** *blast*

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*]:
  $Res\ x \odot y = Parallel\ [Res\ x,\ y]$
  by (*simp add*: *resource-par-def*)
lemma *resource-parallel-res* [*code-unfold*]:
  $Parallel\ [Res\ x,\ Parallel\ ys] = Parallel\ (Res\ x\ \#\ ys)$
  by (*metis resource-par-is-parallel*(*1*) *resource-par-res*)

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 −
  have $Parallel\ [Empty,\ x] = x$
   by (*metis append-Nil resource-empty resource-parallel-nested resource-singleton*)
  then show *?thesis*
    by (*simp add*: *resource-par-def*)
qed

lemma *resource-par-unitR* [*simp*]:
  $x \odot Empty = x$
proof −
  have $Parallel\ [x,\ Empty] = x$
    by (*metis resource-empty resource-par-is-parallel*(*1*) *resource-singleton*)
  then show *?thesis*
    by (*simp add*: *resource-par-def*)
qed

lemma *resource-par-assoc* [*simp*]:
  $(a \odot b) \odot c = a \odot (b \odot c)$
  by (*metis resource-par-def resource-par-is-parallel*(*1*) *resource-par-nested-start*)

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*)
  by (*simp add*: *resource-par-def*)

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*])
  unfolding *resource-par-def*
  by *transfer*
   (*meson res-term-equiv-normal-rewr res-term-parallel-cons res-term-equiv.singleton-both*

res-term-equiv-imp-normal-rewr)

7.5 Lifting Parallel Parts

lift-definition *parallel-parts* :: ('*a*, '*b*) *resource* ⇒ ('*a*, '*b*) *resource list*
  is *ResTerm.parallel-parts* **by** (*simp add*: *equiv-parallel-parts*)

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 −
  show *parallel-parts Empty* = []
    by (*simp add*: *parallel-parts.abs-eq resource-constr-abs-eq*)
  show *parallel-parts Anything* = [*Anything*]
    by (*simp add*: *parallel-parts.abs-eq resource-constr-abs-eq*)
  show *parallel-parts* (*Res a*) = [*Res a*]
    by (*simp add*: *parallel-parts.abs-eq resource-constr-abs-eq*)
  show *parallel-parts* (*Copyable b*) = [*Copyable b*]
    by (*simp add*: *parallel-parts.abs-eq Copyable-def*)
  show *parallel-parts* (*Parallel xs*) = *concat* (*map parallel-parts xs*)
  proof (*induct xs*)
    case *Nil*
    then show *?case*
      by (*simp add*: *parallel-parts.abs-eq Empty-def*)
  next
    case (*Cons a xs*)
    then show *?case*
      by (*simp add*: *parallel-parts.abs-eq Parallel-def*, *simp add*: *parallel-parts-def*)
  qed
  show *parallel-parts* (*NonD x y*) = [*NonD x y*]
    by (*simp add*: *parallel-parts.abs-eq NonD-def*)
  show *parallel-parts* (*Executable x y*) = [*Executable x y*]
    by (*simp add*: *parallel-parts.abs-eq Executable-def*)
  show *parallel-parts* (*Repeatable x y*) = [*Repeatable x y*]
    by (*simp add*: *parallel-parts.abs-eq Repeatable-def*)
qed

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

lemma *resource-eq-parallel-parts*:
  x = *Parallel* (*parallel-parts x*)
  by *transfer* (*rule parallel-parts-eq*)

Resources with equal parallel parts are equal

lemma *parallel-parts-cong*:
parallel-parts x = *parallel-parts y* ⟹ *x* = *y*
by (*metis resource-eq-parallel-parts*)

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

lemma *parallel-parts-par*:
parallel-parts (*a* ⊙ *b*) = *parallel-parts a* @ *parallel-parts b*
by (*simp add*: *resource-par-def parallel-parts-simps*)

7.6 Lifting Parallelisation

lift-definition *parallelise* :: ('*a*, '*b*) *resource list* ⇒ ('*a*, '*b*) *resource*
is *ResTerm.parallelise*
by (*metis equiv-parallel-parts res-term-equiv.parallel parallel-parts-parallelise-eq*)

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*)
by (*transfer*, *simp*)+

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*)))
by *transfer* (*simp add*: *normal-rewr-is-normal-dir*)

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

lemma *resource-parallel-eq*:
Parallel xs = *Parallel ys* ⟹ *concat* (*map parallel-parts xs*) = *concat* (*map parallel-parts ys*)
by (*fastforce simp add*: *parallel-parts-simps*(*5*)[*symmetric*])

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*
by (*simp-all add*: *set1-resource of-resource-parallel*)

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*
by (*simp-all add*: *set2-resource of-resource-parallel*)

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* **by** (*simp add*: *res-term-equiv.parallel list-all2I*)

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

lemma *nresource-simp*:
nresource n x = *Parallel* (*replicate n x*)
by (*transfer*, *simp*)

Parallel product of replications is a replication for the combined amount

lemma *nresource-par*:
nresource x a ⊙ *nresource y a* = *nresource* (*x*+*y*) *a*
by (*simp add*: *nresource-simp replicate-add*)

7.9 Lifting Resource Refinement

lift-definition *refine-resource*
:: (′*a* ⇒ (′*x*, ′*y*) *resource*) ⇒ (′*b* ⇒ ′*y*) ⇒ (′*a*, ′*b*) *resource* ⇒ (′*x*, ′*y*) *resource*
is *refine-res-term* **by** (*simp add*: *refine-res-term-eq*)

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*)
by (*transfer*, *simp*)+

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*)
by *transfer* (*simp add*: *res-term-equiv-normal-rewr refine-res-term-eq*)

Refinement passes through resource product

lemma *refine-resource-par*:
refine-resource f g ($x \odot y$) = *refine-resource f g x* $\odot$ *refine-resource f g y*
by (*simp add*: *resource-par-def*)

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* $\odot$ *input q*
| *input* (*Opt p q*) = *NonD* (*input p*) (*input q*)
| *input* (*Represent p*) = *Empty*
| *input* (*Identity a*) = *a*
| *input* (*Swap a b*) = $a \odot b$

| *input* (*InjectL a b*) = *a*
| *input* (*InjectR a b*) = *b*
| *input* (*OptDistrIn a b c*) = *a* $\odot$ (*NonD b c*)
| *input* (*OptDistrOut a b c*) = *NonD* (*a* $\odot$ *b*) (*a* $\odot$ *c*)
| *input* (*Duplicate a*) = *Copyable a*
| *input* (*Erase a*) = *Copyable a*
| *input* (*Apply a b*) = *a* $\odot$ (*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*)
by (*induct x*) *simp-all*

primrec *output* :: ('*a*, '*b*, '*l*, '*m*) *process* $\Rightarrow$ ('*a*, '*b*) *resource*
where
output (*Primitive ins outs l m*) = *outs*
| *output* (*Seq p q*) = *output q*
| *output* (*Par p q*) = *output p* $\odot$ *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* $\odot$ *a*
| *output* (*InjectL a b*) = *NonD a b*
| *output* (*InjectR a b*) = *NonD a b*
| *output* (*OptDistrIn a b c*) = *NonD* (*a* $\odot$ *b*) (*a* $\odot$ *c*)
| *output* (*OptDistrOut a b c*) = *a* $\odot$ (*NonD b c*)
| *output* (*Duplicate a*) = *Copyable a* $\odot$ *Copyable a*
| *output* (*Erase a*) = *Empty*
| *output* (*Apply a b*) = *b*
| *output* (*Repeat a b*) = (*Repeatable a b*) $\odot$ (*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*)
by (*induct x*) *simp-all*

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*)
 by (*induct x*) *simp-all*

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
 let *?x* = *Seq* (*Identity* (*Res a*)) (*Identity* (*Res b*))
 let *?f* = $\lambda x.$ *undefined* — Note that the value used can be anything
 show $\neg$ *valid ?x*
  using *assms resource-res-eq* **by** *fastforce*
 show *valid* (*map-process ?f g h i ?x*)
  by *simp*
qed
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
let *?x* = *Seq* (*Identity* (*Copyable a*)) (*Identity* (*Copyable b*))
let *?g* = $\lambda x.$ *undefined* — Note that the value used can be anything
show $\neg$ *valid ?x*
using *assms resource-copyable-eq* **by** *fastforce*
show *valid* (*map-process f ?g h i ?x*)
by *simp*
qed

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*)
using *assms* **by** (*induct x* ; *simp* ; *metis injD resource.inj-map*)

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* × ('a, 'b) *resource* × 'l × 'm

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

fun *Primitive-unc* :: ('a, 'b, 'l, 'm) *prim-pars* ⇒ ('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* ⇒ ('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*)
by (*induct x*) *simp-all*

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*)
by (*induct x* ; *simp add*: *resource-par-def*)
lemma *process-refineRes-output* [*simp*]:
output (*process-refineRes f g x*) = *refine-resource f g* (*output x*)
by (*induct x* ; *simp add*: *resource-par-def*)
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*)
by (*induct x* ; *simp add*: *image-Un*)
lemma *process-refineRes-valid* [*simp*]:
valid x $\Longrightarrow$ *valid* (*process-refineRes f g x*)
by (*induct x* ; *simp*)

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* $\Rightarrow$ ($'a$, $'b$, $'l$, $'m$) *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 $\neq$ [] $\Longrightarrow$ *input* (*seq-process-list xs*) = *input* (*hd xs*)
by (*induct xs*) *simp-all*

lemma *seq-process-list-output* [*simp*]:
xs $\neq$ [] $\Longrightarrow$ *output* (*seq-process-list xs*) = *output* (*last xs*)
by (*induct xs*) *simp-all*

lemma *seq-process-list-valid*:
valid (*seq-process-list xs*)
= (*list-all valid xs*
$\wedge$ ($\forall$ *i* :: *nat*. *i* < *length xs* − *1* $\longrightarrow$ *output* (*xs* ! *i*) = *input* (*xs* ! *Suc i*)))

proof (*induct xs*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a xs*)
  then show *?case*
    by (*simp add*: *hd-conv-nth nth-Cons'*)
      (*metis Suc-less-eq Suc-pred diff-Suc-1' length-greater-0-conv zero-less-Suc*)
qed

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

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*)
  by *simp-all*

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*)
  by (*induct xs* ; *simp*)

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*
  by (*induct xs*) *simp-all*

lemma *par-process-list-output* [*simp*]:
  output (*par-process-list xs*) = *foldr* (⊙) (*map output xs*) *Empty*
  by (*induct xs*) *simp-all*

lemma *par-process-list-valid* [*simp*]:
  valid (*par-process-list xs*) = *list-all valid xs*
  by (*induct xs* ; *clarsimp*)

lemma *par-process-list-primitives* [*simp*]:
  primitives (*par-process-list xs*) = *concat* (*map primitives xs*)
  by (*induct xs* ; *simp*)

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*)
by *simp-all*

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*)
by (*induct xs* ; *simp*)

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*)
by *simp-all*

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
$process\text{-}subst\ P\ f\ (Primitive\ a\ b\ l\ m) = (if\ P\ a\ b\ l\ m\ then\ f\ a\ b\ l\ m\ else\ Primitive\ a\ b\ l\ m)$
$|\ process\text{-}subst\ P\ f\ (Identity\ a) = Identity\ a$
$|\ process\text{-}subst\ P\ f\ (Swap\ a\ b) = Swap\ a\ b$
$|\ process\text{-}subst\ P\ f\ (Seq\ p\ q) = Seq\ (process\text{-}subst\ P\ f\ p)\ (process\text{-}subst\ P\ f\ q)$
$|\ process\text{-}subst\ P\ f\ (Par\ p\ q) = Par\ (process\text{-}subst\ P\ f\ p)\ (process\text{-}subst\ P\ f\ q)$
$|\ process\text{-}subst\ P\ f\ (Opt\ p\ q) = Opt\ (process\text{-}subst\ P\ f\ p)\ (process\text{-}subst\ P\ f\ q)$
$|\ process\text{-}subst\ P\ f\ (InjectL\ a\ b) = InjectL\ a\ b$
$|\ process\text{-}subst\ P\ f\ (InjectR\ a\ b) = InjectR\ a\ b$
$|\ process\text{-}subst\ P\ f\ (OptDistrIn\ a\ b\ c) = OptDistrIn\ a\ b\ c$
$|\ process\text{-}subst\ P\ f\ (OptDistrOut\ a\ b\ c) = OptDistrOut\ a\ b\ c$
$|\ process\text{-}subst\ P\ f\ (Duplicate\ a) = Duplicate\ a$
$|\ process\text{-}subst\ P\ f\ (Erase\ a) = Erase\ a$
$|\ process\text{-}subst\ P\ f\ (Represent\ p) = Represent\ (process\text{-}subst\ P\ f\ p)$
$|\ process\text{-}subst\ P\ f\ (Apply\ a\ b) = Apply\ a\ b$
$|\ process\text{-}subst\ P\ f\ (Repeat\ a\ b) = Repeat\ a\ b$
$|\ process\text{-}subst\ P\ f\ (Close\ a\ b) = Close\ a\ b$
$|\ process\text{-}subst\ P\ f\ (Once\ a\ b) = Once\ a\ b$
$|\ process\text{-}subst\ P\ f\ (Forget\ a) = Forget\ a$

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

lemma *process-subst-no-target*:
$(\bigwedge a\ b\ l\ m.\ (a,\ b,\ l,\ m) \in set\ (primitives\ x) \Longrightarrow \neg\ P\ a\ b\ l\ m) \Longrightarrow process\text{-}subst\ P\ f\ x = x$
by $(induct\ x,\ auto)$

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

lemma *process-subst-no-prims*:
$primitives\ x = [] \Longrightarrow process\text{-}subst\ P\ f\ x = x$
by $(fastforce\ intro\text{:}\ process\text{-}subst\text{-}no\text{-}target)$

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

lemma *process-subst-input* [*simp*]:
$(\bigwedge a\ b\ l\ m.\ P\ a\ b\ l\ m \Longrightarrow input\ (f\ a\ b\ l\ m) = a) \Longrightarrow input\ (process\text{-}subst\ P\ f\ x) = input\ x$
by $(induct\ x)$ *simp-all*

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 $output\ (process\text{-}subst\ P\ f\ x) = output\ x$
using *assms* **by** $(induct\ x)$ *simp-all*

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*
  using *assms* **by** (*induct x*) *simp-all*

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-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*))
  by (*induct x*) *simp-all*

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-subst P f x*))
   shows $\neg\ P\ a\ b\ l\ m$
  using *assms*
proof (*induct x*)
  case (*Primitive x1 x2 x3 x4*)
  then show *?case*
   by *simp* (*smt* (*verit*) *empty-iff empty-set fst-conv primitives.simps*(*1*) *set-ConsD snd-conv*)
next case (*Seq x1 x2*) **then show** *?case* **by** *simp blast*
next case (*Par x1 x2*) **then show** *?case* **by** *simp blast*
next case (*Opt x1 x2*) **then show** *?case* **by** *simp blast*
next case (*Represent x*) **then show** *?case* **by** *simp*
next case (*Identity x*) **then show** *?case* **by** *simp*
next case (*Swap x1 x2*) **then show** *?case* **by** *simp*
next case (*InjectL x1 x2*) **then show** *?case* **by** *simp*
next case (*InjectR x1 x2*) **then show** *?case* **by** *simp*
next case (*OptDistrIn x1 x2 x3*) **then show** *?case* **by** *simp*
next case (*OptDistrOut x1 x2 x3*) **then show** *?case* **by** *simp*
next case (*Duplicate x*) **then show** *?case* **by** *simp*
next case (*Erase x*) **then show** *?case* **by** *simp*
next case (*Apply x1 x2*) **then show** *?case* **by** *simp*
next case (*Repeat x1 x2*) **then show** *?case* **by** *simp*
next case (*Close x1 x2*) **then show** *?case* **by** *simp*

next case (*Once x1 x2*) **then show** *?case* **by** *simp*
next case (*Forget x*) **then show** *?case* **by** *simp*
qed

Process substitution distributes over list-based sequential and parallel composition

lemma *par-process-list-subst*:
process-subst P f (*par-process-list xs*) = *par-process-list* (*map* (*process-subst P f*) *xs*)
by (*induct xs* ; *simp*)

lemma *seq-process-list-subst*:
process-subst P f (*seq-process-list xs*) = *seq-process-list* (*map* (*process-subst P f*) *xs*)
by (*induct xs* ; *simp*)

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* ⇒ (*'a*, *'b*) *resource* ⇒ (*'a*, *'b*) *resource* ⇒ *bool*
where *spec P a b* ≡ *input P* = *a* ∧ *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* ⇒ ('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 ∼ *y* ⟹ *copyableToRes-term x* ∼ *copyableToRes-term y*
proof (*induct x y rule*: *res-term-equiv.induct*)
case *nil* **then show** *?case* **by** *simp*
next case (*singleton a*) **then show** *?case* **by** *simp*
next
case (*merge x y z*)
then show *?case*
using *res-term-equiv.merge* **by** *force*
next case *empty* **then show** *?case* **by** *simp*
next case *anything* **then show** *?case* **by** *simp*
next case (*res x*) **then show** *?case* **by** *simp*
next case (*copyable x*) **then show** *?case* **by** *simp*
next
case (*parallel xs ys*)
then show *?case*
by (*simp add*: *list.rel-map list-all2-mono res-term-equiv.parallel*)
next case (*nondet x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.nondet*)
next case (*executable x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.executable*)
next case (*repeatable x y u v*) **then show** *?case* **by** (*simp add*: *res-term-equiv.repeatable*)
next case (*sym x y*) **then show** *?case* **by** (*metis res-term-equiv.sym*)
next case (*trans x y z*) **then show** *?case* **by** (*metis res-term-equiv.trans*)
qed

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*
by (*cases x*) *simp-all*
lemma *copyableToRes-term-has-Empty* [*simp*]:
list-ex is-Empty (*map copyableToRes-term xs*) = *list-ex is-Empty xs*
by (*induct xs*) *simp-all*
lemma *copyableToRes-term-has-no-Empty* [*simp*]:
list-all (λ*x*. ¬ *is-Empty x*) (*map copyableToRes-term xs*) = *list-all* (λ*x*. ¬ *is-Empty x*) *xs*
by (*induct xs*) *simp-all*
lemma *copyableToRes-term-is-Parallel* [*simp*]:
is-Parallel (*copyableToRes-term x*) = *is-Parallel x*
by (*cases x*) *simp-all*
lemma *copyableToRes-term-has-Parallel* [*simp*]:
list-ex is-Parallel (*map copyableToRes-term xs*) = *list-ex is-Parallel xs*
by (*induct xs*) *simp-all*
lemma *copyableToRes-term-has-no-Parallel* [*simp*]:
list-all (λ*x*. ¬ *is-Parallel x*) (*map copyableToRes-term xs*) = *list-all* (λ*x*. ¬ *is-Parallel x*) *xs*
by (*induct xs*) *simp-all*

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 (*induct x*)
case (*Res x*) **then show** *?case* **by** *simp*
next case (*Copyable x*) **then show** *?case* **by** *simp*
next case *Empty* **then show** *?case* **by** *simp*
next case *Anything* **then show** *?case* **by** *simp*
next
case (*Parallel xs*)
then show *?case*
proof (*induct xs rule*: *induct-list012*)
case *1* **then show** *?case* **by** *simp*
next case (*2 x*) **then show** *?case* **by** *simp*
next
case (*3 x y zs*)
then have [*simp*]: *list-all normalised* (*map ?f zs*) = *list-all normalised zs*
by (*simp add*: *Ball-set*[*symmetric*])
show *?case*
using *3* **by** *simp*
qed
next case (*NonD x1 x2*) **then show** *?case* **by** *simp*
next case (*Executable x1 x2*) **then show** *?case* **by** *simp*
next case (*Repeatable x1 x2*) **then show** *?case* **by** *simp*
qed

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 (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next case (*Cons a xs*) **then show** *?case* **by** (*cases a*) *simp-all*
qed

lemma *merge-one-parallel-copyableToRes-term-commute*:
merge-one-parallel (*map copyableToRes-term xs*) = *map copyableToRes-term* (*merge-one-parallel xs*)
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next case (*Cons a xs*) **then show** *?case* **by** (*cases a*) *simp-all*
qed

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

```
(step x))
proof (induct x rule: step-induct′)
    case Empty then show ?case by simp
next case Anything then show ?case by simp
next case (Res a) then show ?case by simp
next case (Copyable x) then show ?case by simp
next case (NonD-L x y) then show ?case by simp
next case (NonD-R x y) then show ?case by simp
next case (NonD x y) then show ?case by simp
next case (Executable-L x y) then show ?case by simp
next case (Executable-R x y) then show ?case by simp
next case (Executable x y) then show ?case by simp
next case (Repeatable-L x y) then show ?case by simp
next case (Repeatable-R x y) then show ?case by simp
next case (Repeatable x y) then show ?case by simp
next
  case (Par-Norm xs)
  moreover have list-ex (λx. ¬ normalised x) (map ?f xs)
    using Par-Norm(2) by (fastforce simp add: Bex-set[symmetric])
  ultimately show ?case
    by simp
next
  case (Par-Par xs)
  moreover have ¬ list-ex (λx. ¬ normalised x) (map ?f xs)
    using Par-Par(2) by (simp add: Bex-set[symmetric])
  ultimately show ?case
    by (simp add: merge-one-parallel-copyableToRes-term-commute)
next
  case (Par-Empty xs)
  moreover have ¬ list-ex (λx. ¬ normalised x) (map ?f xs)
    using Par-Empty(2) by (simp add: Bex-set[symmetric])
  moreover have ¬ list-ex is-Parallel xs
    using Par-Empty(3) not-list-ex by metis
  ultimately show ?case
    by (simp add: remove-one-empty-copyableToRes-term-commute)
next case Par-Nil then show ?case by simp
next case (Par-Single u) then show ?case by simp
next
  case (Par v vb vc)
  moreover have ¬ list-ex (λx. ¬ normalised x) (map ?f (v # vb # vc))
    using Par(2) by (simp add: Bex-set[symmetric])
  moreover have ¬ list-ex is-Parallel (v # vb # vc)
    using Par(3) not-list-ex by metis
  moreover have ¬ list-ex is-Empty (v # vb # vc)
    using Par(4) not-list-ex by metis
  ultimately show ?case
    by simp
qed
```

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 (*induct x rule*: *normal-rewr.induct*)
  case (*1 x*)
  then show *?case*
  proof (*cases normalised x*)
    case *True*
    then show *?thesis*
      by *simp*
  next
    case *False*
    then show *?thesis*
      using *1* **by** *simp* (*metis step-copyableToRes-term normal-rewr-step*)
  qed
qed

Copyable term replacement is injective

lemma *copyableToRes-term-inj*:
  copyableToRes-term x = *copyableToRes-term y* ⟹ *x* = *y*
proof (*induct x arbitrary*: *y*)
    case (*Res x*) **then show** *?case* **by** (*cases y*) *simp-all*
next case (*Copyable x*) **then show** *?case* **by** (*cases y*) *simp-all*
next case *Empty* **then show** *?case* **by** (*cases y*) *simp-all*
next case *Anything* **then show** *?case* **by** (*cases y*) *simp-all*
next
  case (*Parallel x*)
  then show *?case*
    by (*cases y*) (*simp-all*, *metis list.inj-map-strong*)
next case (*NonD x1 x2*) **then show** *?case* **by** (*cases y*) *simp-all*
next case (*Executable x1 x2*) **then show** *?case* **by** (*cases y*) *simp-all*
next case (*Repeatable x1 x2*) **then show** *?case* **by** (*cases y*) *simp-all*
qed

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* **by** (*rule copyableToRes-term-equiv*)

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*)

$copyableToRes\ (Repeatable\ x\ y) = Repeatable\ (copyableToRes\ x)\ (copyableToRes\ y)$
by $(transfer,\ simp)+$

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

lemma *copyableToRes-inj*:
fixes $x\ y :: ('a,\ 'b)\ resource$
shows $(copyableToRes\ x :: ('a + 'b,\ 'c)\ resource) = copyableToRes\ y \Longrightarrow x = y$
proof *transfer*
fix $x\ y :: ('a,\ 'b)\ res\text{-}term$
assume $(copyableToRes\text{-}term\ x :: ('a + 'b,\ 'c)\ res\text{-}term) \sim copyableToRes\text{-}term\ y$
then show $x \sim y$
unfolding *res-term-equiv-is-normal-rewr normal-rewr-copyableToRes-term*
by (*rule copyableToRes-term-inj*)
qed

lemma *copyableToRes-eq-conv* [*simp*]:
$(copyableToRes\ x = copyableToRes\ y) = (x = y)$
by (*metis copyableToRes-inj*)

Resource-level replacement can then be applied over a process

primrec $process\text{-}copyableToRes :: ('a,\ 'b,\ 'l,\ 'm)\ process \Rightarrow ('a + 'b,\ 'c,\ 'l,\ 'm)\ process$
where
$process\text{-}copyableToRes\ (Primitive\ ins\ outs\ l\ m) =$
$Primitive\ (copyableToRes\ ins)\ (copyableToRes\ outs)\ l\ m$
$|\ process\text{-}copyableToRes\ (Identity\ a) = Identity\ (copyableToRes\ a)$
$|\ process\text{-}copyableToRes\ (Swap\ a\ b) = Swap\ (copyableToRes\ a)\ (copyableToRes\ b)$
$|\ process\text{-}copyableToRes\ (Seq\ p\ q) = Seq\ (process\text{-}copyableToRes\ p)\ (process\text{-}copyableToRes\ q)$
$|\ process\text{-}copyableToRes\ (Par\ p\ q) = Par\ (process\text{-}copyableToRes\ p)\ (process\text{-}copyableToRes\ q)$
$|\ process\text{-}copyableToRes\ (Opt\ p\ q) = Opt\ (process\text{-}copyableToRes\ p)\ (process\text{-}copyableToRes\ q)$
$|\ process\text{-}copyableToRes\ (InjectL\ a\ b) = InjectL\ (copyableToRes\ a)\ (copyableToRes\ b)$
$|\ process\text{-}copyableToRes\ (InjectR\ a\ b) = InjectR\ (copyableToRes\ a)\ (copyableToRes\ b)$
$|\ process\text{-}copyableToRes\ (OptDistrIn\ a\ b\ c) =$
$OptDistrIn\ (copyableToRes\ a)\ (copyableToRes\ b)\ (copyableToRes\ c)$
$|\ process\text{-}copyableToRes\ (OptDistrOut\ a\ b\ c) =$
$OptDistrOut\ (copyableToRes\ a)\ (copyableToRes\ b)\ (copyableToRes\ c)$
$|\ process\text{-}copyableToRes\ (Duplicate\ a) = undefined$
— There is no sensible definition for *Duplicate*, but we will not need one
$|\ process\text{-}copyableToRes\ (Erase\ a) = undefined$
— 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
fix *x* :: (*'a*, *'b*, *'l*, *'m*) *process*
term *process-copyableToRes* (*makeDuplEraToPrim x*)
:: (*'a* + *'b*, *'c*, *'l* + *String.literal*, *'m* + *unit*) *process*
end

lemma *eliminateCopyable-input*:
input (*process-copyableToRes* (*makeDuplEraToPrim x*)) = *copyableToRes* (*input x*)
by (*induct x*) (*simp-all add*: *resource-par-def*)

lemma *eliminateCopyable-output*:
output (*process-copyableToRes* (*makeDuplEraToPrim x*)) = *copyableToRes* (*output x*)
by (*induct x*) (*simp-all add*: *resource-par-def eliminateCopyable-input*)

lemma *eliminateCopyable-valid*:
valid (*process-copyableToRes* (*makeDuplEraToPrim x*)) = *valid x*
by (*induct x*)
(*simp-all add*: *resource-par-def eliminateCopyable-input eliminateCopyable-output*)

lemma *set2-process-eliminateCopyable*:

fixes *x* :: (*'a*, *'b*, *'l*, *'m*) *process*
shows *set2-process* (*process-copyableToRes* (*makeDuplEraToPrim x*)) = {}
proof −
have [*simp*]: *set2-resource* (*copyableToRes x*) = {}
for *x* :: (*'a*, *'b*) *resource*
by (*induct x rule*: *resource-induct*) *simp-all*
show *?thesis*
by (*induct x*) *simp-all*
qed

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.