Structural Metatheory for Finitary One-Dimensional Higher Inductive Types

Arthur Freitas Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

November 12, 2026

Abstract

This stand-alone entry develops the structural metatheory of a finitary, one-dimensional, nondependent calculus of higher inductive types directly in Isabelle/HOL. It has no dependency on a homotopy type theory object logic. Its terms, raw typing, and reduction are inductive, and its signature environment uniformly describes ordinary inductive types and higher inductive types.

A signature uniformly declares finite lists of point and path constructors. Each endpoint expression is checked against constructor arities and argument types, and instantiating a checked endpoint is proved to preserve typing. Free dimension support is defined structurally and proved to be bounded by the ambient dimension context; substitution at a fresh dimension is therefore the identity on every well-typed term. Path constructors are *cubical-style*: a constructor takes a dimension argument, its faces are obtained by substituting the two constant dimensions, and its eliminator computes judgmentally. A path method is accepted only when both typed faces are joinable with the eliminations of the corresponding declared endpoints, and hence convertible to them. Structural elimination of a checked endpoint is proved well typed. The resulting boundary-computation square states that eliminating either declared face is convertible to taking the same face of the path method after computation. This prevents the interval counterexample in which a constant segment method disagrees with one of the point methods.

Boundary admissibility is no longer left only as a quantification over every context and every typed constructor instance. A finite open-term criterion checks the two faces of each path method once, under canonical variables appended for that constructor's arguments. A soundness theorem for simultaneous substitution transports these finitely many open obligations to all typed instances. Strong normalization and confluence are also packaged as a canonical normal-form function, and typed joinability is proved equivalent to equality of canonical normal forms. Consequently, the finite boundary criterion has an equivalent normal-form formulation. The canonical function uses Hilbert choice,

so this is a theorem-level comparison criterion rather than an executable normalizer. Separately, the hereditary checked judgment remains explicitly a semantic safety invariant: every reduct must satisfy the local structural predicate.

The structural metatheory proves dimension-context weakening, de Bruijn weakening, capture-avoiding term substitution, dimension substitution, and one-step and multi-step preservation of raw typing and hereditary safety. Every raw well-typed term is strongly normalizing. Local confluence on regular terms, together with typing regularity and strong normalization, yields Church–Rosser confluence and unique normal forms. Closed checked terms therefore reach canonical normal forms; at higher inductive type these are point constructors with canonical arguments. A signature with no point constructors has no closed checked inhabitant.

Contents

1 Scope **4**

2 Related work and position **4**

3 The formal development **5**

4 A structural HIT calculus **5**
- 4.1 Types, dimensions, endpoints, and terms 6
- 4.2 Instantiating an endpoint 7
- 4.3 Substitution for term variables 7
- 4.4 Substitution for dimension variables 8
- 4.5 Iterated application and eliminator arguments 8

5 Reduction and conversion **9**
- 5.1 Normal forms and the strong-normalization predicate 12
- 5.2 Reduction inside constructor arguments 13

6 Typing **14**
- 6.1 Applying a method to the eliminator arguments 16

7 Well-formed signatures and endpoints **17**

8 Dimension support **19**

9 Boundary-coherent elimination **22**
- 9.1 The boundary-computation square 27

10 Structural properties of typing **32**

11 Term substitution **36**

12 Dimension substitution **40**

13 Subject reduction **43**

14 Progress and canonical forms **50**
14.1 Canonical values 50
14.2 Normal forms are closed under the relevant subterms 51
14.3 Progress 51
14.4 The empty signature 55

15 Strong normalization **56**
15.1 Dimension erasure and regular terms 56
15.2 Renaming and simultaneous substitution 64
15.3 Hereditary computability at HIT types 79
15.4 A Kripke logical relation 87
15.5 The hereditary Kripke candidate 93
15.6 Substitution and reducibility 97
15.7 Constructor and method arguments 106
15.8 The fundamental theorem 120

16 Unconditional canonical results and consistency **126**

17 Confluence **127**
17.1 A dimension-capture critical pair 127
17.2 Multi-step compatibility 128
17.3 Dimension-safe reduction 135
17.4 Local peaks 147

18 Canonical normal-form comparison **170**

19 Finite open-term boundary checking **173**

20 The schema at work **194**
20.1 Well-formed signatures 195
20.2 Boundaries of path constructors 197
20.3 Boundary-coherent path methods 197
20.4 Computation rules 202
20.5 Typing 203
20.6 The loop is a value that only a dimension makes possible . . 204

21 AI assistance **204**

1 Scope

The calculus has two type formers, function spaces and higher inductive types drawn from the signature environment. Constructor arguments are either parameters of fixed higher inductive type or recursive occurrences. The schema covers the circle, interval, suspensions, and the indiscrete paths of propositional truncation. It does not express general quotients by a relation family whose witness type depends on earlier arguments.

Infinitary constructors, dependent constructor arguments, indexed families, dependent eliminators, higher path constructors, dimension abstraction, composition, and Kan operations are outside the scope. Thus "cubical-style" describes the dimension-indexed path syntax and judgmental path computation; this is not a formalization of cubical type theory. The metatheory does prove strong normalization and Church–Rosser confluence for its stated finitary, one-dimensional, nondependent calculus; it does not claim those results for the richer systems excluded above.

2 Related work and position

The general shape of the schema follows the informal descriptions of higher inductive definitions in the Homotopy Type Theory book [9] and the schemata studied by Lumsdaine and Shulman [6] and by Sojakova [7]. Kaposi and Kovács give a substantially more expressive theory of signatures for higher inductive-inductive types, formalize its syntactic translations in Agda, and generate induction principles [5]. The present calculus is narrower—nondependent, one-sorted at each signature index, and one-dimensional—but proves an operational metatheory not supplied by that signature translation: explicit reduction, subject reduction, strong normalization, Church–Rosser, unique normal forms, and consistency.

The closest Isabelle-specific precedent is Chen's Isabelle/HoTT [2], the first implementation of homotopy type theory in Isabelle. It uses Isabelle's logical-framework infrastructure for intensional dependent type theory, type checking, elaboration, Isar proofs, and foundational HoTT results. Individual higher inductive constants and rules can be postulated in such an object logic, but Isabelle/HoTT does not supply a uniform HIT signature schema, an open boundary-admissibility criterion, or the operational metatheory proved here. This entry instead reifies its calculus as datatypes and relations in a stand-alone Isabelle/HOL session.

Veltri and van der Weide define internal signatures for finitary 1-truncated higher inductive types and construct their algebras from groupoid quotients in UniMath [10]. Their result is an internal semantic construction with initial-algebra content. This entry instead studies an external first-order term calculus and verifies its operational properties in Isabelle; it

neither constructs semantic models nor claims the expressive generality of their signatures.

The judgmental treatment of path constructors is inspired by cubical type theory [3]. Cavallo and Harper provide a general indexed cubical inductive schema with computational higher inductive types and canonicity [1]; Huber proves canonicity for cubical type theory [4]; and Sterling and Angiuli prove normalization and decidability of judgmental equality for Cartesian cubical type theory [8]. In contrast, the present system has no composition, Kan operations, dimension abstraction, indexed families, or higher path constructors. Its distinct contribution is a compact one-dimensional operational calculus with explicit boundary-coherence and dimension-capture analyses, together with machine-checked strong normalization and Church–Rosser proofs.

Dimension variables are natural numbers, and the index 0 is reserved for the dimension bound by the path methods of an eliminator. Ambient dimension contexts therefore consist of strictly positive indices, and dimension substitution never has to shift.

3 The formal development

theory *HIT-Syntax*
  imports *Main*
begin

4 A structural HIT calculus

This stand-alone Isabelle/HOL session fixes a small type theory whose *only* type formers are function spaces and higher inductive types drawn from a signature environment. The calculus is finitary, one-dimensional and non-dependent: signatures contain finite lists of point and path constructors, constructor arguments have fixed higher-inductive types, and eliminators have a constant motive. The development studies syntax, reduction, typing, signature well-formedness, free dimension support, boundary coherence, substitution, subject reduction, progress, multi-step type safety, strong normalization, confluence, and canonical forms.

Two design decisions make the treatment uniform.

- Higher inductive types are not built in one at a time. A signature lists point constructors and path constructors, each with a list of argument specifications, and each path constructor with two endpoints expressed over its own arguments. Ordinary inductive types are the signatures with no path constructors, so the natural numbers, the unit type and the empty type are instances of the same schema.

- Path constructors are *cubical-style*: a path constructor applied to its arguments takes a further dimension argument, and the eliminator computes on it judgmentally. The endpoints are not separate rules but the result of substituting the two constant dimensions, so the boundary of a path constructor is part of dimension substitution rather than of reduction. This is a small syntax with dimension variables, not a formalization of cubical type theory or its Kan operations.

Dimension variables are natural numbers. Index *0* is reserved: it is the dimension bound by the path methods of an eliminator. Ambient dimension contexts therefore consist of strictly positive indices, and no shifting is ever required.

4.1 Types, dimensions, endpoints, and terms

datatype *ty* =
  Fun ty ty
| *Hit nat*

An argument of a constructor is either a parameter, whose type is one of the higher inductive types of the signature environment, or a recursive occurrence of the type being described. Restricting parameters to higher inductive types is no loss for the usual examples: ordinary inductive types are themselves instances of the schema. What it excludes is infinitary constructors, whose arguments are functions into the type being defined.

datatype *argspec* =
  Par nat
| *Rec*

An endpoint of a path constructor is a point-constructor term over the arguments of that path constructor.

datatype *endpt* =
  EArg nat
| *ECon nat endpt list*

datatype *dim* =
  D0
| *D1*
| *DVar nat*

record *hsig* =
  pcons :: *argspec list list*
  lcons :: *argspec list list*
  lends :: (*endpt* × *endpt*) *list*

datatype *tm* =
  Var nat

```
| Lam ty tm
| App tm tm
| PCon nat nat tm list
| LCon nat nat tm list dim
| HElim nat ty tm list tm list tm

type-synonym sigenv = nat ⇒ hsig
```

4.2 Instantiating an endpoint

```
primrec instE :: nat ⇒ tm list ⇒ endpt ⇒ tm where
  instE h as (EArg j) = as ! j
| instE h as (ECon k es) = PCon h k (map (instE h as) es)

definition lhsOf :: sigenv ⇒ nat ⇒ nat ⇒ tm list ⇒ tm where
  lhsOf S h k as = instE h as (fst (lends (S h) ! k))

definition rhsOf :: sigenv ⇒ nat ⇒ nat ⇒ tm list ⇒ tm where
  rhsOf S h k as = instE h as (snd (lends (S h) ! k))
```

The smart constructor for path constructors. Applying a path constructor at a constant dimension *is* the corresponding endpoint; there is no separate boundary rule.

```
definition mkL :: sigenv ⇒ nat ⇒ nat ⇒ tm list ⇒ dim ⇒ tm where
  mkL S h k as d =
    (case d of
       D0 ⇒ lhsOf S h k as
     | D1 ⇒ rhsOf S h k as
     | DVar j ⇒ LCon h k as (DVar j))

lemma mkL-simps [simp]:
  mkL S h k as D0 = lhsOf S h k as
  mkL S h k as D1 = rhsOf S h k as
  mkL S h k as (DVar j) = LCon h k as (DVar j)
  by (simp-all add: mkL-def)
```

4.3 Substitution for term variables

```
primrec lift :: nat ⇒ tm ⇒ tm where
  lift n (Var i) = (if i < n then Var i else Var (Suc i))
| lift n (Lam A t) = Lam A (lift (Suc n) t)
| lift n (App t u) = App (lift n t) (lift n u)
| lift n (PCon h k as) = PCon h k (map (lift n) as)
| lift n (LCon h k as d) = LCon h k (map (lift n) as) d
| lift n (HElim h C pms lms t) =
    HElim h C (map (lift n) pms) (map (lift n) lms) (lift n t)

primrec subst :: nat ⇒ tm ⇒ tm ⇒ tm where
  subst n s (Var i) =
```

```
    (if i < n then Var i else if i = n then s else Var (i − 1))
| subst n s (Lam A t) = Lam A (subst (Suc n) (lift 0 s) t)
| subst n s (App t u) = App (subst n s t) (subst n s u)
| subst n s (PCon h k as) = PCon h k (map (subst n s) as)
| subst n s (LCon h k as d) = LCon h k (map (subst n s) as) d
| subst n s (HElim h C pms lms t) =
    HElim h C (map (subst n s) pms) (map (subst n s) lms) (subst n s t)
```

4.4 Substitution for dimension variables

```
primrec dsubstD :: nat ⇒ dim ⇒ dim ⇒ dim where
  dsubstD n d D0 = D0
| dsubstD n d D1 = D1
| dsubstD n d (DVar j) = (if j = n then d else DVar j)

primrec dsubst :: sigenv ⇒ nat ⇒ dim ⇒ tm ⇒ tm where
  dsubst S n d (Var i) = Var i
| dsubst S n d (Lam A t) = Lam A (dsubst S n d t)
| dsubst S n d (App t u) = App (dsubst S n d t) (dsubst S n d u)
| dsubst S n d (PCon h k as) = PCon h k (map (dsubst S n d) as)
| dsubst S n d (LCon h k as e) =
    mkL S h k (map (dsubst S n d) as) (dsubstD n d e)
| dsubst S n d (HElim h C pms lms t) =
    HElim h C (map (dsubst S n d) pms)
      (if n = 0 then lms else map (dsubst S n d) lms)
      (dsubst S n d t)
```

4.5 Iterated application and eliminator arguments

```
definition apps :: tm ⇒ tm list ⇒ tm where
  apps t xs = foldl App t xs

lemma apps-Nil [simp]: apps t [] = t
  by (simp add: apps-def)

lemma apps-Cons [simp]: apps t (x # xs) = apps (App t x) xs
  by (simp add: apps-def)

lemma apps-append: apps t (xs @ ys) = apps (apps t xs) ys
  by (simp add: apps-def)
```

When an eliminator meets a constructor, every argument is handed to the method, and every *recursive* argument is handed over a second time together with the value of the eliminator on it.

```
primrec elimArgs ::
  nat ⇒ ty ⇒ tm list ⇒ tm list ⇒ argspec list ⇒ tm list ⇒ tm list
where
  elimArgs h C pms lms [] as = []
| elimArgs h C pms lms (s # ss) as =
```

```
  (case as of
     [] ⇒ []
   | a # as' ⇒
       (case s of
          Par h' ⇒ [a]
        | Rec ⇒ [a, HElim h C pms lms a])
       @ elimArgs h C pms lms ss as')

lemma elimArgs-Nil-args [simp]: elimArgs h C pms lms ss [] = []
  by (induct ss) simp-all

end
theory HIT-Reduction
  imports HIT-Syntax
begin
```

5 Reduction and conversion

There are exactly two computation rules: the usual β-rule, and the rule that an eliminator applied to a constructor hands the arguments to the corresponding method. The second rule has two instances, one for point constructors and one for path constructors. The instance for path constructors is the cubical-style computation rule: the method is a term with a free dimension variable, namely the reserved index *0*, and the reduct first instantiates that variable with the dimension of the path constructor and only then applies the constructor and recursive arguments. This order is significant under nested eliminators: the bound dimension of a method must not capture a dimension occurring freely in an argument supplied to that method. There is no boundary rule: applying a path constructor at a constant dimension is, by the definition of *mkL*, already the corresponding endpoint.

```
definition dclosed0 :: sigenv ⇒ tm ⇒ bool where
  dclosed0 S t ⟷ (∀j. dsubst S 0 (DVar j) t = t)

lemma dclosed0-list-iff:
  (∀j. map (dsubst S 0 (DVar j)) xs = xs) ⟷
    list-all (dclosed0 S) xs
  by (induct xs) (auto simp: dclosed0-def)

lemma dclosed0-Var [simp]: dclosed0 S (Var i)
  unfolding dclosed0-def by simp

lemma dclosed0-Lam [simp]:
  dclosed0 S (Lam A t) ⟷ dclosed0 S t
  unfolding dclosed0-def by simp

lemma dclosed0-App [simp]:
  dclosed0 S (App t u) ⟷
```

```
    dclosed0 S t ∧ dclosed0 S u
  unfolding dclosed0-def by auto

lemma dclosed0-PCon [simp]:
  dclosed0 S (PCon h k as) ⟷
    list-all (dclosed0 S) as
  unfolding dclosed0-def
  by (induct as) auto

lemma dclosed0-LCon [simp]:
  dclosed0 S (LCon h k as (DVar i)) ⟷
    i ≠ 0 ∧ list-all (dclosed0 S) as
proof (cases i)
  case 0
  then show ?thesis
    unfolding dclosed0-def by auto
next
  case (Suc i)
  then show ?thesis
    unfolding dclosed0-def
    by (induct as) auto
qed

lemma dclosed0-HElim [simp]:
  dclosed0 S (HElim h C pms lms t) ⟷
    list-all (dclosed0 S) pms ∧ dclosed0 S t
  unfolding dclosed0-def
  by (induct pms) auto

inductive red :: sigenv ⇒ tm ⇒ tm ⇒ bool for S :: sigenv where
  beta:
    dclosed0 S u ⟹
      red S (App (Lam A t) u) (subst 0 u t)
| elimPoint:
    k < length (pcons (S h)) ⟹
      red S (HElim h C pms lms (PCon h k as))
        (apps (pms ! k) (elimArgs h C pms lms (pcons (S h) ! k) as))
| elimPath:
    k < length (lcons (S h)) ⟹
      red S (HElim h C pms lms (LCon h k as (DVar j)))
        (apps (dsubst S 0 (DVar j) (lms ! k))
          (elimArgs h C pms lms (lcons (S h) ! k) as))
| appL: red S t t' ⟹ red S (App t u) (App t' u)
| appR: red S u u' ⟹ red S (App t u) (App t u')
| lam: red S t t' ⟹ red S (Lam A t) (Lam A t')
| pconArg:
    red S a a' ⟹
      red S (PCon h k (xs @ a # ys)) (PCon h k (xs @ a' # ys))
| lconArg:
```

```
    red S a a' ⟹
      red S (LCon h k (xs @ a # ys) d) (LCon h k (xs @ a' # ys) d)
| elimPm:
    red S m m' ⟹
      red S (HElim h C (xs @ m # ys) lms t)
        (HElim h C (xs @ m' # ys) lms t)
| elimLm:
    red S m m' ⟹
      red S (HElim h C pms (xs @ m # ys) t)
        (HElim h C pms (xs @ m' # ys) t)
| elimScrutinee:
    red S t t' ⟹
      red S (HElim h C pms lms t) (HElim h C pms lms t')

abbreviation reds :: sigenv ⇒ tm ⇒ tm ⇒ bool where
  reds S ≡ (red S)**

definition conv :: sigenv ⇒ tm ⇒ tm ⇒ bool where
  conv S = (λt u. (λa b. red S a b ∨ red S b a)** t u)

lemma conv-refl: conv S t t
  by (simp add: conv-def)

lemma conv-sym:
  assumes conv S t u
  shows conv S u t
proof −
  have symp (λa b. red S a b ∨ red S b a)
    by (simp add: symp-def)
  then have symp ((λa b. red S a b ∨ red S b a)**)
    by (rule symp-rtranclp)
  with assms show ?thesis
    unfolding conv-def by (simp add: symp-def)
qed

lemma conv-trans:
  conv S t u ⟹ conv S u v ⟹ conv S t v
  unfolding conv-def by (rule rtranclp-trans)

lemma red-conv: red S t u ⟹ conv S t u
  unfolding conv-def by auto

lemma reds-conv: reds S t u ⟹ conv S t u
proof (induct rule: rtranclp-induct)
  case base
  show ?case by (rule conv-refl)
next
  case (step y z)
  from step(3) red-conv[OF step(2)] show ?case by (rule conv-trans)
```

qed

definition *joinable* :: *sigenv* ⇒ *tm* ⇒ *tm* ⇒ *bool* **where**
joinable S t u ⟷ (∃ *v*. *reds S t v* ∧ *reds S u v*)

lemma *joinable-refl* [*simp*]: *joinable S t t*
unfolding *joinable-def* **by** *auto*

lemma *joinable-sym*:
joinable S t u ⟹ *joinable S u t*
unfolding *joinable-def* **by** *blast*

lemma *joinableI*:
reds S t v ⟹ *reds S u v* ⟹ *joinable S t u*
unfolding *joinable-def* **by** *blast*

lemma *joinable-conv*:
assumes *joinable S t u*
shows *conv S t u*
proof −
from *assms* **obtain** *v* **where** *tv*: *reds S t v* **and** *uv*: *reds S u v*
unfolding *joinable-def* **by** *blast*
show *?thesis*
by (*rule conv-trans*[*OF reds-conv*[*OF tv*] *conv-sym*[*OF reds-conv*[*OF uv*]]])
qed

5.1 Normal forms and the strong-normalization predicate

definition *nform* :: *sigenv* ⇒ *tm* ⇒ *bool* **where**
nform S t ⟷ (∄ *u*. *red S t u*)

inductive *SN* :: *sigenv* ⇒ *tm* ⇒ *bool* **for** *S* :: *sigenv* **where**
SNI: (⋀*u*. *red S t u* ⟹ *SN S u*) ⟹ *SN S t*

lemma *SN-step*: *SN S t* ⟹ *red S t u* ⟹ *SN S u*
by (*induct rule*: *SN.induct*) *blast*

lemma *SN-reds*: *reds S t u* ⟹ *SN S t* ⟹ *SN S u*
by (*induct rule*: *rtranclp-induct*) (*auto intro*: *SN-step*)

lemma *SN-nform*: *nform S t* ⟹ *SN S t*
unfolding *nform-def* **by** (*rule SNI*) *blast*

The strong-normalization predicate passes from a one-hole context to the term filling it, provided the context is monotone for reduction.

lemma *SN-context*:
assumes *mono*: ⋀*a b*. *red S a b* ⟹ *red S* (*f a*) (*f b*)
and *sn*: *SN S s*
and *eq*: *s* = *f t*

```
  shows SN S t
  using sn eq
proof (induct arbitrary: t rule: SN.induct)
  case (SNI s)
  show ?case
  proof (rule SN.SNI)
    fix v
    assume red S t v
    then have red S (f t) (f v) by (rule mono)
    with SNI(3) have red S s (f v) by simp
    from SNI(2)[OF this refl] show SN S v .
  qed
qed

lemma SN-App1: SN S (App t u) ⟹ SN S t
  by (rule SN-context[where f = λx. App x u]) (auto intro: red.appL)

lemma SN-App2: SN S (App t u) ⟹ SN S u
  by (rule SN-context[where f = λx. App t x]) (auto intro: red.appR)

lemma SN-apps-head: SN S (apps t xs) ⟹ SN S t
  by (induct xs arbitrary: t) (auto dest: SN-App1)
```

5.2 Reduction inside constructor arguments

```
lemma red-PCon-arg:
  assumes i < length as and red S (as ! i) a'
  shows red S (PCon h k as) (PCon h k (as[i := a']))
proof −
  have as = take i as @ as ! i # drop (Suc i) as
    using assms(1) by (simp add: id-take-nth-drop)
  moreover have as[i := a'] = take i as @ a' # drop (Suc i) as
    using assms(1) by (simp add: upd-conv-take-nth-drop)
  ultimately show ?thesis
    using red.pconArg[OF assms(2), of h k take i as drop (Suc i) as]
    by simp
qed

lemma SN-PCon-arg:
  assumes sn: SN S (PCon h k as) and i: i < length as
  shows SN S (as ! i)
proof (rule SN-context[where f = λx. PCon h k (as[i := x])])
  fix a b
  assume red S a b
  moreover have as[i := a] ! i = a using i by simp
  ultimately have red S (PCon h k (as[i := a]))
      (PCon h k ((as[i := a])[i := b]))
    using i by (metis length-list-update red-PCon-arg)
  then show red S (PCon h k (as[i := a])) (PCon h k (as[i := b]))
```

```
    by simp
next
  show SN S (PCon h k as) by (rule sn)
next
  show PCon h k as = PCon h k (as[i := as ! i]) by simp
qed

lemma red-LCon-arg:
  assumes i < length as and red S (as ! i) a'
  shows red S (LCon h k as d) (LCon h k (as[i := a']) d)
proof −
  have as = take i as @ as ! i # drop (Suc i) as
    using assms(1) by (simp add: id-take-nth-drop)
  moreover have as[i := a'] = take i as @ a' # drop (Suc i) as
    using assms(1) by (simp add: upd-conv-take-nth-drop)
  ultimately show ?thesis
    using red.lconArg[OF assms(2), of h k take i as drop (Suc i) as d]
    by simp
qed

lemma SN-LCon-arg:
  assumes sn: SN S (LCon h k as d) and i: i < length as
  shows SN S (as ! i)
proof (rule SN-context[where f = λx. LCon h k (as[i := x]) d])
  fix a b
  assume red S a b
  moreover have as[i := a] ! i = a using i by simp
  ultimately have red S (LCon h k (as[i := a]) d)
      (LCon h k ((as[i := a])[i := b]) d)
    using i by (metis length-list-update red-LCon-arg)
  then show red S (LCon h k (as[i := a]) d)
      (LCon h k (as[i := b]) d)
    by simp
next
  show SN S (LCon h k as d) by (rule sn)
next
  show LCon h k as d = LCon h k (as[i := as ! i]) d by simp
qed

end
theory HIT-Typing
  imports HIT-Reduction
begin
```

6 Typing

A judgment carries an ambient dimension context, a set of dimension variables, and an ordinary typing context. The reserved index *0* never belongs

to an ambient dimension context: it is bound by the path methods of an eliminator, and the typing rule for the eliminator is the only place where it enters scope.

```
primrec argTy :: nat ⇒ argspec ⇒ ty where
  argTy h (Par h') = Hit h'
| argTy h Rec = Hit h
```

A method for a constructor consumes the arguments of that constructor, and in addition, for every recursive argument, the value of the eliminator on it. This is exactly the list produced by *elimArgs*.

```
primrec methodTy :: nat ⇒ ty ⇒ argspec list ⇒ ty where
  methodTy h C [] = C
| methodTy h C (s # ss) =
    (case s of
       Par h' ⇒ Fun (Hit h') (methodTy h C ss)
     | Rec ⇒ Fun (Hit h) (Fun C (methodTy h C ss)))
```

```
inductive wt :: sigenv ⇒ nat set ⇒ ty list ⇒ tm ⇒ ty ⇒ bool
  for S :: sigenv
where
  wtVar:
    i < length G ⟹ wt S D G (Var i) (G ! i)
| wtLam:
    wt S D (A # G) t B ⟹ wt S D G (Lam A t) (Fun A B)
| wtApp:
    wt S D G t (Fun A B) ⟹ wt S D G u A ⟹ wt S D G (App t u) B
| wtPCon:
    k < length (pcons (S h)) ⟹
      length as = length (pcons (S h) ! k) ⟹
      (∀ i < length as.
        wt S D G (as ! i) (argTy h (pcons (S h) ! k ! i))) ⟹
      wt S D G (PCon h k as) (Hit h)
| wtLCon:
    j ∈ D ⟹ k < length (lcons (S h)) ⟹
      length as = length (lcons (S h) ! k) ⟹
      (∀ i < length as.
        wt S D G (as ! i) (argTy h (lcons (S h) ! k ! i))) ⟹
      wt S D G (LCon h k as (DVar j)) (Hit h)
| wtElim:
    wt S D G t (Hit h) ⟹
      length pms = length (pcons (S h)) ⟹
      length lms = length (lcons (S h)) ⟹
      (∀ k < length pms.
        wt S D G (pms ! k) (methodTy h C (pcons (S h) ! k))) ⟹
      (∀ k < length lms.
        wt S (insert 0 D) G (lms ! k)
          (methodTy h C (lcons (S h) ! k))) ⟹
      wt S D G (HElim h C pms lms t) C
```

6.1 Applying a method to the eliminator arguments

The computation rules replace an eliminator at a constructor by the corresponding method applied to *elimArgs*. The following lemma is the typing analogue: a method of the declared type, applied to arguments of the declared kinds, lands in the motive.

lemma *wt-apps-elimArgs*:
  assumes *E*: $\bigwedge a.$ *wt S D G a* (*Hit h*) $\Longrightarrow$
    wt S D G (*HElim h C pms lms a*) *C*
  shows $\bigwedge$*as m. length as* = *length ss* $\Longrightarrow$
    ($\forall$ *i* < *length ss. wt S D G* (*as* ! *i*) (*argTy h* (*ss* ! *i*))) $\Longrightarrow$
    wt S D G m (*methodTy h C ss*) $\Longrightarrow$
    wt S D G (*apps m* (*elimArgs h C pms lms ss as*)) *C*
proof (*induct ss*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons s ss*)
  from *Cons*(*2*) **obtain** *a as′* **where** *as*: *as* = *a* # *as′*
    and *len*: *length as′* = *length ss*
    by (*cases as*) *auto*
  have *a0*: *wt S D G a* (*argTy h s*)
    using *Cons*(*3*) *as* **by** *force*
  have *rest*: $\forall$ *i* < *length ss. wt S D G* (*as′* ! *i*) (*argTy h* (*ss* ! *i*))
    using *Cons*(*3*) *as* **by** *fastforce*
  show *?case*
  proof (*cases s*)
    case (*Par h′*)
    then have *wt S D G a* (*Hit h′*) **using** *a0* **by** *simp*
    moreover from *Cons*(*4*) *Par*
      have *wt S D G m* (*Fun* (*Hit h′*) (*methodTy h C ss*)) **by** *simp*
    ultimately have *wt S D G* (*App m a*) (*methodTy h C ss*)
      by (*rule wt.wtApp*[*rotated*])
    then have *wt S D G* (*apps* (*App m a*) (*elimArgs h C pms lms ss as′*)) *C*
      by (*rule Cons*(*1*)[*OF len rest*])
    then show *?thesis* **using** *as Par* **by** *simp*
  next
    case *Rec*
    then have *aH*: *wt S D G a* (*Hit h*) **using** *a0* **by** *simp*
    from *Cons*(*4*) *Rec*
      have *wt S D G m* (*Fun* (*Hit h*) (*Fun C* (*methodTy h C ss*))) **by** *simp*
    from *wt.wtApp*[*OF this aH*]
      have *step*: *wt S D G* (*App m a*) (*Fun C* (*methodTy h C ss*)) **.**
    have *wt S D G* (*App* (*App m a*) (*HElim h C pms lms a*)) (*methodTy h C ss*)
      by (*rule wt.wtApp*[*OF step E*[*OF aH*]])
    then have *wt S D G*
        (*apps* (*App* (*App m a*) (*HElim h C pms lms a*))
          (*elimArgs h C pms lms ss as′*)) *C*
      by (*rule Cons*(*1*)[*OF len rest*])

```
    then show ?thesis using as Rec by simp
  qed
qed

end
theory HIT-Signature
  imports HIT-Typing
begin
```

7 Well-formed signatures and endpoints

Endpoint expressions are checked against the constructor arguments in scope. An endpoint argument has the type declared by its argument specification, while an endpoint constructor must be a declared point constructor with well-typed endpoint arguments of its own.

```
inductive wf-endpt ::
  sigenv ⇒ nat ⇒ argspec list ⇒ endpt ⇒ ty ⇒ bool
  for S :: sigenv and h :: nat and ss :: argspec list
where
  wfEArg:
    i < length ss ⟹
      wf-endpt S h ss (EArg i) (argTy h (ss ! i))
| wfECon:
    k < length (pcons (S h)) ⟹
      length es = length (pcons (S h) ! k) ⟹
      (∀ i < length es.
        wf-endpt S h ss (es ! i)
          (argTy h (pcons (S h) ! k ! i))) ⟹
      wf-endpt S h ss (ECon k es) (Hit h)

definition wf-hsig :: sigenv ⇒ nat ⇒ bool where
  wf-hsig S h ⟷
    length (lends (S h)) = length (lcons (S h)) ∧
    (∀ k < length (lcons (S h)).
      wf-endpt S h (lcons (S h) ! k) (fst (lends (S h) ! k)) (Hit h) ∧
      wf-endpt S h (lcons (S h) ! k) (snd (lends (S h) ! k)) (Hit h))

definition wf-sigenv :: sigenv ⇒ bool where
  wf-sigenv S ⟷ (∀ h. wf-hsig S h)

lemma wf-hsig-lend:
  assumes wf-hsig S h and k < length (lcons (S h))
  shows k < length (lends (S h))
  using assms unfolding wf-hsig-def by simp

lemma wf-hsig-lhs:
  assumes wf-hsig S h and k < length (lcons (S h))
  shows wf-endpt S h (lcons (S h) ! k) (fst (lends (S h) ! k)) (Hit h)
```

using *assms* **unfolding** *wf-hsig-def* **by** *blast*

lemma *wf-hsig-rhs*:
assumes *wf-hsig S h* **and** $k <$ *length* (*lcons* (*S h*))
shows *wf-endpt S h* (*lcons* (*S h*) ! *k*) (*snd* (*lends* (*S h*) ! *k*)) (*Hit h*)
using *assms* **unfolding** *wf-hsig-def* **by** *blast*

theorem *wf-endpt-instE*:
assumes *wf*: *wf-endpt S h ss e A*
and *len*: *length as* = *length ss*
and *args*: ∀ *i* < *length as*. *wt S D G* (*as* ! *i*) (*argTy h* (*ss* ! *i*))
shows *wt S D G* (*instE h as e*) *A*
using *wf*
proof (*induct rule*: *wf-endpt.induct*)
case (*wfEArg i*)
then show *?case*
using *len args* **by** *simp*
next
case (*wfECon k es*)
have *wt S D G* (*PCon h k* (*map* (*instE h as*) *es*)) (*Hit h*)
proof (*rule wt.wtPCon*)
show *k* < *length* (*pcons* (*S h*)) **by** (*rule wfECon*(*1*))
next
show *length* (*map* (*instE h as*) *es*) = *length* (*pcons* (*S h*) ! *k*)
using *wfECon*(*2*) **by** *simp*
next
show ∀ *i* < *length* (*map* (*instE h as*) *es*).
wt S D G ((*map* (*instE h as*) *es*) ! *i*)
(*argTy h* (*pcons* (*S h*) ! *k* ! *i*))
using *wfECon* **by** *simp*
qed
then show *?case* **by** *simp*
qed

corollary *wf-hsig-lhs-typed*:
assumes *wf*: *wf-hsig S h*
and *k*: *k* < *length* (*lcons* (*S h*))
and *len*: *length as* = *length* (*lcons* (*S h*) ! *k*)
and *args*: ∀ *i* < *length as*.
wt S D G (*as* ! *i*) (*argTy h* (*lcons* (*S h*) ! *k* ! *i*))
shows *wt S D G* (*lhsOf S h k as*) (*Hit h*)
unfolding *lhsOf-def*
by (*rule wf-endpt-instE*[*OF wf-hsig-lhs*[*OF wf k*] *len args*])

corollary *wf-hsig-rhs-typed*:
assumes *wf*: *wf-hsig S h*
and *k*: *k* < *length* (*lcons* (*S h*))
and *len*: *length as* = *length* (*lcons* (*S h*) ! *k*)
and *args*: ∀ *i* < *length as*.

```
    wt S D G (as ! i) (argTy h (lcons (S h) ! k ! i))
  shows wt S D G (rhsOf S h k as) (Hit h)
  unfolding rhsOf-def
  by (rule wf-endpt-instE[OF wf-hsig-rhs[OF wf k] len args])

end
theory HIT-Dimensions
  imports HIT-Signature
begin
```

8 Dimension support

```
fun dim-support :: dim ⇒ nat set where
  dim-support D0 = {}
| dim-support D1 = {}
| dim-support (DVar j) = {j}

fun fdims :: tm ⇒ nat set where
  fdims (Var i) = {}
| fdims (Lam A t) = fdims t
| fdims (App t u) = fdims t ∪ fdims u
| fdims (PCon h k as) = ⋃(fdims ‘ set as)
| fdims (LCon h k as d) = ⋃(fdims ‘ set as) ∪ dim-support d
| fdims (HElim h C pms lms t) =
    ⋃(fdims ‘ set pms) ∪ (⋃(fdims ‘ set lms) − {0}) ∪ fdims t
```

The reserved dimension index *0* is bound in path methods, hence it is removed from the free support of the path-method list of an eliminator.

```
lemma fdims-list-subset:
  assumes ∀ i < length xs. fdims (xs ! i) ⊆ D
  shows ⋃(fdims ‘ set xs) ⊆ D
proof
  fix x
  assume x ∈ ⋃(fdims ‘ set xs)
  then obtain a where a: a ∈ set xs and x: x ∈ fdims a
    by blast
  from a obtain i where i: i < length xs and a′: a = xs ! i
    by (auto simp: in-set-conv-nth)
  have x′: x ∈ fdims (xs ! i) using x by (simp add: a′)
  from subsetD[OF assms[rule-format, OF i] x′] show x ∈ D .
qed

theorem wt-fdims:
  assumes wt S D G t A
  shows fdims t ⊆ D
  using assms
proof (induct rule: wt.induct)
  case wtVar
  then show ?case by simp
```

next
case *wtLam*
then show *?case* **by** *simp*
next
case *wtApp*
then show *?case* **by** *simp*
next
case (*wtPCon k h as D G*)
have *args*: $\forall\, i < \mathit{length}\ \mathit{as}.\ \mathit{fdims}\ (\mathit{as}\ !\ i) \subseteq D$
using *wtPCon*(*2*,*3*) **by** *simp*
have $\bigcup(\mathit{fdims}\ `\ \mathit{set}\ \mathit{as}) \subseteq D$
by (*rule fdims-list-subset*[*OF args*])
then show *?case* **by** *simp*
next
case (*wtLCon j D k h as G*)
have *args*: $\forall\, i < \mathit{length}\ \mathit{as}.\ \mathit{fdims}\ (\mathit{as}\ !\ i) \subseteq D$
using *wtLCon*(*3*,*4*) **by** *simp*
have $\bigcup(\mathit{fdims}\ `\ \mathit{set}\ \mathit{as}) \subseteq D$
by (*rule fdims-list-subset*[*OF args*])
then show *?case* **using** *wtLCon*(*1*) **by** *auto*
next
case (*wtElim D G t h pms lms C*)
have *pms*: $\forall\, i < \mathit{length}\ \mathit{pms}.\ \mathit{fdims}\ (\mathit{pms}\ !\ i) \subseteq D$
using *wtElim*(*3*,*5*) **by** *simp*
have *lms*: $\forall\, i < \mathit{length}\ \mathit{lms}.\ \mathit{fdims}\ (\mathit{lms}\ !\ i) \subseteq \mathit{insert}\ 0\ D$
using *wtElim*(*4*,*6*) **by** *simp*
have *pms′*: $\bigcup(\mathit{fdims}\ `\ \mathit{set}\ \mathit{pms}) \subseteq D$
by (*rule fdims-list-subset*[*OF pms*])
have *lms′*: $\bigcup(\mathit{fdims}\ `\ \mathit{set}\ \mathit{lms}) \subseteq \mathit{insert}\ 0\ D$
by (*rule fdims-list-subset*[*OF lms*])
have $\bigcup(\mathit{fdims}\ `\ \mathit{set}\ \mathit{lms}) - \{0\} \subseteq D$
using *lms′* **by** *blast*
with *wtElim*(*2*) *pms′* **show** *?case* **by** *auto*
qed

lemma *dsubstD-fresh*:
assumes $n \notin \mathit{dim\text{-}support}\ e$
shows *dsubstD n d e* = *e*
using *assms* **by** (*cases e*) *auto*

theorem *wt-dsubst-fresh*:
assumes *typed*: *wt S D G t A* **and** *fresh*: $n \notin D$
shows *dsubst S n d t* = *t*
using *typed fresh*
proof (*induct arbitrary*: *n d rule*: *wt.induct*)
case *wtVar*
then show *?case* **by** *simp*
next
case *wtLam*

```
  then show ?case by simp
next
  case wtApp
  then show ?case by simp
next
  case (wtPCon k h as D G)
  have args: ∀ a ∈ set as. dsubst S n d a = a
  proof
    fix a
    assume a ∈ set as
    then obtain i where i: i < length as and a: a = as ! i
      by (auto simp: in-set-conv-nth)
    from wtPCon(2,3) i have dsubst S n d (as ! i) = as ! i
      using wtPCon.prems by blast
    then show dsubst S n d a = a by (simp add: a)
  qed
  then show ?case by (simp add: map-idI)
next
  case (wtLCon j D k h as G)
  have args: ∀ a ∈ set as. dsubst S n d a = a
  proof
    fix a
    assume a ∈ set as
    then obtain i where i: i < length as and a: a = as ! i
      by (auto simp: in-set-conv-nth)
    from wtLCon(3,4) i have dsubst S n d (as ! i) = as ! i
      using wtLCon.prems by blast
    then show dsubst S n d a = a by (simp add: a)
  qed
  have j ≠ n using wtLCon(1) wtLCon.prems by blast
  with args show ?case by (simp add: map-idI)
next
  case (wtElim D G t h pms lms C)
  have t: dsubst S n d t = t
    using wtElim(2) wtElim.prems by blast
  have pms: ∀ m ∈ set pms. dsubst S n d m = m
  proof
    fix m
    assume m ∈ set pms
    then obtain i where i: i < length pms and m: m = pms ! i
      by (auto simp: in-set-conv-nth)
    from wtElim(5) i have dsubst S n d (pms ! i) = pms ! i
      using wtElim.prems by blast
    then show dsubst S n d m = m by (simp add: m)
  qed
  show ?case
  proof (cases n = 0)
    case True
    with t pms show ?thesis by (simp add: map-idI)
```

```
  next
    case False
    have fresh0: n ∉ insert 0 D
      using False wtElim.prems by simp
    have lms: ∀ m ∈ set lms. dsubst S n d m = m
    proof
      fix m
      assume m ∈ set lms
      then obtain i where i: i < length lms and m: m = lms ! i
        by (auto simp: in-set-conv-nth)
      from wtElim(6) i have dsubst S n d (lms ! i) = lms ! i
        using fresh0 by blast
      then show dsubst S n d m = m by (simp add: m)
    qed
    with False t pms show ?thesis by (simp add: map-idI)
  qed
qed

end
theory HIT-Coherence
  imports HIT-Dimensions
begin
```

9 Boundary-coherent elimination

Eliminating an endpoint is defined structurally. Endpoint arguments are recursive occurrences and are therefore eliminated recursively; endpoint point constructors compute with the corresponding point method.

```
primrec elimEnd ::
  sigenv ⇒ nat ⇒ ty ⇒ tm list ⇒ tm list ⇒
    tm list ⇒ endpt ⇒ tm
where
  elimEnd S h C pms lms as (EArg i) =
    HElim h C pms lms (as ! i)
| elimEnd S h C pms lms as (ECon k es) =
    apps (pms ! k)
      (elimArgs h C pms lms (pcons (S h) ! k)
        (map (instE h as) es))

theorem wt-elimEnd:
  assumes endpoint: wf-endpt S h ss e (Hit h)
    and len: length as = length ss
    and args: ∀ i < length as.
      wt S D G (as ! i) (argTy h (ss ! i))
    and plen: length pms = length (pcons (S h))
    and llen: length lms = length (lcons (S h))
    and ptyped: ∀ k < length pms.
      wt S D G (pms ! k) (methodTy h C (pcons (S h) ! k))
```

```
    and ltyped: ∀ k < length lms.
      wt S (insert 0 D) G (lms ! k)
        (methodTy h C (lcons (S h) ! k))
  shows wt S D G (elimEnd S h C pms lms as e) C
  using endpoint
proof cases
  case (wfEArg i)
  have ai: wt S D G (as ! i) (Hit h)
    using wfEArg len args by simp
  have wt S D G (HElim h C pms lms (as ! i)) C
    by (rule wt.wtElim[OF ai plen llen ptyped ltyped])
  then show ?thesis using wfEArg by simp
next
  case (wfECon k es)
  have inst-args: ∀ i < length es.
      wt S D G (instE h as (es ! i))
        (argTy h (pcons (S h) ! k ! i))
  proof (intro allI impI)
    fix i
    assume i: i < length es
    from wfECon(4)[rule-format, OF i]
    show wt S D G (instE h as (es ! i))
        (argTy h (pcons (S h) ! k ! i))
      by (rule wf-endpt-instE[OF - len args])
  qed
  have inst-args': ∀ i < length (pcons (S h) ! k).
      wt S D G ((map (instE h as) es) ! i)
        (argTy h (pcons (S h) ! k ! i))
    using wfECon(3) inst-args by simp
  have method: wt S D G (pms ! k)
      (methodTy h C (pcons (S h) ! k))
    using wfECon(2) plen ptyped by auto
  have recursive:
    ⋀a. wt S D G a (Hit h) ⟹
      wt S D G (HElim h C pms lms a) C
    by (rule wt.wtElim[OF - plen llen ptyped ltyped])
  have wt S D G
      (apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k)
          (map (instE h as) es))) C
    by (rule wt-apps-elimArgs[OF recursive])
      (use wfECon(3) inst-args' method in simp-all)
  then show ?thesis using wfECon by simp
qed

theorem conv-HElim-instE-elimEnd:
  assumes endpoint: wf-endpt S h ss e (Hit h)
  shows conv S
    (HElim h C pms lms (instE h as e))
```

```
    (elimEnd S h C pms lms as e)
  using endpoint
proof cases
  case (wfEArg i)
  then show ?thesis by (simp add: conv-refl)
next
  case (wfECon k es)
  have red S
      (HElim h C pms lms (PCon h k (map (instE h as) es)))
      (apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k)
          (map (instE h as) es)))
    by (rule red.elimPoint[where S = S and h = h and k = k])
      (rule wfECon(2))
  from red-conv[OF this] show ?thesis using wfECon by simp
qed

definition boundary-args :: sigenv ⇒ tm list ⇒ bool where
  boundary-args S as ⟷
    map (dsubst S 0 D0) as = as ∧
    map (dsubst S 0 D1) as = as

lemma typed-terms-boundary-args:
  assumes zero: 0 ∉ D
    and typed: ∀ i < length as. ∃ A. wt S D G (as ! i) A
  shows boundary-args S as
proof −
  have fixed: ∀ a ∈ set as.
      dsubst S 0 D0 a = a ∧ dsubst S 0 D1 a = a
  proof
    fix a
    assume a ∈ set as
    then obtain i where i: i < length as and a: a = as ! i
      by (auto simp: in-set-conv-nth)
    from typed[rule-format, OF i] obtain A where
      wt S D G (as ! i) A by blast
    from wt-dsubst-fresh[OF this zero, of D0]
      wt-dsubst-fresh[OF this zero, of D1]
    show dsubst S 0 D0 a = a ∧ dsubst S 0 D1 a = a
      by (simp add: a)
  qed
  have map (dsubst S 0 D0) as = as
    using fixed by (simp add: map-idI)
  moreover have map (dsubst S 0 D1) as = as
    using fixed by (simp add: map-idI)
  ultimately show ?thesis unfolding boundary-args-def by blast
qed

corollary typed-constructor-args-boundary:
```

assumes $0 \notin D$
and $\forall\, i < \mathit{length}\ \mathit{as}.\ \mathit{wt}\ S\ D\ G\ (\mathit{as}\ !\ i)\ (\mathit{argTy}\ h\ (\mathit{ss}\ !\ i))$
shows *boundary-args S as*
by (*rule typed-terms-boundary-args*[*OF assms*(*1*)]) (*use assms*(*2*) **in** *blast*)

definition *coherent-methods* ::
$\mathit{sigenv} \Rightarrow \mathit{nat} \Rightarrow \mathit{ty} \Rightarrow \mathit{tm\ list} \Rightarrow \mathit{tm\ list} \Rightarrow \mathit{bool}$
where
coherent-methods S h C pms lms $\longleftrightarrow$
$\mathit{length}\ \mathit{pms} = \mathit{length}\ (\mathit{pcons}\ (S\ h)) \wedge$
$\mathit{length}\ \mathit{lms} = \mathit{length}\ (\mathit{lcons}\ (S\ h)) \wedge$
$(\forall\, k < \mathit{length}\ (\mathit{lcons}\ (S\ h)).\ \forall\, \mathit{as}.$
$\mathit{length}\ \mathit{as} = \mathit{length}\ (\mathit{lcons}\ (S\ h)\ !\ k) \longrightarrow$
boundary-args S as $\longrightarrow$
conv S
(*apps* (*dsubst S 0 D0* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
(*elimEnd S h C pms lms as* (*fst* (*lends* (*S h*) ! *k*))) $\wedge$
conv S
(*apps* (*dsubst S 0 D1* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
(*elimEnd S h C pms lms as* (*snd* (*lends* (*S h*) ! *k*))))

lemma *coherent-methods-lengths*:
assumes *coherent-methods S h C pms lms*
shows *length pms* = *length* (*pcons* (*S h*))
and *length lms* = *length* (*lcons* (*S h*))
using *assms* **unfolding** *coherent-methods-def* **by** *blast*+

lemma *coherent-methods-zero*:
assumes *coh*: *coherent-methods S h C pms lms*
and *k*: *k* < *length* (*lcons* (*S h*))
and *len*: *length as* = *length* (*lcons* (*S h*) ! *k*)
and *boundary*: *boundary-args S as*
shows *conv S*
(*apps* (*dsubst S 0 D0* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
(*elimEnd S h C pms lms as* (*fst* (*lends* (*S h*) ! *k*)))
using *coh k len boundary* **unfolding** *coherent-methods-def* **by** *blast*

lemma *coherent-methods-one*:
assumes *coh*: *coherent-methods S h C pms lms*
and *k*: *k* < *length* (*lcons* (*S h*))
and *len*: *length as* = *length* (*lcons* (*S h*) ! *k*)
and *boundary*: *boundary-args S as*
shows *conv S*
(*apps* (*dsubst S 0 D1* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
(*elimEnd S h C pms lms as* (*snd* (*lends* (*S h*) ! *k*)))

using *coh k len boundary* **unfolding** *coherent-methods-def* **by** *blast*

definition *typed-coherent-methods* ::
sigenv ⇒ *nat* ⇒ *ty* ⇒ *tm list* ⇒ *tm list* ⇒ *bool*
where
typed-coherent-methods S h C pms lms ⟷
length pms = *length* (*pcons* (*S h*)) ∧
length lms = *length* (*lcons* (*S h*)) ∧
(∀ *k D G as*.
k < *length* (*lcons* (*S h*)) ⟶
length as = *length* (*lcons* (*S h*) ! *k*) ⟶
0 ∉ *D* ⟶
(∀ *i* < *length as*.
wt S D G (*as* ! *i*) (*argTy h* (*lcons* (*S h*) ! *k* ! *i*))) ⟶
joinable S
(*apps* (*dsubst S 0 D0* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
(*elimEnd S h C pms lms as* (*fst* (*lends* (*S h*) ! *k*))) ∧
joinable S
(*apps* (*dsubst S 0 D1* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
(*elimEnd S h C pms lms as* (*snd* (*lends* (*S h*) ! *k*))))

lemma *typed-coherent-methods-lengths*:
assumes *typed-coherent-methods S h C pms lms*
shows *length pms* = *length* (*pcons* (*S h*))
and *length lms* = *length* (*lcons* (*S h*))
using *assms* **unfolding** *typed-coherent-methods-def* **by** *blast*+

lemma *typed-coherent-methods-zero*:
assumes *coh*: *typed-coherent-methods S h C pms lms*
and *k*: *k* < *length* (*lcons* (*S h*))
and *len*: *length as* = *length* (*lcons* (*S h*) ! *k*)
and *zero*: *0* ∉ *D*
and *args*: ∀ *i* < *length as*.
wt S D G (*as* ! *i*) (*argTy h* (*lcons* (*S h*) ! *k* ! *i*))
shows *joinable S*
(*apps* (*dsubst S 0 D0* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
(*elimEnd S h C pms lms as* (*fst* (*lends* (*S h*) ! *k*)))
using *assms* **unfolding** *typed-coherent-methods-def* **by** *blast*

lemma *typed-coherent-methods-one*:
assumes *coh*: *typed-coherent-methods S h C pms lms*
and *k*: *k* < *length* (*lcons* (*S h*))
and *len*: *length as* = *length* (*lcons* (*S h*) ! *k*)
and *zero*: *0* ∉ *D*
and *args*: ∀ *i* < *length as*.
wt S D G (*as* ! *i*) (*argTy h* (*lcons* (*S h*) ! *k* ! *i*))

shows *joinable S*
(*apps* (*dsubst S 0 D1* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
(*elimEnd S h C pms lms as* (*snd* (*lends* (*S h*) ! *k*)))
using *assms* **unfolding** *typed-coherent-methods-def* **by** *blast*

9.1 The boundary-computation square

theorem *elim-path-face-zero*:
assumes *sig*: *wf-hsig S h*
and *coh*: *coherent-methods S h C pms lms*
and *k*: *k* < *length* (*lcons* (*S h*))
and *len*: *length as* = *length* (*lcons* (*S h*) ! *k*)
and *boundary*: *boundary-args S as*
shows *conv S*
(*HElim h C pms lms* (*mkL S h k as D0*))
(*apps* (*dsubst S 0 D0* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
proof −
let *?e* = *fst* (*lends* (*S h*) ! *k*)
have *endpoint*: *wf-endpt S h* (*lcons* (*S h*) ! *k*) *?e* (*Hit h*)
by (*rule wf-hsig-lhs*[*OF sig k*])
have *left*: *conv S*
(*HElim h C pms lms* (*mkL S h k as D0*))
(*elimEnd S h C pms lms as ?e*)
using *conv-HElim-instE-elimEnd*[*OF endpoint*,
where *C* = *C* **and** *pms* = *pms* **and** *lms* = *lms* **and** *as* = *as*]
by (*simp add*: *lhsOf-def*)
have *right*: *conv S*
(*apps* (*dsubst S 0 D0* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
(*elimEnd S h C pms lms as ?e*)
by (*rule coherent-methods-zero*[*OF coh k len boundary*])
from *conv-trans*[*OF left conv-sym*[*OF right*]] **show** *?thesis* **.**
qed

theorem *elim-path-face-one*:
assumes *sig*: *wf-hsig S h*
and *coh*: *coherent-methods S h C pms lms*
and *k*: *k* < *length* (*lcons* (*S h*))
and *len*: *length as* = *length* (*lcons* (*S h*) ! *k*)
and *boundary*: *boundary-args S as*
shows *conv S*
(*HElim h C pms lms* (*mkL S h k as D1*))
(*apps* (*dsubst S 0 D1* (*lms* ! *k*))
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*))
proof −
let *?e* = *snd* (*lends* (*S h*) ! *k*)
have *endpoint*: *wf-endpt S h* (*lcons* (*S h*) ! *k*) *?e* (*Hit h*)

```
    by (rule wf-hsig-rhs[OF sig k])
  have left: conv S
      (HElim h C pms lms (mkL S h k as D1))
      (elimEnd S h C pms lms as ?e)
    using conv-HElim-instE-elimEnd[OF endpoint,
        where C = C and pms = pms and lms = lms and as = as]
    by (simp add: rhsOf-def)
  have right: conv S
      (apps (dsubst S 0 D1 (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as))
      (elimEnd S h C pms lms as ?e)
    by (rule coherent-methods-one[OF coh k len boundary])
  from conv-trans[OF left conv-sym[OF right]] show ?thesis .
qed

theorem elim-path-boundary-square:
  assumes wf-hsig S h
    and coherent-methods S h C pms lms
    and k < length (lcons (S h))
    and length as = length (lcons (S h) ! k)
    and boundary-args S as
  shows conv S
      (HElim h C pms lms (mkL S h k as D0))
      (apps (dsubst S 0 D0 (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as))
    and conv S
      (HElim h C pms lms (mkL S h k as D1))
      (apps (dsubst S 0 D1 (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as))
proof −
  show conv S
      (HElim h C pms lms (mkL S h k as D0))
      (apps (dsubst S 0 D0 (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as))
    by (rule elim-path-face-zero[OF assms])
  show conv S
      (HElim h C pms lms (mkL S h k as D1))
      (apps (dsubst S 0 D1 (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as))
    by (rule elim-path-face-one[OF assms])
qed

corollary typed-elim-path-boundary-square:
  assumes sig: wf-hsig S h
    and coh: typed-coherent-methods S h C pms lms
    and k: k < length (lcons (S h))
    and len: length as = length (lcons (S h) ! k)
    and zero: 0 ∉ D
    and args: ∀ i < length as.
```

```
    wt S D G (as ! i) (argTy h (lcons (S h) ! k ! i))
  shows conv S
    (HElim h C pms lms (mkL S h k as D0))
    (apps (dsubst S 0 D0 (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as))
   and conv S
    (HElim h C pms lms (mkL S h k as D1))
    (apps (dsubst S 0 D1 (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as))
proof −
  show conv S
    (HElim h C pms lms (mkL S h k as D0))
    (apps (dsubst S 0 D0 (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as))
  proof −
    let ?e = fst (lends (S h) ! k)
    have endpoint: wf-endpt S h (lcons (S h) ! k) ?e (Hit h)
      by (rule wf-hsig-lhs[OF sig k])
    have left: conv S
      (HElim h C pms lms (mkL S h k as D0))
      (elimEnd S h C pms lms as ?e)
      using conv-HElim-instE-elimEnd[OF endpoint,
        where C = C and pms = pms and lms = lms and as = as]
      by (simp add: lhsOf-def)
    have right-join: joinable S
      (apps (dsubst S 0 D0 (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as))
      (elimEnd S h C pms lms as ?e)
      by (rule typed-coherent-methods-zero[OF coh k len zero args])
    have right: conv S
      (apps (dsubst S 0 D0 (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as))
      (elimEnd S h C pms lms as ?e)
      by (rule joinable-conv[OF right-join])
    from conv-trans[OF left conv-sym[OF right]] show ?thesis .
  qed
  show conv S
    (HElim h C pms lms (mkL S h k as D1))
    (apps (dsubst S 0 D1 (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as))
  proof −
    let ?e = snd (lends (S h) ! k)
    have endpoint: wf-endpt S h (lcons (S h) ! k) ?e (Hit h)
      by (rule wf-hsig-rhs[OF sig k])
    have left: conv S
      (HElim h C pms lms (mkL S h k as D1))
      (elimEnd S h C pms lms as ?e)
      using conv-HElim-instE-elimEnd[OF endpoint,
        where C = C and pms = pms and lms = lms and as = as]
```

```
    by (simp add: rhsOf-def)
  have right-join: joinable S
    (apps (dsubst S 0 D1 (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as))
    (elimEnd S h C pms lms as ?e)
    by (rule typed-coherent-methods-one[OF coh k len zero args])
  have right: conv S
    (apps (dsubst S 0 D1 (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as))
    (elimEnd S h C pms lms as ?e)
    by (rule joinable-conv[OF right-join])
  from conv-trans[OF left conv-sym[OF right]] show ?thesis .
 qed
qed
```

The raw typing relation is useful for syntax-directed metatheory. The local *checked* predicate requires every constructor signature to be well formed, every subterm to be checked, and every eliminator to carry path methods whose typed boundary instances are joinable. The *cwt* judgment is deliberately a hereditary semantic safety predicate, not an algorithmic type-checking judgment: every reduct must pass that local check. This makes the safety invariant explicit rather than assuming that an unindexed structural predicate is stable under arbitrary binder substitution. A later theory supplies a finite open-term boundary criterion and proves it sound for all typed constructor instances, together with an equivalent canonical-normal-form comparison criterion.

```
primrec checked :: sigenv ⇒ tm ⇒ bool where
  checked S (Var i) = True
| checked S (Lam A t) = checked S t
| checked S (App t u) = (checked S t ∧ checked S u)
| checked S (PCon h k as) =
    (wf-hsig S h ∧ list-all (checked S) as)
| checked S (LCon h k as d) =
    (wf-hsig S h ∧ list-all (checked S) as)
| checked S (HElim h C pms lms t) =
    (wf-hsig S h ∧ typed-coherent-methods S h C pms lms ∧
      list-all (checked S) pms ∧ list-all (checked S) lms ∧
      checked S t)

definition hereditarily-checked :: sigenv ⇒ tm ⇒ bool where
  hereditarily-checked S t ⟷
    (∀ u. reds S t u ⟶ checked S u)

definition cwt ::
  sigenv ⇒ nat set ⇒ ty list ⇒ tm ⇒ ty ⇒ bool
where
  cwt S D G t A ⟷
    0 ∉ D ∧ wt S D G t A ∧ hereditarily-checked S t
```

lemma *cwt-wt*: *cwt S D G t A* $\Longrightarrow$ *wt S D G t A*
unfolding *cwt-def* **by** *blast*

lemma *cwt-checked*: *cwt S D G t A* $\Longrightarrow$ *checked S t*
unfolding *cwt-def hereditarily-checked-def* **by** *blast*

lemma *cwt-dimension-context*: *cwt S D G t A* $\Longrightarrow$ *0* $\notin$ *D*
unfolding *cwt-def* **by** *blast*

lemma *cwt-hereditarily-checked*:
cwt S D G t A $\Longrightarrow$ *hereditarily-checked S t*
unfolding *cwt-def* **by** *blast*

lemma *cwtI*:
assumes *0* $\notin$ *D* **and** *wt S D G t A*
and *hereditarily-checked S t*
shows *cwt S D G t A*
using *assms* **unfolding** *cwt-def* **by** *blast*

lemma *hereditarily-checked-step*:
assumes *source*: *hereditarily-checked S t* **and** *step*: *red S t u*
shows *hereditarily-checked S u*
proof (*unfold hereditarily-checked-def*, *intro allI impI*)
fix *v*
assume *uv*: *reds S u v*
have *tu*: *reds S t u*
by (*rule r-into-rtranclp*[**where** *r* = *red S*, *OF step*])
have *reds S t v* **by** (*rule rtranclp-trans*[*OF tu uv*])
with *source* **show** *checked S v*
unfolding *hereditarily-checked-def* **by** *blast*
qed

lemma *hereditarily-checked-steps*:
assumes *source*: *hereditarily-checked S t* **and** *steps*: *reds S t u*
shows *hereditarily-checked S u*
proof (*unfold hereditarily-checked-def*, *intro allI impI*)
fix *v*
assume *uv*: *reds S u v*
have *reds S t v* **by** (*rule rtranclp-trans*[*OF steps uv*])
with *source* **show** *checked S v*
unfolding *hereditarily-checked-def* **by** *blast*
qed

lemma *checked-PCon-wf*:
checked S (*PCon h k as*) $\Longrightarrow$ *wf-hsig S h*
by *simp*

lemma *checked-LCon-wf*:

checked S (LCon h k as d) ⟹ *wf-hsig S h*
by *simp*

lemma *checked-HElim-coherent*:
checked S (HElim h C pms lms t) ⟹
typed-coherent-methods S h C pms lms
by *simp*

end
theory *HIT-Substitution*
imports *HIT-Coherence*
begin

10 Structural properties of typing

fun *ins-ty* :: *nat* ⇒ *ty* ⇒ *ty list* ⇒ *ty list* **where**
ins-ty 0 A G = *A* # *G*
| *ins-ty* (*Suc n*) *A* [] = [*A*]
| *ins-ty* (*Suc n*) *A* (*B* # *G*) = *B* # *ins-ty n A G*

lemma *length-ins-ty* [*simp*]:
assumes *n* ≤ *length G*
shows *length* (*ins-ty n A G*) = *Suc* (*length G*)
using *assms*
proof (*induct n arbitrary*: *G*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then obtain *B G'* **where** *G*: *G* = *B* # *G'*
by (*cases G*) *auto*
from *Suc.prems G* **have** *n* ≤ *length G'* **by** *simp*
from *Suc.hyps*[*OF this*] **show** *?case* **by** (*simp add*: *G*)
qed

lemma *nth-ins-ty-lt*:
assumes *i* < *n* **and** *n* ≤ *length G*
shows *ins-ty n A G* ! *i* = *G* ! *i*
using *assms*
proof (*induct n arbitrary*: *G i*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then obtain *B G'* **where** *G*: *G* = *B* # *G'*
by (*cases G*) *auto*
show *?case*
proof (*cases i*)
case *0*

```
    then show ?thesis by (simp add: G)
  next
    case (Suc i')
    from Suc.prems G Suc have i' < n and n ≤ length G'
      by simp-all
    from Suc.hyps[OF this] Suc show ?thesis by (simp add: G)
  qed
qed

lemma nth-ins-ty-eq:
  assumes n ≤ length G
  shows ins-ty n A G ! n = A
  using assms
proof (induct n arbitrary: G)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then obtain B G' where G: G = B # G'
    by (cases G) auto
  from Suc.prems G have n ≤ length G' by simp
  from Suc.hyps[OF this] show ?case by (simp add: G)
qed

lemma nth-ins-ty-ge:
  assumes n ≤ i and i < length G
  shows ins-ty n A G ! Suc i = G ! i
  using assms
proof (induct n arbitrary: G i)
  case 0
  then show ?case by simp
next
  case (Suc n)
  then obtain B G' where G: G = B # G'
    by (cases G) auto
  from Suc.prems(1) obtain i' where i: i = Suc i'
    by (cases i) auto
  from Suc.prems G i have n ≤ i' and i' < length G'
    by simp-all
  from Suc.hyps[OF this] show ?case by (simp add: G i)
qed

lemma wt-dimension-mono:
  assumes deriv: wt S D G t A and sub: D ⊆ D'
  shows wt S D' G t A
  using deriv sub
proof (induct arbitrary: D' rule: wt.induct)
  case (wtVar i G D)
  then show ?case by (auto intro: wt.wtVar)
```

```
next
  case (wtLam D A G t B)
  from wtLam(2)[OF wtLam.prems] show ?case
    by (rule wt.wtLam)
next
  case (wtApp D G t A B u)
  from wtApp(2)[OF wtApp.prems] wtApp(4)[OF wtApp.prems]
  show ?case by (rule wt.wtApp)
next
  case (wtPCon k h as D G)
  have args: ∀ i < length as.
      wt S D' G (as ! i) (argTy h (pcons (S h) ! k ! i))
    using wtPCon(3) wtPCon.prems by blast
  show ?case by (rule wt.wtPCon[OF wtPCon(1,2) args])
next
  case (wtLCon j D k h as G)
  have args: ∀ i < length as.
      wt S D' G (as ! i) (argTy h (lcons (S h) ! k ! i))
    using wtLCon(4) wtLCon.prems by blast
  have j ∈ D' using wtLCon(1) wtLCon.prems by blast
  then show ?case by (rule wt.wtLCon[OF - wtLCon(2,3) args])
next
  case (wtElim D G t h pms lms C)
  have t: wt S D' G t (Hit h)
    by (rule wtElim(2)[OF wtElim.prems])
  have pms: ∀ k < length pms.
      wt S D' G (pms ! k) (methodTy h C (pcons (S h) ! k))
    using wtElim(5) wtElim.prems by blast
  have sub0: insert 0 D ⊆ insert 0 D'
    using wtElim.prems by blast
  have lms: ∀ k < length lms.
      wt S (insert 0 D') G (lms ! k) (methodTy h C (lcons (S h) ! k))
    using wtElim(6) sub0 by blast
  show ?case by (rule wt.wtElim[OF t wtElim(3,4) pms lms])
qed

theorem wt-lift:
  assumes deriv: wt S D G t A and n: n ≤ length G
  shows wt S D (ins-ty n B G) (lift n t) A
  using deriv n
proof (induct arbitrary: n B rule: wt.induct)
  case (wtVar i G D)
  show ?case
  proof (cases i < n)
    case True
    have nth: ins-ty n B G ! i = G ! i
      by (rule nth-ins-ty-lt[OF True wtVar.prems])
    have i < length (ins-ty n B G)
      using wtVar(1) wtVar.prems by simp
```

```
    then have wt S D (ins-ty n B G) (Var i)
        (ins-ty n B G ! i)
      by (rule wt.wtVar)
    then have wt S D (ins-ty n B G) (Var i) (G ! i)
      using nth by simp
    with True show ?thesis by simp
  next
    case False
    have nth: ins-ty n B G ! Suc i = G ! i
      by (rule nth-ins-ty-ge) (use False wtVar in auto)
    have Suc i < length (ins-ty n B G)
      using wtVar(1) wtVar.prems by simp
    then have wt S D (ins-ty n B G) (Var (Suc i))
        (ins-ty n B G ! Suc i)
      by (rule wt.wtVar)
    then have wt S D (ins-ty n B G) (Var (Suc i)) (G ! i)
      using nth by simp
    with False show ?thesis by simp
  qed
next
  case (wtLam D A G t C)
  have Suc n ≤ length (A # G)
    using wtLam.prems by simp
  from wtLam(2)[OF this, of B]
  have wt S D (A # ins-ty n B G) (lift (Suc n) t) C
    by simp
  then have wt S D (ins-ty n B G) (Lam A (lift (Suc n) t)) (Fun A C)
    by (rule wt.wtLam)
  then show ?case by simp
next
  case (wtApp D G t A C u)
  from wtApp(2)[OF wtApp.prems, of B]
    wtApp(4)[OF wtApp.prems, of B]
  show ?case by (auto intro: wt.wtApp)
next
  case (wtPCon k h as D G)
  have wt S D (ins-ty n B G) (PCon h k (map (lift n) as)) (Hit h)
  proof (rule wt.wtPCon)
    show k < length (pcons (S h)) by (rule wtPCon(1))
  next
    show length (map (lift n) as) = length (pcons (S h) ! k)
      using wtPCon(2) by simp
  next
    show ∀ i < length (map (lift n) as).
        wt S D (ins-ty n B G) ((map (lift n) as) ! i)
          (argTy h (pcons (S h) ! k ! i))
      using wtPCon by simp
  qed
  then show ?case by simp
```

next
case (*wtLCon j D k h as G*)
have *wt S D* (*ins-ty n B G*)
(*LCon h k* (*map* (*lift n*) *as*) (*DVar j*)) (*Hit h*)
proof (*rule wt.wtLCon*)
show $j \in D$ **by** (*rule wtLCon*(*1*))
next
show $k <$ *length* (*lcons* (*S h*)) **by** (*rule wtLCon*(*2*))
next
show *length* (*map* (*lift n*) *as*) = *length* (*lcons* (*S h*) ! *k*)
using *wtLCon*(*3*) **by** *simp*
next
show $\forall i <$ *length* (*map* (*lift n*) *as*).
wt S D (*ins-ty n B G*) ((*map* (*lift n*) *as*) ! *i*)
(*argTy h* (*lcons* (*S h*) ! *k* ! *i*))
using *wtLCon* **by** *simp*
qed
then show *?case* **by** *simp*
next
case (*wtElim D G t h pms lms C*)
have *wt S D* (*ins-ty n B G*)
(*HElim h C* (*map* (*lift n*) *pms*) (*map* (*lift n*) *lms*) (*lift n t*)) *C*
proof (*rule wt.wtElim*)
show *wt S D* (*ins-ty n B G*) (*lift n t*) (*Hit h*)
using *wtElim* **by** *simp*
next
show *length* (*map* (*lift n*) *pms*) = *length* (*pcons* (*S h*))
using *wtElim* **by** *simp*
next
show *length* (*map* (*lift n*) *lms*) = *length* (*lcons* (*S h*))
using *wtElim* **by** *simp*
next
show $\forall k <$ *length* (*map* (*lift n*) *pms*).
wt S D (*ins-ty n B G*) ((*map* (*lift n*) *pms*) ! *k*)
(*methodTy h C* (*pcons* (*S h*) ! *k*))
using *wtElim* **by** *simp*
next
show $\forall k <$ *length* (*map* (*lift n*) *lms*).
wt S (*insert 0 D*) (*ins-ty n B G*) ((*map* (*lift n*) *lms*) ! *k*)
(*methodTy h C* (*lcons* (*S h*) ! *k*))
using *wtElim* **by** *simp*
qed
then show *?case* **by** *simp*
qed

11 Term substitution

lemma *nth-ins-ty-gt*:
assumes $n < i$ **and** $i <$ *Suc* (*length G*) **and** $n \leq$ *length G*

```
  shows ins-ty n A G ! i = G ! (i − 1)
proof −
  from assms(1) obtain j where i: i = Suc j and n: n ≤ j
    by (cases i) auto
  from assms(2) i have j: j < length G by simp
  from nth-ins-ty-ge[OF n j, of A] show ?thesis
    by (simp add: i)
qed

theorem wt-subst-aux:
  assumes deriv: wt S D H t B
    and ctx: H = ins-ty n A G
    and replacement: wt S D G s A
    and n: n ≤ length G
  shows wt S D G (subst n s t) B
  using deriv ctx replacement n
proof (induct arbitrary: n A G s rule: wt.induct)
  case (wtVar i H D)
  have ilen: i < Suc (length G)
    using wtVar(1) wtVar.prems(1,3) by simp
  show ?case
  proof (cases i < n)
    case True
    have nth: H ! i = G ! i
      using wtVar.prems(1,3) True nth-ins-ty-lt by blast
    have i < length G using True wtVar.prems(3) by linarith
    then have wt S D G (Var i) (G ! i) by (rule wt.wtVar)
    with True nth show ?thesis by simp
  next
    case False
    have not-lt: ¬ i < n by (rule False)
    show ?thesis
    proof (cases i = n)
      case True
      have H ! i = A
        using wtVar.prems(1,3) True nth-ins-ty-eq by blast
      with True wtVar.prems(2) show ?thesis by simp
    next
      case False
      have ni: n < i using not-lt False by linarith
      have nth: H ! i = G ! (i − 1)
        using wtVar.prems(1,3) ilen ni nth-ins-ty-gt by blast
      have i − 1 < length G using ilen ni by linarith
      then have wt S D G (Var (i − 1)) (G ! (i − 1))
        by (rule wt.wtVar)
      with not-lt False nth show ?thesis by simp
    qed
  qed
next
```

```
  case (wtLam D B0 H t C)
  have source: B0 # H = ins-ty (Suc n) A (B0 # G)
    using wtLam.prems(1) by simp
  have n': Suc n ≤ length (B0 # G)
    using wtLam.prems(3) by simp
  have lifted: wt S D (B0 # G) (lift 0 s) A
    using wt-lift[OF wtLam.prems(2), of 0 B0] by simp
  from wtLam(2)[OF source lifted n']
  have wt S D (B0 # G) (subst (Suc n) (lift 0 s) t) C .
  then have wt S D G (Lam B0 (subst (Suc n) (lift 0 s) t)) (Fun B0 C)
    by (rule wt.wtLam)
  then show ?case by simp
next
  case (wtApp D H t A0 B0 u)
  have t: wt S D G (subst n s t) (Fun A0 B0)
    by (rule wtApp(2)[OF wtApp.prems])
  have u: wt S D G (subst n s u) A0
    by (rule wtApp(4)[OF wtApp.prems])
  from wt.wtApp[OF t u] show ?case by simp
next
  case (wtPCon k h as D H)
  have args: ∀ i < length as.
      wt S D G (subst n s (as ! i))
        (argTy h (pcons (S h) ! k ! i))
    using wtPCon(3) wtPCon.prems by blast
  have wt S D G (PCon h k (map (subst n s) as)) (Hit h)
  proof (rule wt.wtPCon)
    show k < length (pcons (S h)) by (rule wtPCon(1))
  next
    show length (map (subst n s) as) = length (pcons (S h) ! k)
      using wtPCon(2) by simp
  next
    show ∀ i < length (map (subst n s) as).
        wt S D G ((map (subst n s) as) ! i)
          (argTy h (pcons (S h) ! k ! i))
      using args by simp
  qed
  then show ?case by simp
next
  case (wtLCon j D k h as H)
  have args: ∀ i < length as.
      wt S D G (subst n s (as ! i))
        (argTy h (lcons (S h) ! k ! i))
    using wtLCon(4) wtLCon.prems by blast
  have wt S D G (LCon h k (map (subst n s) as) (DVar j)) (Hit h)
  proof (rule wt.wtLCon)
    show j ∈ D by (rule wtLCon(1))
  next
    show k < length (lcons (S h)) by (rule wtLCon(2))
```

```
  next
    show length (map (subst n s) as) = length (lcons (S h) ! k)
      using wtLCon(3) by simp
  next
    show ∀ i < length (map (subst n s) as).
        wt S D G ((map (subst n s) as) ! i)
          (argTy h (lcons (S h) ! k ! i))
      using args by simp
  qed
  then show ?case by simp
next
  case (wtElim D H t h pms lms C)
  have t: wt S D G (subst n s t) (Hit h)
    by (rule wtElim(2)[OF wtElim.prems])
  have pms: ∀ k < length pms.
      wt S D G (subst n s (pms ! k))
        (methodTy h C (pcons (S h) ! k))
    using wtElim(5) wtElim.prems by blast
  have replacement0: wt S (insert 0 D) G s A
    by (rule wt-dimension-mono[OF wtElim.prems(2)]) blast
  have lms: ∀ k < length lms.
      wt S (insert 0 D) G (subst n s (lms ! k))
        (methodTy h C (lcons (S h) ! k))
    using wtElim(6) wtElim.prems(1,3) replacement0 by blast
  have wt S D G
      (HElim h C (map (subst n s) pms) (map (subst n s) lms)
        (subst n s t)) C
  proof (rule wt.wtElim[OF t])
    show length (map (subst n s) pms) = length (pcons (S h))
      using wtElim(3) by simp
  next
    show length (map (subst n s) lms) = length (lcons (S h))
      using wtElim(4) by simp
  next
    show ∀ k < length (map (subst n s) pms).
        wt S D G ((map (subst n s) pms) ! k)
          (methodTy h C (pcons (S h) ! k))
      using pms by simp
  next
    show ∀ k < length (map (subst n s) lms).
        wt S (insert 0 D) G ((map (subst n s) lms) ! k)
          (methodTy h C (lcons (S h) ! k))
      using lms by simp
  qed
  then show ?case by simp
qed

corollary wt-subst:
  assumes wt S D (ins-ty n A G) t B
```

and *wt S D G s A*
and $n \leq$ *length G*
shows *wt S D G* (*subst n s t*) *B*
by (*rule wt-subst-aux*[*OF assms*(*1*) *refl assms*(*2*,*3*)])

12 Dimension substitution

fun *dim-set* :: *dim* ⇒ *nat set* **where**
dim-set D0 = {}
| *dim-set D1* = {}
| *dim-set* (*DVar j*) = {*j*}

definition *dctx-subst* :: *nat* ⇒ *dim* ⇒ *nat set* ⇒ *nat set* **where**
dctx-subst n d D = (*D* − {*n*}) ∪ *dim-set d*

lemma *dctx-subst-insert-nonzero*:
assumes $n \neq 0$
shows *dctx-subst n d* (*insert 0 D*) = *insert 0* (*dctx-subst n d D*)
using *assms* **unfolding** *dctx-subst-def* **by** *auto*

lemma *dctx-subst-bound-subset*:
insert 0 D ⊆ *insert 0* (*dctx-subst 0 d D*)
unfolding *dctx-subst-def* **by** *auto*

theorem *wt-dimension-subst*:
assumes *deriv*: *wt S D G t A* **and** *wf*: *wf-sigenv S*
shows *wt S* (*dctx-subst n d D*) *G* (*dsubst S n d t*) *A*
using *deriv wf*
proof (*induct arbitrary*: *n d rule*: *wt.induct*)
case (*wtVar i G D*)
then show *?case* **by** (*auto intro*: *wt.wtVar*)
next
case (*wtLam D A G t B*)
from *wtLam*(*2*)[*OF wtLam.prems*, *of n d*]
show *?case* **by** (*auto intro*: *wt.wtLam*)
next
case (*wtApp D G t A B u*)
from *wtApp*(*2*)[*OF wtApp.prems*, *of n d*]
wtApp(*4*)[*OF wtApp.prems*, *of n d*]
show *?case* **by** (*auto intro*: *wt.wtApp*)
next
case (*wtPCon k h as D G*)
have *args*: ∀ *i* < *length as*.
wt S (*dctx-subst n d D*) *G* (*dsubst S n d* (*as* ! *i*))
(*argTy h* (*pcons* (*S h*) ! *k* ! *i*))
using *wtPCon*(*3*) *wtPCon.prems* **by** *blast*
have *wt S* (*dctx-subst n d D*) *G*
(*PCon h k* (*map* (*dsubst S n d*) *as*)) (*Hit h*)
proof (*rule wt.wtPCon*)

```
    show k < length (pcons (S h)) by (rule wtPCon(1))
  next
    show length (map (dsubst S n d) as) = length (pcons (S h) ! k)
      using wtPCon(2) by simp
  next
    show ∀ i < length (map (dsubst S n d) as).
        wt S (dctx-subst n d D) G ((map (dsubst S n d) as) ! i)
          (argTy h (pcons (S h) ! k ! i))
      using args by simp
  qed
  then show ?case by simp
next
  case (wtLCon j D k h as G)
  let ?D' = dctx-subst n d D
  let ?as' = map (dsubst S n d) as
  have args: ∀ i < length as.
      wt S ?D' G (dsubst S n d (as ! i))
        (argTy h (lcons (S h) ! k ! i))
    using wtLCon(4) wtLCon.prems by blast
  have args': ∀ i < length ?as'.
      wt S ?D' G (?as' ! i) (argTy h (lcons (S h) ! k ! i))
    using args by simp
  have len: length ?as' = length (lcons (S h) ! k)
    using wtLCon(3) by simp
  have sig: wf-hsig S h
    using wtLCon.prems unfolding wf-sigenv-def by blast
  show ?case
  proof (cases j = n)
    case False
    have j ∈ ?D'
      using wtLCon(1) False unfolding dctx-subst-def by auto
    then have wt S ?D' G (LCon h k ?as' (DVar j)) (Hit h)
      by (rule wt.wtLCon[OF - wtLCon(2) len args'])
    with False show ?thesis by simp
  next
    case True
    show ?thesis
    proof (cases d)
      case D0
      have wt S ?D' G (lhsOf S h k ?as') (Hit h)
        by (rule wf-hsig-lhs-typed[OF sig wtLCon(2) len args'])
      with True D0 show ?thesis by simp
    next
      case D1
      have wt S ?D' G (rhsOf S h k ?as') (Hit h)
        by (rule wf-hsig-rhs-typed[OF sig wtLCon(2) len args'])
      with True D1 show ?thesis by simp
    next
      case (DVar m)
```

```
      have m ∈ ?D'
        unfolding dctx-subst-def DVar by simp
      then have wt S ?D' G (LCon h k ?as' (DVar m)) (Hit h)
        by (rule wt.wtLCon[OF - wtLCon(2) len args'])
      with True DVar show ?thesis by simp
    qed
  qed
next
  case (wtElim D G t h pms lms C)
  let ?D' = dctx-subst n d D
  let ?pms' = map (dsubst S n d) pms
  let ?lms' = if n = 0 then lms else map (dsubst S n d) lms
  have t: wt S ?D' G (dsubst S n d t) (Hit h)
    by (rule wtElim(2)[OF wtElim.prems])
  have pms: ∀ k < length pms.
      wt S ?D' G (dsubst S n d (pms ! k))
        (methodTy h C (pcons (S h) ! k))
    using wtElim(5) wtElim.prems by blast
  have pms': ∀ k < length ?pms'.
      wt S ?D' G (?pms' ! k) (methodTy h C (pcons (S h) ! k))
    using pms by simp
  have lms': ∀ k < length ?lms'.
      wt S (insert 0 ?D') G (?lms' ! k)
        (methodTy h C (lcons (S h) ! k))
  proof (cases n = 0)
    case True
    have source: ∀ k < length lms.
        wt S (insert 0 D) G (lms ! k)
          (methodTy h C (lcons (S h) ! k))
      using wtElim(6) by blast
    have sub: insert 0 D ⊆ insert 0 ?D'
      using True dctx-subst-bound-subset by simp
    have ∀ k < length lms.
        wt S (insert 0 ?D') G (lms ! k)
          (methodTy h C (lcons (S h) ! k))
      using source
      by (metis wt-dimension-mono sub)
    with True show ?thesis by simp
  next
    case False
    have target: ∀ k < length lms.
        wt S (dctx-subst n d (insert 0 D)) G
          (dsubst S n d (lms ! k))
          (methodTy h C (lcons (S h) ! k))
      using wtElim(6) wtElim.prems by blast
    have eq: dctx-subst n d (insert 0 D) = insert 0 ?D'
      by (rule dctx-subst-insert-nonzero[OF False])
    from target False eq show ?thesis by simp
  qed
```

```
  have wt S ?D' G (HElim h C ?pms' ?lms' (dsubst S n d t)) C
  proof (rule wt.wtElim[OF t])
    show length ?pms' = length (pcons (S h))
      using wtElim(3) by simp
  next
    show length ?lms' = length (lcons (S h))
      using wtElim(4) by simp
  next
    show ∀ k < length ?pms'.
        wt S ?D' G (?pms' ! k) (methodTy h C (pcons (S h) ! k))
      by (rule pms')
  next
    show ∀ k < length ?lms'.
        wt S (insert 0 ?D') G (?lms' ! k)
          (methodTy h C (lcons (S h) ! k))
      by (rule lms')
  qed
  then show ?case by simp
qed

end
theory HIT-Preservation
  imports HIT-Substitution
begin
```

13 Subject reduction

```
lemma all-nth-list-update:
  assumes all: ∀ j < length xs. P j (xs ! j)
    and i: i < length xs
    and replacement: P i x
  shows ∀ j < length (xs[i := x]). P j ((xs[i := x]) ! j)
  using assms by (auto simp: nth-list-update)

theorem subject-reduction:
  assumes wf: wf-sigenv S
    and step: red S t u
    and typing: wt S D G t A
  shows wt S D G u A
  using step typing
proof (induct arbitrary: D G A rule: red.induct)
  case (beta u B t)
  then obtain C where
    body: wt S D (B # G) t C
    and arg: wt S D G u B
    and A: A = C
    by (auto elim: wt.cases)
  have wt S D G (subst 0 u t) C
    by (rule wt-subst[OF - arg, of 0]) (use body in simp-all)
```

```
  with A show ?case by simp
next
  case (elimPoint k h C pms lms as)
  from elimPoint.prems obtain
    scrut: wt S D G (PCon h k as) (Hit h)
    and plen: length pms = length (pcons (S h))
    and llen: length lms = length (lcons (S h))
    and ptyped: ∀ i < length pms.
      wt S D G (pms ! i) (methodTy h C (pcons (S h) ! i))
    and ltyped: ∀ i < length lms.
      wt S (insert 0 D) G (lms ! i) (methodTy h C (lcons (S h) ! i))
    and A: A = C
    by (auto elim: wt.cases)
  from scrut obtain
    len: length as = length (pcons (S h) ! k)
    and args: ∀ i < length as.
      wt S D G (as ! i) (argTy h (pcons (S h) ! k ! i))
    by (auto elim: wt.cases)
  have method: wt S D G (pms ! k) (methodTy h C (pcons (S h) ! k))
    using elimPoint(1) plen ptyped by auto
  have args': ∀ i < length (pcons (S h) ! k).
      wt S D G (as ! i) (argTy h (pcons (S h) ! k ! i))
    using len args by simp
  have recursive:
    ⋀a. wt S D G a (Hit h) ⟹ wt S D G (HElim h C pms lms a) C
    by (rule wt.wtElim[OF - plen llen ptyped ltyped])
  have wt S D G
      (apps (pms ! k) (elimArgs h C pms lms (pcons (S h) ! k) as)) C
    by (rule wt-apps-elimArgs[OF recursive len args' method])
  with A show ?case by simp
next
  case (elimPath k h C pms lms as j)
  from elimPath.prems obtain
    scrut: wt S D G (LCon h k as (DVar j)) (Hit h)
    and plen: length pms = length (pcons (S h))
    and llen: length lms = length (lcons (S h))
    and ptyped: ∀ i < length pms.
      wt S D G (pms ! i) (methodTy h C (pcons (S h) ! i))
    and ltyped: ∀ i < length lms.
      wt S (insert 0 D) G (lms ! i) (methodTy h C (lcons (S h) ! i))
    and A: A = C
    by (auto elim: wt.cases)
  from scrut obtain
    jD: j ∈ D
    and len: length as = length (lcons (S h) ! k)
    and args: ∀ i < length as.
      wt S D G (as ! i) (argTy h (lcons (S h) ! k ! i))
    by (auto elim: wt.cases)
  have method: wt S (insert 0 D) G (lms ! k)
```

```
    (methodTy h C (lcons (S h) ! k))
  using elimPath(1) llen ltyped by auto
have args0: ∀ i < length as.
    wt S (insert 0 D) G (as ! i) (argTy h (lcons (S h) ! k ! i))
  using args by (metis subset-insertI wt-dimension-mono)
have args0': ∀ i < length (lcons (S h) ! k).
    wt S (insert 0 D) G (as ! i) (argTy h (lcons (S h) ! k ! i))
  using len args0 by simp
have ptyped0: ∀ i < length pms.
    wt S (insert 0 D) G (pms ! i) (methodTy h C (pcons (S h) ! i))
  using ptyped by (metis subset-insertI wt-dimension-mono)
have recursive:
  ⋀a. wt S (insert 0 D) G a (Hit h) ⟹
    wt S (insert 0 D) G (HElim h C pms lms a) C
proof −
  fix a
  assume a: wt S (insert 0 D) G a (Hit h)
  show wt S (insert 0 D) G (HElim h C pms lms a) C
  proof (rule wt.wtElim)
    show wt S (insert 0 D) G a (Hit h) by (rule a)
  next
    show length pms = length (pcons (S h)) by (rule plen)
  next
    show length lms = length (lcons (S h)) by (rule llen)
  next
    show ∀ i < length pms.
        wt S (insert 0 D) G (pms ! i)
          (methodTy h C (pcons (S h) ! i))
      by (rule ptyped0)
  next
    show ∀ i < length lms.
        wt S (insert 0 (insert 0 D)) G (lms ! i)
          (methodTy h C (lcons (S h) ! i))
      using ltyped by simp
  qed
qed
have method': wt S (dctx-subst 0 (DVar j) (insert 0 D)) G
    (dsubst S 0 (DVar j) (lms ! k))
    (methodTy h C (lcons (S h) ! k))
  by (rule wt-dimension-subst[OF method wf])
have methodD: wt S D G (dsubst S 0 (DVar j) (lms ! k))
    (methodTy h C (lcons (S h) ! k))
proof (rule wt-dimension-mono[OF method'])
  show dctx-subst 0 (DVar j) (insert 0 D) ⊆ D
    using jD unfolding dctx-subst-def by auto
qed
have recursiveD:
  ⋀a. wt S D G a (Hit h) ⟹ wt S D G (HElim h C pms lms a) C
  by (rule wt.wtElim[OF - plen llen ptyped ltyped])
```

```
  have argsD: ∀ i < length (lcons (S h) ! k).
      wt S D G (as ! i) (argTy h (lcons (S h) ! k ! i))
    using len args by simp
  have wt S D G
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as)) C
    by (rule wt-apps-elimArgs[OF recursiveD len argsD methodD])
  with A show ?case by simp
next
  case (appL t t' u)
  from appL.prems obtain B where
    t: wt S D G t (Fun B A) and u: wt S D G u B
    by (auto elim: wt.cases)
  have wt S D G t' (Fun B A) by (rule appL(2)[OF t])
  from wt.wtApp[OF this u] show ?case .
next
  case (appR u u' t)
  from appR.prems obtain B where
    t: wt S D G t (Fun B A) and u: wt S D G u B
    by (auto elim: wt.cases)
  have wt S D G u' B by (rule appR(2)[OF u])
  from wt.wtApp[OF t this] show ?case .
next
  case (lam t t' B)
  from lam.prems obtain C where
    body: wt S D (B # G) t C and A: A = Fun B C
    by (auto elim: wt.cases)
  have wt S D (B # G) t' C by (rule lam(2)[OF body])
  from wt.wtLam[OF this] A show ?case by simp
next
  case (pconArg a a' h k xs ys)
  let ?as = xs @ a # ys
  let ?i = length xs
  from pconArg.prems obtain
    k: k < length (pcons (S h))
    and len: length ?as = length (pcons (S h) ! k)
    and args: ∀ j < length ?as.
      wt S D G (?as ! j) (argTy h (pcons (S h) ! k ! j))
    and A: A = Hit h
    by (auto elim: wt.cases)
  have i: ?i < length ?as by simp
  have a: wt S D G a (argTy h (pcons (S h) ! k ! ?i))
    using args[rule-format, OF i] by simp
  have a': wt S D G a' (argTy h (pcons (S h) ! k ! ?i))
    by (rule pconArg(2)[OF a])
  have updated: ∀ j < length (?as[?i := a']).
      wt S D G ((?as[?i := a']) ! j)
        (argTy h (pcons (S h) ! k ! j))
    by (rule all-nth-list-update[OF args i a'])
```

```
  have target: xs @ a' # ys = ?as[?i := a'] by simp
  have wt S D G (PCon h k (?as[?i := a'])) (Hit h)
  proof (rule wt.wtPCon)
    show k < length (pcons (S h)) by (rule k)
  next
    show length (?as[?i := a']) = length (pcons (S h) ! k)
      using len by simp
  next
    show ∀ j < length (?as[?i := a']).
        wt S D G ((?as[?i := a']) ! j)
          (argTy h (pcons (S h) ! k ! j))
      by (rule updated)
  qed
  with target A show ?case by simp
next
  case (lconArg a a' h k xs ys d)
  let ?as = xs @ a # ys
  let ?i = length xs
  from lconArg.prems obtain j where d: d = DVar j
    and jD: j ∈ D
    and k: k < length (lcons (S h))
    and len: length ?as = length (lcons (S h) ! k)
    and args: ∀ q < length ?as.
      wt S D G (?as ! q) (argTy h (lcons (S h) ! k ! q))
    and A: A = Hit h
    by (auto elim: wt.cases)
  have i: ?i < length ?as by simp
  have a: wt S D G a (argTy h (lcons (S h) ! k ! ?i))
    using args[rule-format, OF i] by simp
  have a': wt S D G a' (argTy h (lcons (S h) ! k ! ?i))
    by (rule lconArg(2)[OF a])
  have updated: ∀ q < length (?as[?i := a']).
      wt S D G ((?as[?i := a']) ! q)
        (argTy h (lcons (S h) ! k ! q))
    by (rule all-nth-list-update[OF args i a'])
  have target: xs @ a' # ys = ?as[?i := a'] by simp
  have wt S D G (LCon h k (?as[?i := a']) (DVar j)) (Hit h)
  proof (rule wt.wtLCon)
    show j ∈ D by (rule jD)
  next
    show k < length (lcons (S h)) by (rule k)
  next
    show length (?as[?i := a']) = length (lcons (S h) ! k)
      using len by simp
  next
    show ∀ q < length (?as[?i := a']).
        wt S D G ((?as[?i := a']) ! q)
          (argTy h (lcons (S h) ! k ! q))
      by (rule updated)
```

```
  qed
  with target d A show ?case by simp
next
  case (elimPm m m' h C xs ys lms t)
  let ?pms = xs @ m # ys
  let ?i = length xs
  from elimPm.prems obtain
    scrut: wt S D G t (Hit h)
    and plen: length ?pms = length (pcons (S h))
    and llen: length lms = length (lcons (S h))
    and ptyped: ∀ q < length ?pms.
      wt S D G (?pms ! q) (methodTy h C (pcons (S h) ! q))
    and ltyped: ∀ q < length lms.
      wt S (insert 0 D) G (lms ! q) (methodTy h C (lcons (S h) ! q))
    and A: A = C
    by (auto elim: wt.cases)
  have i: ?i < length ?pms by simp
  have m: wt S D G m (methodTy h C (pcons (S h) ! ?i))
    using ptyped[rule-format, OF i] by simp
  have m': wt S D G m' (methodTy h C (pcons (S h) ! ?i))
    by (rule elimPm(2)[OF m])
  have updated: ∀ q < length (?pms[?i := m']).
      wt S D G ((?pms[?i := m']) ! q)
        (methodTy h C (pcons (S h) ! q))
    by (rule all-nth-list-update[OF ptyped i m'])
  have target: xs @ m' # ys = ?pms[?i := m'] by simp
  have wt S D G (HElim h C (?pms[?i := m']) lms t) C
    by (rule wt.wtElim[OF scrut]) (use plen llen updated ltyped in simp-all)
  with target A show ?case by simp
next
  case (elimLm m m' h C pms xs ys t)
  let ?lms = xs @ m # ys
  let ?i = length xs
  from elimLm.prems obtain
    scrut: wt S D G t (Hit h)
    and plen: length pms = length (pcons (S h))
    and llen: length ?lms = length (lcons (S h))
    and ptyped: ∀ q < length pms.
      wt S D G (pms ! q) (methodTy h C (pcons (S h) ! q))
    and ltyped: ∀ q < length ?lms.
      wt S (insert 0 D) G (?lms ! q) (methodTy h C (lcons (S h) ! q))
    and A: A = C
    by (auto elim: wt.cases)
  have i: ?i < length ?lms by simp
  have m: wt S (insert 0 D) G m (methodTy h C (lcons (S h) ! ?i))
    using ltyped[rule-format, OF i] by simp
  have m': wt S (insert 0 D) G m' (methodTy h C (lcons (S h) ! ?i))
    by (rule elimLm(2)[OF m])
  have updated: ∀ q < length (?lms[?i := m']).
```

```
      wt S (insert 0 D) G ((?lms[?i := m']) ! q)
        (methodTy h C (lcons (S h) ! q))
    by (rule all-nth-list-update[OF ltyped i m'])
  have target: xs @ m' # ys = ?lms[?i := m'] by simp
  have wt S D G (HElim h C pms (?lms[?i := m']) t) C
    by (rule wt.wtElim[OF scrut]) (use plen llen ptyped updated in simp-all)
  with target A show ?case by simp
next
  case (elimScrutinee t t' h C pms lms)
  from elimScrutinee.prems obtain
    scrut: wt S D G t (Hit h)
    and plen: length pms = length (pcons (S h))
    and llen: length lms = length (lcons (S h))
    and ptyped: ∀ q < length pms.
      wt S D G (pms ! q) (methodTy h C (pcons (S h) ! q))
    and ltyped: ∀ q < length lms.
      wt S (insert 0 D) G (lms ! q) (methodTy h C (lcons (S h) ! q))
    and A: A = C
    by (auto elim: wt.cases)
  have wt S D G t' (Hit h) by (rule elimScrutinee(2)[OF scrut])
  then have wt S D G (HElim h C pms lms t') C
    by (rule wt.wtElim[OF - plen llen ptyped ltyped])
  with A show ?case by simp
qed

corollary subject-reduction-steps:
  assumes wf-sigenv S and wt S D G t A and reds S t u
  shows wt S D G u A
  using assms(3,2)
proof (induct rule: rtranclp-induct)
  case base
  then show ?case .
next
  case step
  then show ?case
    by (meson subject-reduction[OF assms(1)])
qed

corollary checked-source-subject-reduction:
  assumes wf-sigenv S and cwt S D G t A and red S t u
  shows cwt S D G u A
proof (rule cwtI)
  show 0 ∉ D by (rule cwt-dimension-context[OF assms(2)])
  show wt S D G u A
    by (rule subject-reduction[OF assms(1,3) cwt-wt[OF assms(2)]])
  show hereditarily-checked S u
    by (rule hereditarily-checked-step[
        OF cwt-hereditarily-checked[OF assms(2)] assms(3)])
qed
```

corollary *checked-source-subject-reduction-steps*:
  assumes *wf-sigenv S* **and** *cwt S D G t A* **and** *reds S t u*
  shows *cwt S D G u A*
proof (*rule cwtI*)
  show *0* $\notin$ *D* **by** (*rule cwt-dimension-context*[*OF assms*(*2*)])
  show *wt S D G u A*
    by (*rule subject-reduction-steps*[*OF assms*(*1*) *cwt-wt*[*OF assms*(*2*)] *assms*(*3*)])
  show *hereditarily-checked S u*
    by (*rule hereditarily-checked-steps*[
        OF cwt-hereditarily-checked[*OF assms*(*2*)] *assms*(*3*)])
qed

end
theory *HIT-Canonicity*
  imports *HIT-Preservation*
begin

14 Progress and canonical forms

Progress says that a well-typed closed term is canonical at its type or takes a reduction step. Together with multi-step subject reduction this gives type safety for every reduct of a closed checked term. Canonical forms for normal terms follow immediately.

- Path constructors do not contribute closed values. A path constructor term needs a dimension variable, and there are none in the empty dimension context, so the only closed values of a higher inductive type are the point-constructor ones. This is the sense in which the cubical-style computation rule for path constructors does not destroy the canonical-forms classification.
- The proof is a *progress* argument: at every stage, if the term were not already a value, a computation rule would apply, contradicting normality. In particular the eliminator case uses the computation rule for point constructors, and the constructor case uses the fact that a normal constructor term has normal arguments.

A signature with no point constructors therefore has no closed normal inhabitant. This theory records the general bridge from normalization to empty-signature consistency; *HIT-Normalization* subsequently proves the premise and derives the unconditional corollary.

14.1 Canonical values

primrec *argHit* :: *nat* $\Rightarrow$ *argspec* $\Rightarrow$ *nat* **where**

```
  argHit h (Par h') = h'
| argHit h Rec = h

inductive canonical :: sigenv ⇒ nat ⇒ tm ⇒ bool
  for S :: sigenv
where
  canonicalI:
    k < length (pcons (S h)) ⟹
      length as = length (pcons (S h) ! k) ⟹
      (∀ i < length as.
        canonical S (argHit h (pcons (S h) ! k ! i)) (as ! i)) ⟹
      canonical S h (PCon h k as)

definition canonAt :: sigenv ⇒ ty ⇒ tm ⇒ bool where
  canonAt S A t =
    (case A of
       Hit h ⇒ canonical S h t
     | Fun A1 A2 ⇒ (∃ B u. t = Lam B u))

lemma canonAt-Hit [simp]: canonAt S (Hit h) t = canonical S h t
  by (simp add: canonAt-def)

lemma canonAt-Fun [simp]:
  canonAt S (Fun A1 A2) t = (∃ B u. t = Lam B u)
  by (simp add: canonAt-def)
```

14.2 Normal forms are closed under the relevant subterms

```
lemma nform-App1: nform S (App t u) ⟹ nform S t
  unfolding nform-def by (blast intro: red.appL)

lemma nform-elim-scrutinee:
  nform S (HElim h C pms lms t) ⟹ nform S t
  unfolding nform-def by (blast intro: red.elimScrutinee)

lemma nform-PCon-arg:
  assumes nform S (PCon h k as) and i < length as
  shows nform S (as ! i)
  using assms unfolding nform-def by (blast intro: red-PCon-arg)
```

14.3 Progress

```
theorem progress-aux:
  assumes wt S D G t A
  shows D = {} ⟶ G = [] ⟶
    canonAt S A t ∨ (∃ u. red S t u)
  using assms
proof (induct rule: wt.induct)
  case (wtVar i G D)
  show ?case
```

```
  proof (intro impI)
    assume G = []
    with wtVar show canonAt S (G ! i) (Var i) ∨ (∃ u. red S (Var i) u)
      by simp
  qed
next
  case (wtLam D A G t B)
  show ?case by auto
next
  case (wtApp D G t A B u)
  show ?case
  proof (intro impI)
    assume d: D = {} and g: G = []
    from wtApp(2) d g have
      canonAt S (Fun A B) t ∨ (∃ t'. red S t t') by blast
    then show canonAt S B (App t u) ∨ (∃ v. red S (App t u) v)
    proof
      assume canonAt S (Fun A B) t
      then obtain A' body where t = Lam A' body by auto
      have closed: dclosed0 S u
        unfolding dclosed0-def
        using wt-dsubst-fresh[OF wtApp.hyps(3)] d by simp
      have red S (App (Lam A' body) u) (subst 0 u body)
        by (rule red.beta[OF closed])
      with ‹t = Lam A' body› 
      have red S (App t u) (subst 0 u body) by simp
      then show ?thesis by blast
    next
      assume ∃ t'. red S t t'
      then show ?thesis by (blast intro: red.appL)
    qed
  qed
next
  case (wtPCon k h as D G)
  show ?case
  proof (intro impI)
    assume d: D = {} and g: G = []
    have alternatives: ∀ i < length as.
        canonical S (argHit h (pcons (S h) ! k ! i)) (as ! i) ∨
        (∃ u. red S (as ! i) u)
    proof (intro allI impI)
      fix i
      assume i: i < length as
      from wtPCon(3) i d g have
        canonAt S (argTy h (pcons (S h) ! k ! i)) (as ! i) ∨
          (∃ u. red S (as ! i) u)
        by blast
      then show canonical S (argHit h (pcons (S h) ! k ! i)) (as ! i) ∨
          (∃ u. red S (as ! i) u)
```

```
      by (cases pcons (S h) ! k ! i) simp-all
    qed
    show canonAt S (Hit h) (PCon h k as) ∨
      (∃ u. red S (PCon h k as) u)
    proof (cases ∀ i < length as.
      canonical S (argHit h (pcons (S h) ! k ! i)) (as ! i))
     case True
     have canonical S h (PCon h k as)
      by (rule canonicalI[OF wtPCon(1,2) True])
     then show ?thesis by simp
    next
     case False
     then obtain i where i: i < length as
      and not-canonical:
       ¬ canonical S (argHit h (pcons (S h) ! k ! i)) (as ! i)
      by blast
     from alternatives[rule-format, OF i] not-canonical obtain u where
      red S (as ! i) u by blast
     from red-PCon-arg[OF i this] show ?thesis by blast
    qed
  qed
next
  case (wtLCon j D k h as G)
  show ?case
  proof (intro impI)
    assume D = {}
    with wtLCon(1) show
      canonAt S (Hit h) (LCon h k as (DVar j)) ∨
       (∃ u. red S (LCon h k as (DVar j)) u)
      by simp
  qed
next
  case (wtElim D G t h pms lms C)
  show ?case
  proof (intro impI)
    assume d: D = {} and g: G = []
    from wtElim(2) d g have
      canonical S h t ∨ (∃ t'. red S t t') by simp
    then show canonAt S C (HElim h C pms lms t) ∨
      (∃ u. red S (HElim h C pms lms t) u)
    proof
     assume canonical S h t
     then obtain k as where t: t = PCon h k as
      and k: k < length (pcons (S h))
      by (auto elim: canonical.cases)
     have red S (HElim h C pms lms (PCon h k as))
       (apps (pms ! k) (elimArgs h C pms lms (pcons (S h) ! k) as))
      by (rule red.elimPoint[where S = S and h = h and k = k, OF k])
     with t show ?thesis by blast
```

```
    next
      assume ∃ t'. red S t t'
      then show ?thesis by (blast intro: red.elimScrutinee)
    qed
  qed
qed

theorem progress:
  assumes cwt S {} [] t A
  shows canonAt S A t ∨ (∃ u. red S t u)
  using progress-aux[OF cwt-wt[OF assms]] by simp

theorem type-safety:
  assumes wf: wf-sigenv S
    and typed: cwt S {} [] t A
    and reduces: reds S t u
  shows canonAt S A u ∨ (∃ v. red S u v)
proof −
  have cwt S {} [] u A
    by (rule checked-source-subject-reduction-steps[OF wf typed reduces])
  from progress[OF this] show ?thesis .
qed

theorem canonical-forms-raw:
  assumes wt S {} [] t A and nform S t
  shows canonAt S A t
  using progress-aux[OF assms(1)] assms(2)
  unfolding nform-def by blast

theorem canonical-forms:
  assumes cwt S {} [] t A and nform S t
  shows canonAt S A t
  using canonical-forms-raw[OF cwt-wt[OF assms(1)] assms(2)] .

corollary canonical-forms-hit:
  assumes cwt S {} [] t (Hit h) and nform S t
  shows canonical S h t
  using canonical-forms[OF assms] by simp

definition normalizes :: sigenv ⇒ tm ⇒ bool where
  normalizes S t ⟷ (∃ u. reds S t u ∧ nform S u)

theorem normalizing-term-has-canonical-result:
  assumes wf: wf-sigenv S
    and typed: cwt S {} [] t A
    and normalization: normalizes S t
  shows ∃ u. reds S t u ∧ nform S u ∧ canonAt S A u
proof −
  from normalization obtain u where reduces: reds S t u
```

and *normal*: *nform S u*
unfolding *normalizes-def* **by** *blast*
have *wt S* {} [] *u A*
by (*rule subject-reduction-steps*[*OF wf cwt-wt*[*OF typed*] *reduces*])
from *canonical-forms-raw*[*OF this normal*] *reduces normal* **show** *?thesis* **by** *blast*
qed

14.4 The empty signature

A signature with no point constructors has no closed normal inhabitant. At this stage the conclusion classifies normal terms only. The following bridge is discharged by the strong-normalization theorem in the next theory.

theorem *no-canonical-without-constructors*:
assumes *pcons* (*S h*) = []
shows ¬ *canonical S h t*
proof
assume *canonical S h t*
then show *False* **using** *assms* **by** (*auto elim*: *canonical.cases*)
qed

corollary *no-closed-normal-inhabitant*:
assumes *pcons* (*S h*) = []
and *cwt S* {} [] *t* (*Hit h*)
and *nform S t*
shows *False*
using *canonical-forms-hit*[*OF assms*(*2*,*3*)]
no-canonical-without-constructors[**where** *S* = *S* **and** *h* = *h*, *OF assms*(*1*)]
by *simp*

Subject reduction connects this classification to any explicitly supplied normalization sequence. Thus no checked closed inhabitant of an empty signature can reduce to a normal form.

corollary *empty-type-has-no-normalizing-inhabitant*:
assumes *wf*: *wf-sigenv S*
and *empty*: *pcons* (*S h*) = []
and *typed*: *cwt S* {} [] *t* (*Hit h*)
and *reduces*: *reds S t u*
and *normal*: *nform S u*
shows *False*
proof −
have *wt S* {} [] *u* (*Hit h*)
by (*rule subject-reduction-steps*[*OF wf cwt-wt*[*OF typed*] *reduces*])
from *canonical-forms-raw*[*OF this normal*]
no-canonical-without-constructors[**where** *S* = *S* **and** *h* = *h* **and** *t* = *u*, *OF empty*]
show *False* **by** *simp*
qed

corollary *normalization-implies-empty-consistency*:
  assumes *wf*: *wf-sigenv S*
    and *empty*: *pcons* (*S h*) = []
    and *normalization*: ⋀*t*. *cwt S* {} [] *t* (*Hit h*) ⟹ *normalizes S t*
  shows ¬ (∃ *t*. *cwt S* {} [] *t* (*Hit h*))
proof
  assume ∃ *t*. *cwt S* {} [] *t* (*Hit h*)
  then obtain *t* **where** *typed*: *cwt S* {} [] *t* (*Hit h*) **by** *blast*
 from *normalizing-term-has-canonical-result*[*OF wf typed normalization*[*OF typed*]]
  obtain *u* **where** *canonical S h u* **by** *auto*
  with *no-canonical-without-constructors*[
      where *S* = *S* **and** *h* = *h* **and** *t* = *u*, *OF empty*]
  show *False* **by** *blast*
qed

end
theory *HIT-Normalization*
  imports *HIT-Canonicity*
begin

15 Strong normalization

15.1 Dimension erasure and regular terms

fun *variable-dims* :: *tm* ⇒ *bool* **where**
  variable-dims (*Var i*) = *True*
| *variable-dims* (*Lam A t*) = *variable-dims t*
| *variable-dims* (*App t u*) = (*variable-dims t* ∧ *variable-dims u*)
| *variable-dims* (*PCon h k as*) = *list-all variable-dims as*
| *variable-dims* (*LCon h k as d*) =
    (*list-all variable-dims as* ∧ (∃ *j*. *d* = *DVar j*))
| *variable-dims* (*HElim h C pms lms t*) =
    (*list-all variable-dims pms* ∧ *list-all variable-dims lms* ∧
      variable-dims t)

fun *regular* :: *sigenv* ⇒ *tm* ⇒ *bool* **where**
  regular S (*Var i*) = *True*
| *regular S* (*Lam A t*) = *regular S t*
| *regular S* (*App t u*) = (*regular S t* ∧ *regular S u*)
| *regular S* (*PCon h k as*) =
    (*k* < *length* (*pcons* (*S h*)) ∧
      length as = *length* (*pcons* (*S h*) ! *k*) ∧
      list-all (*regular S*) *as*)
| *regular S* (*LCon h k as d*) =
    (*k* < *length* (*lcons* (*S h*)) ∧
      length as = *length* (*lcons* (*S h*) ! *k*) ∧
      list-all (*regular S*) *as* ∧ (∃ *j*. *d* = *DVar j*))
| *regular S* (*HElim h C pms lms t*) =
    (*length pms* = *length* (*pcons* (*S h*)) ∧

```
    length lms = length (lcons (S h)) ∧
    list-all (regular S) pms ∧ list-all (regular S) lms ∧
    regular S t)

fun erase-dims :: sigenv ⇒ tm ⇒ tm where
  erase-dims S (Var i) = Var i
| erase-dims S (Lam A t) = Lam A (erase-dims S t)
| erase-dims S (App t u) = App (erase-dims S t) (erase-dims S u)
| erase-dims S (PCon h k as) = PCon h k (map (erase-dims S) as)
| erase-dims S (LCon h k as d) =
    LCon h k (map (erase-dims S) as) (DVar 1)
| erase-dims S (HElim h C pms lms t) =
    HElim h C (map (erase-dims S) pms) (map (erase-dims S) lms)
      (erase-dims S t)

lemma regular-variable-dims:
  regular S t ⟹ variable-dims t
  by (induct t) (auto simp: list-all-iff)

lemma variable-dims-erase [simp]: variable-dims (erase-dims S t)
  by (induct t) (auto simp: list-all-iff)

lemma dclosed0-erase [simp]: dclosed0 S (erase-dims S t)
  by (induct t) (auto simp: list-all-iff)

lemma erase-dims-idem [simp]:
  erase-dims S (erase-dims S t) = erase-dims S t
  by (induct t) simp-all

lemma variable-dims-dsubst-erase:
  erase-dims S (dsubst S n (DVar j) (erase-dims S t)) =
    erase-dims S t
proof (induct t arbitrary: n j)
  case (LCon h k as d)
  then show ?case by (cases 1 = n) simp-all
qed simp-all

lemma dsubst-zero-erase [simp]:
  dsubst S 0 (DVar j) (erase-dims S t) = erase-dims S t
  by (induct t arbitrary: j) simp-all

lemma erase-dims-dsubst-variable:
  assumes variable-dims t
  shows erase-dims S (dsubst S n (DVar j) t) = erase-dims S t
  using assms
proof (induct t arbitrary: n j)
  case (LCon h k as d)
  then obtain q where d: d = DVar q by auto
  from LCon d show ?case
```

```
    by (cases q = n) (auto simp: list-all-iff)
qed (auto simp: list-all-iff)

lemma erase-dims-lift [simp]:
  erase-dims S (lift n t) = lift n (erase-dims S t)
  by (induct t arbitrary: n) simp-all

lemma erase-dims-subst [simp]:
  erase-dims S (subst n s t) =
    subst n (erase-dims S s) (erase-dims S t)
  by (induct t arbitrary: n s) simp-all

lemma erase-dims-apps [simp]:
  erase-dims S (apps t xs) =
    apps (erase-dims S t) (map (erase-dims S) xs)
  by (induct xs arbitrary: t) simp-all

lemma erase-dims-elimArgs:
  map (erase-dims S)
      (elimArgs h C pms lms ss as) =
    elimArgs h C (map (erase-dims S) pms) (map (erase-dims S) lms) ss
      (map (erase-dims S) as)
  by (induct ss arbitrary: as) (auto split: list.split argspec.split)

lemma regular-wt:
  assumes wt S D G t A
  shows regular S t
  using assms
  by (induct rule: wt.induct)
    (auto simp: list-all-iff in-set-conv-nth)

lemma erase-dims-wt:
  assumes wt S D G t A
  shows wt S {1} G (erase-dims S t) A
  using assms
proof (induct rule: wt.induct)
  case (wtVar i G D)
  then show ?case by (auto intro: wt.wtVar)
next
  case (wtLam D A G t B)
  then show ?case by (auto intro: wt.wtLam)
next
  case (wtApp D G t A B u)
  then show ?case by (auto intro: wt.wtApp)
next
  case (wtPCon k h as D G)
  then show ?case
    by (auto intro!: wt.wtPCon simp: in-set-conv-nth)
next
```

```
  case (wtLCon j D k h as G)
  then show ?case
    by (auto intro!: wt.wtLCon simp: in-set-conv-nth)
next
  case (wtElim D G t h pms lms C)
  have lms: ∀ k < length (map (erase-dims S) lms).
      wt S (insert 0 {1}) G ((map (erase-dims S) lms) ! k)
        (methodTy h C (lcons (S h) ! k))
  proof (intro allI impI)
    fix k
    assume k: k < length (map (erase-dims S) lms)
    have wt S {1} G (erase-dims S (lms ! k))
        (methodTy h C (lcons (S h) ! k))
      using wtElim(6) k by simp
    then have wt S (insert 0 {1}) G (erase-dims S (lms ! k))
        (methodTy h C (lcons (S h) ! k))
      by (rule wt-dimension-mono) auto
    with k show wt S (insert 0 {1}) G
        ((map (erase-dims S) lms) ! k)
        (methodTy h C (lcons (S h) ! k))
      by simp
  qed
  have wt S {1} G
      (HElim h C (map (erase-dims S) pms) (map (erase-dims S) lms)
        (erase-dims S t)) C
  proof (rule wt.wtElim)
    show wt S {1} G (erase-dims S t) (Hit h)
      by (rule wtElim(2))
    show length (map (erase-dims S) pms) = length (pcons (S h))
      using wtElim(3) by simp
    show length (map (erase-dims S) lms) = length (lcons (S h))
      using wtElim(4) by simp
    show ∀ k < length (map (erase-dims S) pms).
        wt S {1} G ((map (erase-dims S) pms) ! k)
          (methodTy h C (pcons (S h) ! k))
      using wtElim(5) by simp
    show ∀ k < length (map (erase-dims S) lms).
        wt S (insert 0 {1}) G ((map (erase-dims S) lms) ! k)
          (methodTy h C (lcons (S h) ! k))
      by (rule lms)
  qed
  then show ?case by simp
qed

lemma regular-lift:
  regular S t ⟹ regular S (lift n t)
  by (induct t arbitrary: n) (auto simp: list-all-iff)

lemma variable-dims-lift:
```

variable-dims t $\Longrightarrow$ *variable-dims* (*lift n t*)
by (*induct t arbitrary*: *n*) (*auto simp*: *list-all-iff*)

lemma *variable-dims-subst*:
assumes *variable-dims s* **and** *variable-dims t*
shows *variable-dims* (*subst n s t*)
using *assms variable-dims-lift*
by (*induct t arbitrary*: *n s*) (*auto simp*: *list-all-iff*)

lemma *regular-subst*:
assumes *regular S s* **and** *regular S t*
shows *regular S* (*subst n s t*)
using *assms regular-lift*
by (*induct t arbitrary*: *n s*) (*auto simp*: *list-all-iff*)

lemma *regular-dsubst-variable*:
assumes *regular S t*
shows *regular S* (*dsubst S n* (*DVar j*) *t*)
using *assms*
proof (*induct t arbitrary*: *n j*)
case (*LCon h k as d*)
then obtain *q* **where** *d*: *d* = *DVar q* **by** *auto*
from *LCon d* **show** *?case*
by (*cases q* = *n*) (*auto simp*: *list-all-iff*)
qed (*auto simp*: *list-all-iff*)

lemma *regular-apps*:
assumes *regular S t* **and** *list-all* (*regular S*) *xs*
shows *regular S* (*apps t xs*)
using *assms* **by** (*induct xs arbitrary*: *t*) *auto*

lemma *regular-elimArgs*:
assumes *plen*: *length pms* = *length* (*pcons* (*S h*))
and *llen*: *length lms* = *length* (*lcons* (*S h*))
and *pms*: *list-all* (*regular S*) *pms*
and *lms*: *list-all* (*regular S*) *lms*
and *args*: *list-all* (*regular S*) *as*
shows *list-all* (*regular S*) (*elimArgs h C pms lms ss as*)
using *args*
proof (*induct ss arbitrary*: *as*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons s ss*)
then show *?case*
using *plen llen pms lms*
by (*cases as*; *cases s*) (*auto simp*: *list-all-iff*)
qed

lemma *regular-red*:
assumes *step*: *red S t u* **and** *source*: *regular S t*
shows *regular S u*
using *step source*
proof (*induct rule*: *red.induct*)
case (*beta u A t*)
then show *?case* **by** (*auto intro*: *regular-subst*)
next
case (*elimPoint k h C pms lms as*)
have *method*: *regular S* (*pms* ! *k*)
using *elimPoint* **by** (*auto simp*: *list-all-iff*)
have *args*: *list-all* (*regular S*) *as*
using *elimPoint* **by** *simp*
have *eargs*: *list-all* (*regular S*)
(*elimArgs h C pms lms* (*pcons* (*S h*) ! *k*) *as*)
by (*rule regular-elimArgs*)
(*use elimPoint args* **in** *simp-all*)
show *?case* **by** (*rule regular-apps*[*OF method eargs*])
next
case (*elimPath k h C pms lms as j*)
have *method0*: *regular S* (*lms* ! *k*)
using *elimPath* **by** (*auto simp*: *list-all-iff*)
have *method*: *regular S* (*dsubst S 0* (*DVar j*) (*lms* ! *k*))
by (*rule regular-dsubst-variable*[*OF method0*])
have *args*: *list-all* (*regular S*) *as*
using *elimPath* **by** *simp*
have *eargs*: *list-all* (*regular S*)
(*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*) *as*)
by (*rule regular-elimArgs*)
(*use elimPath args* **in** *simp-all*)
show *?case* **by** (*rule regular-apps*[*OF method eargs*])
next
case (*appL t t′ u*)
then show *?case* **by** *simp*
next
case (*appR u u′ t*)
then show *?case* **by** *simp*
next
case (*lam t t′ A*)
then show *?case* **by** *simp*
next
case (*pconArg a a′ h k xs ys*)
then show *?case* **by** (*auto simp*: *list-all-iff*)
next
case (*lconArg a a′ h k xs ys d*)
then show *?case* **by** (*auto simp*: *list-all-iff*)
next
case (*elimPm m m′ h C xs ys lms t*)
then show *?case* **by** (*auto simp*: *list-all-iff*)

```
next
  case (elimLm m m' h C pms xs ys t)
  then show ?case by (auto simp: list-all-iff)
next
  case (elimScrutinee t t' h C pms lms)
  then show ?case by simp
qed

lemma red-erase-dims:
  assumes step: red S t u and source: regular S t
  shows red S (erase-dims S t) (erase-dims S u)
  using step source
proof (induct rule: red.induct)
  case (beta u A t)
  then show ?case by (simp add: red.beta)
next
  case (elimPoint k h C pms lms as)
  have kpm: k < length pms using elimPoint by simp
  have step: red S
      (HElim h C (map (erase-dims S) pms) (map (erase-dims S) lms)
        (PCon h k (map (erase-dims S) as)))
      (apps ((map (erase-dims S) pms) ! k)
        (elimArgs h C (map (erase-dims S) pms)
          (map (erase-dims S) lms) (pcons (S h) ! k)
          (map (erase-dims S) as)))
    by (rule red.elimPoint[
          where S = S and h = h and k = k and C = C
            and pms = map (erase-dims S) pms
            and lms = map (erase-dims S) lms
            and as = map (erase-dims S) as,
          OF elimPoint(1)])
  from step kpm show ?case
    by (simp add: erase-dims-elimArgs)
next
  case (elimPath k h C pms lms as j)
  have klm: k < length lms using elimPath by simp
  have method-regular: regular S (lms ! k)
    using elimPath klm by (auto simp: list-all-iff)
  have method-erase:
      erase-dims S (dsubst S 0 (DVar j) (lms ! k)) =
        erase-dims S (lms ! k)
    by (rule erase-dims-dsubst-variable[
          OF regular-variable-dims[OF method-regular]])
  have step: red S
      (HElim h C (map (erase-dims S) pms) (map (erase-dims S) lms)
        (LCon h k (map (erase-dims S) as) (DVar 1)))
      (apps (dsubst S 0 (DVar 1) ((map (erase-dims S) lms) ! k))
        (elimArgs h C (map (erase-dims S) pms)
          (map (erase-dims S) lms) (lcons (S h) ! k)
```

```
      (map (erase-dims S) as)))
  by (rule red.elimPath[
      where S = S and h = h and k = k and C = C
        and pms = map (erase-dims S) pms
        and lms = map (erase-dims S) lms
        and as = map (erase-dims S) as and j = 1,
      OF elimPath(1)])
 from step klm show ?case
  by (simp add: erase-dims-elimArgs method-erase)
next
 case (appL t t' u)
 then show ?case by (auto intro: red.appL)
next
 case (appR u u' t)
 then show ?case by (auto intro: red.appR)
next
 case (lam t t' A)
 then show ?case by (auto intro: red.lam)
next
 case (pconArg a a' h k xs ys)
 have red S
    (PCon h k (map (erase-dims S) xs @ erase-dims S a #
      map (erase-dims S) ys))
    (PCon h k (map (erase-dims S) xs @ erase-dims S a' #
      map (erase-dims S) ys))
  by (rule red.pconArg) (use pconArg in auto)
 then show ?case by simp
next
 case (lconArg a a' h k xs ys d)
 have red S
    (LCon h k (map (erase-dims S) xs @ erase-dims S a #
      map (erase-dims S) ys) (DVar 1))
    (LCon h k (map (erase-dims S) xs @ erase-dims S a' #
      map (erase-dims S) ys) (DVar 1))
  by (rule red.lconArg) (use lconArg in auto)
 then show ?case by simp
next
 case (elimPm m m' h C xs ys lms t)
 have red S
    (HElim h C
      (map (erase-dims S) xs @ erase-dims S m #
        map (erase-dims S) ys)
      (map (erase-dims S) lms) (erase-dims S t))
    (HElim h C
      (map (erase-dims S) xs @ erase-dims S m' #
        map (erase-dims S) ys)
      (map (erase-dims S) lms) (erase-dims S t))
  by (rule red.elimPm) (use elimPm in auto)
 then show ?case by simp
```

```
next
  case (elimLm m m' h C pms xs ys t)
  have red S
      (HElim h C (map (erase-dims S) pms)
        (map (erase-dims S) xs @ erase-dims S m #
          map (erase-dims S) ys) (erase-dims S t))
      (HElim h C (map (erase-dims S) pms)
        (map (erase-dims S) xs @ erase-dims S m' #
          map (erase-dims S) ys) (erase-dims S t))
    by (rule red.elimLm) (use elimLm in auto)
  then show ?case by simp
next
  case (elimScrutinee t t' h C pms lms)
  then show ?case by (auto intro: red.elimScrutinee)
qed

lemma SN-erase-dims-reflect-aux:
  assumes sn: SN S e
  shows ⋀t D G A. e = erase-dims S t ⟹
    wf-sigenv S ⟹ wt S D G t A ⟹ SN S t
  using sn
proof (induct rule: SN.induct)
  case (SNI e)
  show ?case
  proof (rule SN.SNI)
    fix u
    assume step: red S t u
    have reg: regular S t by (rule regular-wt[OF SNI.prems(3)])
    have erased-step: red S e (erase-dims S u)
      using red-erase-dims[OF step reg] SNI.prems(1) by simp
    have typed: wt S D G u A
      by (rule subject-reduction[OF SNI.prems(2) step SNI.prems(3)])
    show SN S u
      by (rule SNI.hyps(2)[OF erased-step refl SNI.prems(2) typed])
  qed
qed

lemma SN-erase-dims-reflect:
  assumes wf-sigenv S and wt S D G t A
    and SN S (erase-dims S t)
  shows SN S t
  by (rule SN-erase-dims-reflect-aux[OF assms(3) refl assms(1,2)])
```

15.2 Renaming and simultaneous substitution

```
fun up-ren :: (nat ⇒ nat) ⇒ nat ⇒ nat where
  up-ren f 0 = 0
| up-ren f (Suc i) = Suc (f i)
```

```
fun vrename :: (nat ⇒ nat) ⇒ tm ⇒ tm where
  vrename f (Var i) = Var (f i)
| vrename f (Lam A t) = Lam A (vrename (up-ren f) t)
| vrename f (App t u) = App (vrename f t) (vrename f u)
| vrename f (PCon h k as) = PCon h k (map (vrename f) as)
| vrename f (LCon h k as d) = LCon h k (map (vrename f) as) d
| vrename f (HElim h C pms lms t) =
    HElim h C (map (vrename f) pms) (map (vrename f) lms)
      (vrename f t)

fun up-sub :: (nat ⇒ tm) ⇒ nat ⇒ tm where
  up-sub s 0 = Var 0
| up-sub s (Suc i) = lift 0 (s i)

fun psubst :: (nat ⇒ tm) ⇒ tm ⇒ tm where
  psubst s (Var i) = s i
| psubst s (Lam A t) = Lam A (psubst (up-sub s) t)
| psubst s (App t u) = App (psubst s t) (psubst s u)
| psubst s (PCon h k as) = PCon h k (map (psubst s) as)
| psubst s (LCon h k as d) = LCon h k (map (psubst s) as) d
| psubst s (HElim h C pms lms t) =
    HElim h C (map (psubst s) pms) (map (psubst s) lms)
      (psubst s t)

definition ext-sub :: tm ⇒ (nat ⇒ tm) ⇒ nat ⇒ tm where
  ext-sub u s i = (case i of 0 ⇒ u | Suc j ⇒ s j)

definition Ren :: nat ⇒ nat ⇒ (nat ⇒ nat) ⇒ bool where
  Ren n m f ⟷ (∀ i < n. f i < m)

lemma up-ren-id [simp]: up-ren id = id
  by (rule ext) (case-tac x; simp)

lemma up-ren-id-apply [simp]: up-ren (λx. x) i = i
  by (cases i) simp-all

lemma up-ren-comp:
  up-ren (f ∘ g) = up-ren f ∘ up-ren g
  by (rule ext) (case-tac x; simp)

lemma vrename-id [simp]: vrename id t = t
proof (induct t)
  case (Lam A t)
  then show ?case by (simp only: vrename.simps up-ren-id)
qed (simp-all add: map-idI)

lemma vrename-comp:
  vrename f (vrename g t) = vrename (f ∘ g) t
  by (induct t arbitrary: f g)
```

(*simp-all add*: *up-ren-comp map-map*)

lemma *erase-dims-vrename* [*simp*]:
erase-dims S (*vrename f t*) = *vrename f* (*erase-dims S t*)
by (*induct t arbitrary*: *f*) (*simp-all add*: *map-map*)

lemma *regular-vrename* [*simp*]:
regular S (*vrename f t*) = *regular S t*
by (*induct t arbitrary*: *f*) (*auto simp*: *list-all-iff*)

lemma *variable-dims-vrename* [*simp*]:
variable-dims (*vrename f t*) = *variable-dims t*
by (*induct t arbitrary*: *f*) (*auto simp*: *list-all-iff*)

lemma *up-sub-Var* [*simp*]: *up-sub Var* = *Var*
by (*rule ext*) (*case-tac x*; *simp*)

lemma *psubst-Var* [*simp*]: *psubst Var t* = *t*
by (*induct t*) (*simp-all add*: *map-idI*)

lemma *erase-dims-up-sub*:
(λ*i*. *erase-dims S* (*up-sub s i*)) =
up-sub (λ*i*. *erase-dims S* (*s i*))
proof
fix *i*
show *erase-dims S* (*up-sub s i*) =
up-sub (λ*i*. *erase-dims S* (*s i*)) *i*
by (*cases i*) *simp-all*
qed

lemma *erase-dims-psubst*:
erase-dims S (*psubst s t*) =
psubst (λ*i*. *erase-dims S* (*s i*)) (*erase-dims S t*)
by (*induct t arbitrary*: *s*)
(*simp-all add*: *map-map erase-dims-up-sub*)

lemma *regular-up-sub*:
assumes ⋀*i*. *regular S* (*s i*)
shows *regular S* (*up-sub s i*)
using *assms* **by** (*cases i*) (*auto intro*: *regular-lift*)

lemma *variable-dims-up-sub*:
assumes ⋀*i*. *variable-dims* (*s i*)
shows *variable-dims* (*up-sub s i*)
using *assms* **by** (*cases i*) (*auto intro*: *variable-dims-lift*)

lemma *regular-psubst*:
assumes *reg*: *regular S t*
and *env*: ⋀*i*. *regular S* (*s i*)

shows *regular S* (*psubst s t*)
using *reg env*
by (*induct t arbitrary*: *s*)
(*auto simp*: *list-all-iff intro*: *regular-up-sub*)

lemma *variable-dims-psubst*:
assumes *dims*: *variable-dims t*
and *env*: $\bigwedge i$. *variable-dims* (*s i*)
shows *variable-dims* (*psubst s t*)
using *dims env*
by (*induct t arbitrary*: *s*)
(*auto simp*: *list-all-iff intro*: *variable-dims-up-sub*)

lemma *Ren-id* [*simp*]: *Ren n n id*
unfolding *Ren-def* **by** *simp*

lemma *Ren-comp*:
assumes *Ren n m f* **and** *Ren m k g*
shows *Ren n k* (*g* $\circ$ *f*)
using *assms* **unfolding** *Ren-def* **by** *auto*

lemma *Ren-up*:
assumes *Ren n m f*
shows *Ren* (*Suc n*) (*Suc m*) (*up-ren f*)
using *assms* **unfolding** *Ren-def*
by (*intro allI impI*) (*case-tac i*; *auto*)

lemma *Ren-Suc*: *Ren n* (*Suc n*) *Suc*
unfolding *Ren-def* **by** *auto*

definition *shift-ren* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
shift-ren n i = (*if i* $<$ *n then i else Suc i*)

lemma *up-ren-shift*:
up-ren (*shift-ren n*) = *shift-ren* (*Suc n*)
unfolding *shift-ren-def*
proof
fix *i*
show *up-ren* (λi. *if i* $<$ *n then i else Suc i*) *i* =
(*if i* $<$ *Suc n then i else Suc i*)
by (*cases i*) *simp-all*
qed

lemma *lift-as-vrename*:
lift n t = *vrename* (*shift-ren n*) *t*
by (*induct t arbitrary*: *n*)
(*simp-all add*: *shift-ren-def up-ren-shift map-map*)

lemma *vrename-lift0*:

```
  vrename (up-ren f) (lift 0 t) = lift 0 (vrename f t)
proof −
  have eq: up-ren f ∘ shift-ren 0 = shift-ren 0 ∘ f
    unfolding shift-ren-def
    by (rule ext) simp
  show ?thesis
    by (simp add: lift-as-vrename vrename-comp eq)
qed

lemma up-sub-vrename:
  (λi. vrename (up-ren f) (up-sub s i)) =
    up-sub (λi. vrename f (s i))
proof
  fix i
  show vrename (up-ren f) (up-sub s i) =
      up-sub (λi. vrename f (s i)) i
    by (cases i) (simp-all add: vrename-lift0)
qed

lemma vrename-psubst:
  vrename f (psubst s t) =
    psubst (λi. vrename f (s i)) t
  by (induct t arbitrary: f s)
    (simp-all add: up-sub-vrename map-map)

lemma up-sub-comp:
  up-sub s ∘ up-ren f = up-sub (s ∘ f)
proof
  fix i
  show (up-sub s ∘ up-ren f) i = up-sub (s ∘ f) i
    by (cases i) simp-all
qed

lemma psubst-vrename:
  psubst s (vrename f t) = psubst (s ∘ f) t
  by (induct t arbitrary: f s)
    (simp-all add: up-sub-comp map-map)

definition single-sub :: nat ⇒ tm ⇒ nat ⇒ tm where
  single-sub n s i =
    (if i < n then Var i else if i = n then s else Var (i − 1))

lemma up-sub-single:
  up-sub (single-sub n s) = single-sub (Suc n) (lift 0 s)
  unfolding single-sub-def
proof
  fix i
  show up-sub (λi.
      if i < n then Var i else if i = n then s else Var (i − 1)) i =
```

```
    (if i < Suc n then Var i
      else if i = Suc n then lift 0 s else Var (i − 1))
    by (cases i) simp-all
qed

lemma subst-as-psubst:
  subst n s t = psubst (single-sub n s) t
  by (induct t arbitrary: n s)
    (simp-all add: single-sub-def up-sub-single map-map)

lemma vrename-subst0:
  vrename f (subst 0 s t) =
    subst 0 (vrename f s) (vrename (up-ren f) t)
proof −
  have eq: (λi. vrename f (single-sub 0 s i)) =
      single-sub 0 (vrename f s) ∘ up-ren f
  proof
    fix i
    show vrename f (single-sub 0 s i) =
        (single-sub 0 (vrename f s) ∘ up-ren f) i
      by (cases i) (simp-all add: single-sub-def)
  qed
  show ?thesis
    by (simp add: subst-as-psubst vrename-psubst psubst-vrename eq)
qed

lemma vrename-apps [simp]:
  vrename f (apps t xs) = apps (vrename f t) (map (vrename f) xs)
  by (induct xs arbitrary: t) simp-all

lemma vrename-elimArgs:
  map (vrename f) (elimArgs h C pms lms ss as) =
    elimArgs h C (map (vrename f) pms) (map (vrename f) lms) ss
      (map (vrename f) as)
  by (induct ss arbitrary: as) (auto split: list.split argspec.split)

lemma vrename-dsubst-variable:
  assumes variable-dims t
  shows vrename f (dsubst S n (DVar j) t) =
    dsubst S n (DVar j) (vrename f t)
  using assms
proof (induct t arbitrary: f n j)
  case (LCon h k as d)
  then obtain q where d: d = DVar q by auto
  from LCon d show ?case
    by (cases q = n) (auto simp: list-all-iff map-map)
qed (auto simp: list-all-iff map-map)

lemma dclosed0-vrename:
```

```
  assumes variable-dims t
  shows dclosed0 S (vrename f t) ⟷ dclosed0 S t
  using assms
  by (induct t arbitrary: f)
    (auto simp: list-all-iff)

lemma red-vrename:
  assumes step: red S t u and source: regular S t
  shows red S (vrename f t) (vrename f u)
  using step source
proof (induct arbitrary: f rule: red.induct)
  case (beta u A t)
  have dims: variable-dims u
    by (rule regular-variable-dims[where S = S]) (use beta in simp)
  have closed: dclosed0 S (vrename f u)
    using beta.hyps(1) dclosed0-vrename[OF dims, of S f] by simp
  show ?case
    by (simp add: vrename-subst0 red.beta[OF closed])
next
  case (elimPoint k h C pms lms as)
  have kpm: k < length pms using elimPoint by simp
  have step: red S
      (HElim h C (map (vrename f) pms) (map (vrename f) lms)
        (PCon h k (map (vrename f) as)))
      (apps ((map (vrename f) pms) ! k)
        (elimArgs h C (map (vrename f) pms) (map (vrename f) lms)
          (pcons (S h) ! k) (map (vrename f) as)))
    by (rule red.elimPoint[
          where S = S and h = h and k = k and C = C
            and pms = map (vrename f) pms
            and lms = map (vrename f) lms
            and as = map (vrename f) as,
          OF elimPoint(1)])
  from step kpm show ?case
    by (simp add: vrename-elimArgs)
next
  case (elimPath k h C pms lms as j)
  have klm: k < length lms using elimPath by simp
  have method-regular: regular S (lms ! k)
    using elimPath klm by (auto simp: list-all-iff)
  have method-dims: variable-dims (lms ! k)
    by (rule regular-variable-dims[OF method-regular])
  have method:
      vrename f (dsubst S 0 (DVar j) (lms ! k)) =
        dsubst S 0 (DVar j) (vrename f (lms ! k))
    by (rule vrename-dsubst-variable[OF method-dims])
  have step: red S
      (HElim h C (map (vrename f) pms) (map (vrename f) lms)
        (LCon h k (map (vrename f) as) (DVar j)))
```

```
    (apps (dsubst S 0 (DVar j) ((map (vrename f) lms) ! k))
      (elimArgs h C (map (vrename f) pms) (map (vrename f) lms)
        (lcons (S h) ! k) (map (vrename f) as)))
  by (rule red.elimPath[
       where S = S and h = h and k = k and C = C
         and pms = map (vrename f) pms
         and lms = map (vrename f) lms
         and as = map (vrename f) as and j = j,
       OF elimPath(1)])
 from step klm method show ?case
  by (simp add: vrename-elimArgs)
next
 case (appL t t' u)
 then show ?case by (auto intro: red.appL)
next
 case (appR u u' t)
 then show ?case by (auto intro: red.appR)
next
 case (lam t t' A)
 then show ?case by (auto intro: red.lam)
next
 case (pconArg a a' h k xs ys)
 have red S
    (PCon h k (map (vrename f) xs @ vrename f a #
      map (vrename f) ys))
    (PCon h k (map (vrename f) xs @ vrename f a' #
      map (vrename f) ys))
  by (rule red.pconArg) (use pconArg in auto)
 then show ?case by simp
next
 case (lconArg a a' h k xs ys d)
 have red S
    (LCon h k (map (vrename f) xs @ vrename f a #
      map (vrename f) ys) d)
    (LCon h k (map (vrename f) xs @ vrename f a' #
      map (vrename f) ys) d)
  by (rule red.lconArg) (use lconArg in auto)
 then show ?case by simp
next
 case (elimPm m m' h C xs ys lms t)
 have red S
    (HElim h C
      (map (vrename f) xs @ vrename f m # map (vrename f) ys)
      (map (vrename f) lms) (vrename f t))
    (HElim h C
      (map (vrename f) xs @ vrename f m' # map (vrename f) ys)
      (map (vrename f) lms) (vrename f t))
  by (rule red.elimPm) (use elimPm in auto)
 then show ?case by simp
```

```
next
  case (elimLm m m' h C pms xs ys t)
  have red S
      (HElim h C (map (vrename f) pms)
        (map (vrename f) xs @ vrename f m # map (vrename f) ys)
        (vrename f t))
      (HElim h C (map (vrename f) pms)
        (map (vrename f) xs @ vrename f m' # map (vrename f) ys)
        (vrename f t))
    by (rule red.elimLm) (use elimLm in auto)
  then show ?case by simp
next
  case (elimScrutinee t t' h C pms lms)
  then show ?case by (auto intro: red.elimScrutinee)
qed

lemma map-decompE:
  assumes xs @ a # ys = map f zs
  obtains us b vs where
    zs = us @ b # vs xs = map f us a = f b ys = map f vs
proof −
  have mapped: map f zs = xs @ (a # ys) using assms by simp
  have ∃ us ws. zs = us @ ws ∧ xs = map f us ∧
      a # ys = map f ws
    by (rule map-eq-append-conv[THEN iffD1, OF mapped])
  then obtain us ws where
    zs: zs = us @ ws and xs: xs = map f us
    and rest: a # ys = map f ws
    by (elim exE conjE)
  from rest obtain b vs where ws: ws = b # vs
    and a: a = f b and ys: ys = map f vs
    by (cases ws) auto
  from zs xs ws a ys that show thesis by simp
qed

lemma red-vrenameE-aux:
  assumes step: red S a v
  shows ⋀q f. regular S q ⟹ a = vrename f q ⟹
    ∃ u. red S q u ∧ v = vrename f u
  using step
proof (induct arbitrary: q f rule: red.induct)
  case (beta u A t)
  from beta show ?case
  proof (cases q)
    case (App x y)
    from beta App show ?thesis
    proof (cases x)
      case (Lam A' body)
      from beta App Lam have
```

```
      type: A' = A
      and body: t = vrename (up-ren f) body
      and arg: u = vrename f y
      by auto
    have red: red S (App (Lam A' body) y) (subst 0 y body)
    proof (rule red.beta)
      have regy: regular S y using beta.prems App Lam by simp
      have dims: variable-dims y
        by (rule regular-variable-dims[OF regy])
      show dclosed0 S y
        using beta.hyps(1) arg dclosed0-vrename[OF dims, of S f]
        by simp
    qed
    have eq: subst 0 u t = vrename f (subst 0 y body)
      using body arg by (simp add: vrename-subst0)
    from red eq App Lam show ?thesis by auto
  qed auto
 qed auto
next
 case (elimPoint k h C pms lms as)
 from elimPoint show ?case
 proof (cases q)
  case (HElim h' C' pms' lms' scrut)
  from elimPoint HElim show ?thesis
  proof (cases scrut)
   case (PCon h'' k' as')
   with HElim elimPoint show ?thesis
    by (auto simp: vrename-elimArgs intro: red.elimPoint)
  qed auto
 qed auto
next
 case (elimPath k h C pms lms as j)
 from elimPath show ?case
 proof (cases q)
  case (HElim h' C' pms' lms' scrut)
  from elimPath HElim show ?thesis
  proof (cases scrut)
   case (LCon h'' k' as' d)
   from elimPath HElim LCon show ?thesis
   proof (cases d)
    case (DVar j')
    have klm: k' < length lms'
     using DVar LCon HElim elimPath by simp
    have mreg: regular S (lms' ! k')
     using DVar LCon HElim elimPath klm
     by (auto simp: list-all-iff)
    have mdims: variable-dims (lms' ! k')
     by (rule regular-variable-dims[OF mreg])
    let ?u = apps (dsubst S 0 (DVar j') (lms' ! k'))
```

```
        (elimArgs h'' C' pms' lms' (lcons (S h'') ! k') as')
      have r: red S
        (HElim h'' C' pms' lms'
          (LCon h'' k' as' (DVar j'))) ?u
        by (rule red.elimPath) (use DVar LCon HElim elimPath in simp)
      have eq:
        vrename f ?u =
          apps (dsubst S 0 (DVar j') (vrename f (lms' ! k')))
            (elimArgs h'' C' (map (vrename f) pms')
              (map (vrename f) lms') (lcons (S h'') ! k')
              (map (vrename f) as'))
        by (simp add: vrename-elimArgs
          vrename-dsubst-variable[OF mdims])
      from r eq DVar LCon HElim elimPath show ?thesis by auto
    qed auto
  qed auto
next
  case (appL t t' u)
  from appL show ?case
  proof (cases q)
    case (App x y)
    have regx: regular S x using appL App by simp
    obtain x' where r: red S x x' and eq: t' = vrename f x'
      using appL.hyps(2)[OF regx, of f] appL App by auto
    have red S (App x y) (App x' y) by (rule red.appL[OF r])
    with eq appL App show ?thesis by auto
  qed auto
next
  case (appR u u' t)
  from appR show ?case
  proof (cases q)
    case (App x y)
    have regy: regular S y using appR App by simp
    obtain y' where r: red S y y' and eq: u' = vrename f y'
      using appR.hyps(2)[OF regy, of f] appR App by auto
    have red S (App x y) (App x y') by (rule red.appR[OF r])
    with eq appR App show ?thesis by auto
  qed auto
next
  case (lam t t' A)
  from lam show ?case
  proof (cases q)
    case (Lam B body)
    have regbody: regular S body using lam Lam by simp
    obtain body' where r: red S body body'
      and eq: t' = vrename (up-ren f) body'
      using lam.hyps(2)[OF regbody, of up-ren f] lam Lam by auto
    have red S (Lam B body) (Lam B body') by (rule red.lam[OF r])
```

```
    with eq lam Lam show ?thesis by auto
  qed auto
next
  case (pconArg a a' h k xs ys)
  from pconArg show ?case
  proof (cases q)
    case (PCon h' k' as)
    from pconArg PCon obtain us b vs where
      as: as = us @ b # vs and xs: xs = map (vrename f) us
      and a: a = vrename f b and ys: ys = map (vrename f) vs
      by (auto elim: map-decompE)
    have regb: regular S b
      using pconArg PCon as by (auto simp: list-all-iff)
    obtain b' where r: red S b b' and eq: a' = vrename f b'
      using pconArg.hyps(2)[OF regb a] by blast
    have red: red S (PCon h k (us @ b # vs))
        (PCon h k (us @ b' # vs))
      by (rule red.pconArg[OF r])
    from red eq pconArg PCon as xs a ys show ?thesis by auto
  qed auto
next
  case (lconArg a a' h k xs ys d)
  from lconArg show ?case
  proof (cases q)
    case (LCon h' k' as d')
    from lconArg LCon obtain us b vs where
      as: as = us @ b # vs and xs: xs = map (vrename f) us
      and a: a = vrename f b and ys: ys = map (vrename f) vs
      by (auto elim: map-decompE)
    have regb: regular S b
      using lconArg LCon as by (auto simp: list-all-iff)
    obtain b' where r: red S b b' and eq: a' = vrename f b'
      using lconArg.hyps(2)[OF regb a] by blast
    have red: red S (LCon h k (us @ b # vs) d)
        (LCon h k (us @ b' # vs) d)
      by (rule red.lconArg[OF r])
    from red eq lconArg LCon as xs a ys show ?thesis by auto
  qed auto
next
  case (elimPm m m' h C xs ys lms t)
  from elimPm show ?case
  proof (cases q)
    case (HElim h' C' pms' lms' t')
    from elimPm HElim obtain us b vs where
      pms: pms' = us @ b # vs and xs: xs = map (vrename f) us
      and m: m = vrename f b and ys: ys = map (vrename f) vs
      by (auto elim: map-decompE)
    have regb: regular S b
      using elimPm HElim pms by (auto simp: list-all-iff)
```

```
    obtain b' where r: red S b b' and eq: m' = vrename f b'
      using elimPm.hyps(2)[OF regb m] by blast
    have red: red S
        (HElim h C (us @ b # vs) lms' t')
        (HElim h C (us @ b' # vs) lms' t')
      by (rule red.elimPm[OF r])
    from red eq elimPm HElim pms xs m ys show ?thesis by auto
  qed auto
next
  case (elimLm m m' h C pms xs ys t)
  from elimLm show ?case
  proof (cases q)
    case (HElim h' C' pms' lms' t')
    from elimLm HElim obtain us b vs where
      lms: lms' = us @ b # vs and xs: xs = map (vrename f) us
      and m: m = vrename f b and ys: ys = map (vrename f) vs
      by (auto elim: map-decompE)
    have regb: regular S b
      using elimLm HElim lms by (auto simp: list-all-iff)
    obtain b' where r: red S b b' and eq: m' = vrename f b'
      using elimLm.hyps(2)[OF regb m] by blast
    have red: red S
        (HElim h C pms' (us @ b # vs) t')
        (HElim h C pms' (us @ b' # vs) t')
      by (rule red.elimLm[OF r])
    from red eq elimLm HElim lms xs m ys show ?thesis by auto
  qed auto
next
  case (elimScrutinee t t' h C pms lms)
  from elimScrutinee show ?case
  proof (cases q)
    case (HElim h' C' pms' lms' body)
    have regbody: regular S body using elimScrutinee HElim by simp
    obtain body' where r: red S body body'
      and eq: t' = vrename f body'
      using elimScrutinee.hyps(2)[OF regbody, of f] elimScrutinee HElim
      by auto
    have red: red S (HElim h C pms' lms' body)
        (HElim h C pms' lms' body')
      by (rule red.elimScrutinee[OF r])
    from red eq elimScrutinee HElim show ?thesis by auto
  qed auto
qed

lemma red-vrenameE:
  assumes reg: regular S q and step: red S (vrename f q) v
  obtains u where red S q u and v = vrename f u
  using red-vrenameE-aux[OF step reg refl] that by blast
```

definition *erased* :: *sigenv* ⇒ *tm* ⇒ *bool* **where**
erased S t ⟷ *erase-dims S t* = *t*

lemma *map-eq-self-iff*:
map f xs = *xs* ⟷ (∀ *x* ∈ *set xs*. *f x* = *x*)
by (*induct xs*) *auto*

lemma *erased-red*:
assumes *step*: *red S t u* **and** *reg*: *regular S t* **and** *fixed*: *erased S t*
shows *erased S u*
using *step reg fixed*
proof (*induct rule*: *red.induct*)
case (*beta u A t*)
then show *?case*
unfolding *erased-def* **by** *simp*
next
case (*elimPoint k h C pms lms as*)
have *fpms*: ∀ *m* ∈ *set pms*. *erase-dims S m* = *m*
using *elimPoint*(*3*) **unfolding** *erased-def*
by (*simp add*: *map-eq-self-iff*)
have *flms*: ∀ *m* ∈ *set lms*. *erase-dims S m* = *m*
using *elimPoint*(*3*) **unfolding** *erased-def*
by (*simp add*: *map-eq-self-iff*)
have *fas*: ∀ *a* ∈ *set as*. *erase-dims S a* = *a*
using *elimPoint*(*3*) **unfolding** *erased-def*
by (*simp add*: *map-eq-self-iff*)
have *kpm*: *k* < *length pms* **using** *elimPoint* **by** *simp*
have *method*: *erase-dims S* (*pms* ! *k*) = *pms* ! *k*
using *fpms kpm* **by** *simp*
have *epms*: *map* (*erase-dims S*) *pms* = *pms*
using *fpms* **by** (*simp add*: *map-eq-self-iff*)
have *elms*: *map* (*erase-dims S*) *lms* = *lms*
using *flms* **by** (*simp add*: *map-eq-self-iff*)
have *eas*: *map* (*erase-dims S*) *as* = *as*
using *fas* **by** (*simp add*: *map-eq-self-iff*)
show *?case*
unfolding *erased-def*
by (*simp add*: *erase-dims-elimArgs method epms elms eas*)
next
case (*elimPath k h C pms lms as j*)
have *fpms*: ∀ *m* ∈ *set pms*. *erase-dims S m* = *m*
using *elimPath*(*3*) **unfolding** *erased-def*
by (*simp add*: *map-eq-self-iff*)
have *flms*: ∀ *m* ∈ *set lms*. *erase-dims S m* = *m*
using *elimPath*(*3*) **unfolding** *erased-def*
by (*simp add*: *map-eq-self-iff*)
have *fas*: ∀ *a* ∈ *set as*. *erase-dims S a* = *a*
using *elimPath*(*3*) **unfolding** *erased-def*
by (*simp add*: *map-eq-self-iff*)

```
  have klm: k < length lms using elimPath by simp
  have method-fixed: erase-dims S (lms ! k) = lms ! k
    using flms klm by simp
  have method-regular: regular S (lms ! k)
    using elimPath klm by (auto simp: list-all-iff)
  have method:
      erase-dims S (dsubst S 0 (DVar j) (lms ! k)) = lms ! k
    using erase-dims-dsubst-variable[
        OF regular-variable-dims[OF method-regular], of S 0 j]
      method-fixed
    by simp
  have method-same:
      dsubst S 0 (DVar j) (lms ! k) = lms ! k
  proof −
    have dsubst S 0 (DVar j) (lms ! k) =
        dsubst S 0 (DVar j) (erase-dims S (lms ! k))
      using method-fixed by simp
    also have ... = erase-dims S (lms ! k) by simp
    also have ... = lms ! k by (rule method-fixed)
    finally show ?thesis .
  qed
  have epms: map (erase-dims S) pms = pms
    using fpms by (simp add: map-eq-self-iff)
  have elms: map (erase-dims S) lms = lms
    using flms by (simp add: map-eq-self-iff)
  have eas: map (erase-dims S) as = as
    using fas by (simp add: map-eq-self-iff)
  show ?case
    unfolding erased-def
    by (simp add: erase-dims-elimArgs method method-fixed method-same
        epms elms eas)
next
  case (appL t t' u)
  then show ?case unfolding erased-def by simp
next
  case (appR u u' t)
  then show ?case unfolding erased-def by simp
next
  case (lam t t' A)
  then show ?case unfolding erased-def by simp
next
  case (pconArg a a' h k xs ys)
  have fixed-a: erased S a
    using pconArg(4) unfolding erased-def
    by (simp add: map-eq-self-iff)
  have fixed-a': erased S a'
    by (rule pconArg.hyps(2)) (use pconArg fixed-a in auto)
  show ?case
    using pconArg(4) fixed-a'
```

unfolding *erased-def*
by (*simp add*: *map-eq-self-iff*)
next
case (*lconArg a a′ h k xs ys d*)
have *fixed-a*: *erased S a*
using *lconArg*(*4*) **unfolding** *erased-def*
by (*simp add*: *map-eq-self-iff*)
have *fixed-a′*: *erased S a′*
by (*rule lconArg.hyps*(*2*)) (*use lconArg fixed-a* **in** *auto*)
show *?case*
using *lconArg*(*4*) *fixed-a′*
unfolding *erased-def*
by (*simp add*: *map-eq-self-iff*)
next
case (*elimPm m m′ h C xs ys lms t*)
have *fixed-m*: *erased S m*
using *elimPm*(*4*) **unfolding** *erased-def*
by (*simp add*: *map-eq-self-iff*)
have *fixed-m′*: *erased S m′*
by (*rule elimPm.hyps*(*2*)) (*use elimPm fixed-m* **in** *auto*)
show *?case*
using *elimPm*(*4*) *fixed-m′*
unfolding *erased-def*
by (*simp add*: *map-eq-self-iff*)
next
case (*elimLm m m′ h C pms xs ys t*)
have *fixed-m*: *erased S m*
using *elimLm*(*4*) **unfolding** *erased-def*
by (*simp add*: *map-eq-self-iff*)
have *fixed-m′*: *erased S m′*
by (*rule elimLm.hyps*(*2*)) (*use elimLm fixed-m* **in** *auto*)
show *?case*
using *elimLm*(*4*) *fixed-m′*
unfolding *erased-def*
by (*simp add*: *map-eq-self-iff*)
next
case (*elimScrutinee t t′ h C pms lms*)
have *fixed-t*: *erased S t*
using *elimScrutinee*(*4*) **unfolding** *erased-def* **by** *simp*
have *fixed-t′*: *erased S t′*
by (*rule elimScrutinee.hyps*(*2*)) (*use elimScrutinee fixed-t* **in** *auto*)
show *?case*
using *elimScrutinee*(*4*) *fixed-t′*
unfolding *erased-def* **by** *simp*
qed

15.3 Hereditary computability at HIT types

fun *neutral* :: *tm* ⇒ *bool* **where**

```
  neutral (Var i) = True
| neutral (App t u) = True
| neutral (HElim h C pms lms t) = True
| neutral - = False

lemma neutral-vrename:
  neutral t ⟹ neutral (vrename f t)
  by (cases t) simp-all

inductive hcomp :: sigenv ⇒ nat ⇒ tm ⇒ bool
  for S :: sigenv
where
  hneutral:
    neutral t ⟹
      (⋀u. red S t u ⟹ hcomp S h u) ⟹
      hcomp S h t
| hpcon:
    k < length (pcons (S h)) ⟹
      length as = length (pcons (S h) ! k) ⟹
      (⋀i. i < length as ⟹
        hcomp S (argHit h (pcons (S h) ! k ! i)) (as ! i)) ⟹
      (⋀u. red S (PCon h k as) u ⟹ hcomp S h u) ⟹
      hcomp S h (PCon h k as)
| hlcon:
    k < length (lcons (S h)) ⟹
      length as = length (lcons (S h) ! k) ⟹
      (⋀i. i < length as ⟹
        hcomp S (argHit h (lcons (S h) ! k ! i)) (as ! i)) ⟹
      (⋀u. red S (LCon h k as (DVar j)) u ⟹ hcomp S h u) ⟹
      hcomp S h (LCon h k as (DVar j))

lemma hcomp-step:
  hcomp S h t ⟹ red S t u ⟹ hcomp S h u
  by (induct rule: hcomp.induct) blast+

lemma hcomp-SN:
  hcomp S h t ⟹ SN S t
  by (induct rule: hcomp.induct) (blast intro: SN.SNI)+

lemma hcomp-Var: hcomp S h (Var i)
  by (rule hcomp.hneutral) (auto elim: red.cases)

inductive list-red :: sigenv ⇒ tm list ⇒ tm list ⇒ bool
  for S :: sigenv
where
  lr-head: red S a b ⟹ list-red S (a # xs) (b # xs)
| lr-tail: list-red S xs ys ⟹ list-red S (a # xs) (a # ys)

inductive SN-list :: sigenv ⇒ tm list ⇒ bool
```

```
  for S :: sigenv
where
  SNL: (⋀ys. list-red S xs ys ⟹ SN-list S ys) ⟹
    SN-list S xs

lemma SN-list-step:
  SN-list S xs ⟹ list-red S xs ys ⟹ SN-list S ys
  by (induct rule: SN-list.induct) blast

lemma SN-list-Cons:
  assumes head: SN S a and tail: SN-list S xs
  shows SN-list S (a # xs)
  using head tail
proof (induct arbitrary: xs rule: SN.induct)
  case (SNI a)
  show ?case
    using SNI.prems
  proof (induct rule: SN-list.induct)
    case (SNL xs)
    have snxs: SN-list S xs
      by (rule SN-list.SNL) (use SNL.hyps in blast)
    show ?case
    proof (rule SN-list.SNL)
      fix ys
      assume step: list-red S (a # xs) ys
      then show SN-list S ys
      proof cases
        case (lr-head b)
        from SNI.hyps(2) snxs lr-head show ?thesis by blast
      next
        case (lr-tail zs)
        from SNL.hyps lr-tail show ?thesis by blast
      qed
    qed
  qed
qed

lemma SN-list-Nil: SN-list S []
  by (rule SN-list.SNL) (auto elim: list-red.cases)

lemma SN-list-all:
  assumes list-all (SN S) xs
  shows SN-list S xs
  using assms
  by (induct xs) (auto intro: SN-list-Nil SN-list-Cons)

lemma SN-lists-induct [consumes 2, case-names step]:
  assumes xs: SN-list S xs and ys: SN-list S ys
    and step: ⋀xs ys.
```

```
    (⋀xs'. list-red S xs xs' ⟹ P xs' ys) ⟹
    (⋀ys'. list-red S ys ys' ⟹ P xs ys') ⟹
    P xs ys
  shows P xs ys
  using xs ys
proof (induct arbitrary: ys rule: SN-list.induct)
  case (SNL xs)
  note x-IH = SNL.hyps(2)
  show ?case
    using SNL.prems
  proof (induct rule: SN-list.induct)
    case (SNL ys)
    note y-IH = SNL.hyps(2)
    have snys: SN-list S ys
      by (rule SN-list.SNL) (use SNL.hyps(1) in blast)
    show ?case
    proof (rule step)
      fix xs'
      assume lr: list-red S xs xs'
      show P xs' ys by (rule x-IH[OF lr snys])
    next
      fix ys'
      assume lr: list-red S ys ys'
      show P xs ys' by (rule y-IH[OF lr])
    qed
  qed
qed

lemma list-red-append:
  red S a b ⟹
    list-red S (xs @ a # ys) (xs @ b # ys)
  by (induct xs) (auto intro: list-red.intros)

lemma red-PCon-list:
  assumes red S (PCon h k as) u
  obtains bs where u = PCon h k bs and list-red S as bs
  using assms
  by (cases rule: red.cases) (auto intro: list-red-append)

lemma red-LCon-list:
  assumes red S (LCon h k as d) u
  obtains bs where u = LCon h k bs d and list-red S as bs
  using assms
  by (cases rule: red.cases) (auto intro: list-red-append)

fun hargs ::
  sigenv ⇒ nat ⇒ argspec list ⇒ tm list ⇒ bool
where
  hargs S h [] [] = True
```

```
| hargs S h [] (a # as) = False
| hargs S h (s # ss) [] = False
| hargs S h (s # ss) (a # as) =
    (hcomp S (argHit h s) a ∧ hargs S h ss as)

lemma hargs-length:
  hargs S h ss as ⟹ length as = length ss
  by (induct ss arbitrary: as) (case-tac as; auto)+

lemma hargs-nth:
  assumes hargs S h ss as and i < length as
  shows hcomp S (argHit h (ss ! i)) (as ! i)
  using assms
proof (induct ss arbitrary: as i)
  case Nil
  then show ?case by (cases as) auto
next
  case (Cons s ss)
  then obtain a as' where as: as = a # as'
    by (cases as) auto
  show ?case
  proof (cases i)
    case 0
    with Cons.prems as show ?thesis by simp
  next
    case (Suc j)
    have tail: hargs S h ss as'
      using Cons.prems as by simp
    have bound: j < length as'
      using Cons.prems as Suc by simp
    have hcomp S (argHit h (ss ! j)) (as' ! j)
      by (rule Cons.hyps[OF tail bound])
    with as Suc show ?thesis by simp
  qed
qed

lemma hargsI:
  assumes length as = length ss
    and ⋀i. i < length as ⟹
      hcomp S (argHit h (ss ! i)) (as ! i)
  shows hargs S h ss as
  using assms
proof (induct ss arbitrary: as)
  case Nil
  then show ?case by (cases as) auto
next
  case (Cons s ss)
  then obtain a as' where as: as = a # as'
    by (cases as) auto
```

```
  have head: hcomp S (argHit h s) a
    using Cons.prems(2)[of 0] as by simp
  have tail: hargs S h ss as'
  proof (rule Cons.hyps)
    show length as' = length ss using Cons.prems(1) as by simp
    fix i
    assume i < length as'
    then have Suc i < length as using as by simp
    from Cons.prems(2)[OF this] as
    show hcomp S (argHit h (ss ! i)) (as' ! i) by simp
  qed
  from head tail as show ?case by simp
qed

lemma hargs-SN:
  assumes hargs S h ss as
  shows list-all (SN S) as
  using assms
  by (induct ss arbitrary: as)
    (case-tac as; auto dest: hcomp-SN)+

lemma hargs-list-red:
  assumes step: list-red S as bs and args: hargs S h ss as
  shows hargs S h ss bs
  using step args
proof (induct arbitrary: ss rule: list-red.induct)
  case (lr-head a b xs)
  then show ?case
    by (cases ss) (auto intro: hcomp-step)
next
  case (lr-tail xs ys a)
  then show ?case by (cases ss) auto
qed

lemma hcomp-PCon:
  assumes k: k < length (pcons (S h))
    and args: hargs S h (pcons (S h) ! k) as
  shows hcomp S h (PCon h k as)
proof −
  have snl: SN-list S as
    by (rule SN-list-all[OF hargs-SN[OF args]])
  from snl args show ?thesis
  proof (induct rule: SN-list.induct)
    case (SNL as)
    have len: length as = length (pcons (S h) ! k)
      by (rule hargs-length[OF SNL.prems])
    show ?case
    proof (rule hcomp.hpcon[
          where S = S and h = h and k = k and as = as, OF k len])
```

```
    fix i
    assume i < length as
    then show hcomp S (argHit h (pcons (S h) ! k ! i)) (as ! i)
      by (rule hargs-nth[OF SNL.prems])
  next
    fix u
    assume step: red S (PCon h k as) u
    then obtain bs where u: u = PCon h k bs
      and lr: list-red S as bs
      by (rule red-PCon-list)
    have hargs: hargs S h (pcons (S h) ! k) bs
      by (rule hargs-list-red[OF lr SNL.prems])
    have hcomp S h (PCon h k bs)
      by (rule SNL.hyps(2)[OF lr hargs])
    with u show hcomp S h u by simp
  qed
 qed
qed

lemma hcomp-LCon:
 assumes k: k < length (lcons (S h))
   and args: hargs S h (lcons (S h) ! k) as
 shows hcomp S h (LCon h k as (DVar j))
proof −
 have snl: SN-list S as
   by (rule SN-list-all[OF hargs-SN[OF args]])
 from snl args show ?thesis
 proof (induct rule: SN-list.induct)
   case (SNL as)
   have len: length as = length (lcons (S h) ! k)
     by (rule hargs-length[OF SNL.prems])
   show ?case
   proof (rule hcomp.hlcon[
         where S = S and h = h and k = k and as = as and j = j,
         OF k len])
     fix i
     assume i < length as
     then show hcomp S (argHit h (lcons (S h) ! k ! i)) (as ! i)
       by (rule hargs-nth[OF SNL.prems])
   next
     fix u
     assume step: red S (LCon h k as (DVar j)) u
     then obtain bs where u: u = LCon h k bs (DVar j)
       and lr: list-red S as bs
       by (rule red-LCon-list)
     have hargs: hargs S h (lcons (S h) ! k) bs
       by (rule hargs-list-red[OF lr SNL.prems])
     have hcomp S h (LCon h k bs (DVar j))
       by (rule SNL.hyps(2)[OF lr hargs])
```

```
    with u show hcomp S h u by simp
   qed
  qed
qed

lemma hcomp-vrename:
  assumes comp: hcomp S h t and reg: regular S t
  shows hcomp S h (vrename f t)
  using comp reg
proof (induct arbitrary: f rule: hcomp.induct)
  case (hneutral t h)
  show ?case
  proof (rule hcomp.hneutral)
   show neutral (vrename f t)
    by (rule neutral-vrename[OF hneutral.hyps(1)])
   fix v
   assume step: red S (vrename f t) v
   then obtain u where r: red S t u and v: v = vrename f u
    by (rule red-vrenameE[OF hneutral.prems])
   have regu: regular S u
    by (rule regular-red[OF r hneutral.prems])
   have hcomp S h (vrename f u)
    by (rule hneutral.hyps(3)[OF r regu])
   with v show hcomp S h v by simp
  qed
next
  case (hpcon k h as)
  have args: hargs S h (pcons (S h) ! k) (map (vrename f) as)
  proof (rule hargsI)
   show length (map (vrename f) as) = length (pcons (S h) ! k)
    using hpcon.hyps(2) by simp
   fix i
   assume i: i < length (map (vrename f) as)
   have ia: i < length as using i by simp
   have regi: regular S (as ! i)
    using hpcon.prems i by (auto simp: list-all-iff)
   have hcomp S (argHit h (pcons (S h) ! k ! i))
      (vrename f (as ! i))
    by (rule hpcon.hyps(4)[OF ia regi])
   with i show hcomp S (argHit h (pcons (S h) ! k ! i))
      ((map (vrename f) as) ! i)
    by simp
  qed
  have hcomp S h (PCon h k (map (vrename f) as))
   by (rule hcomp-PCon[OF hpcon.hyps(1) args])
  then show ?case by simp
next
  case (hlcon k h as j)
  have args: hargs S h (lcons (S h) ! k) (map (vrename f) as)
```

```
  proof (rule hargsI)
    show length (map (vrename f) as) = length (lcons (S h) ! k)
      using hlcon.hyps(2) by simp
    fix i
    assume i: i < length (map (vrename f) as)
    have ia: i < length as using i by simp
    have regi: regular S (as ! i)
      using hlcon.prems i by (auto simp: list-all-iff)
    have hcomp S (argHit h (lcons (S h) ! k ! i))
        (vrename f (as ! i))
      by (rule hlcon.hyps(4)[OF ia regi])
    with i show hcomp S (argHit h (lcons (S h) ! k ! i))
        ((map (vrename f) as) ! i)
      by simp
  qed
  have hcomp S h (LCon h k (map (vrename f) as) (DVar j))
    by (rule hcomp-LCon[OF hlcon.hyps(1) args])
  then show ?case by simp
qed

lemma SN-vrename:
  assumes sn: SN S t and reg: regular S t
  shows SN S (vrename f t)
  using sn reg
proof (induct arbitrary: f rule: SN.induct)
  case (SNI t)
  show ?case
  proof (rule SN.SNI)
    fix v
    assume step: red S (vrename f t) v
    then obtain u where r: red S t u and v: v = vrename f u
      by (rule red-vrenameE[OF SNI.prems])
    have regu: regular S u by (rule regular-red[OF r SNI.prems])
    have SN S (vrename f u) by (rule SNI.hyps(2)[OF r regu])
    with v show SN S v by simp
  qed
qed
```

15.4 A Kripke logical relation

```
primrec RC :: sigenv ⇒ ty ⇒ tm ⇒ bool where
  RC S (Hit h) t =
    (regular S t ∧ erased S t ∧ hcomp S h t)
| RC S (Fun A B) t =
    (regular S t ∧ erased S t ∧
      (∀ u. RC S A u ⟶ RC S B (App t u)))

lemma RC-regular:
  RC S A t ⟹ regular S t
```

```
  by (cases A) auto

lemma RC-erased:
  RC S A t ⟹ erased S t
  by (cases A) auto

definition CR1 :: sigenv ⇒ ty ⇒ bool where
  CR1 S A ⟷ (∀ t. RC S A t ⟶ SN S t)

definition CR2 :: sigenv ⇒ ty ⇒ bool where
  CR2 S A ⟷
    (∀ t u. RC S A t ⟶ red S t u ⟶ RC S A u)

definition CR3 :: sigenv ⇒ ty ⇒ bool where
  CR3 S A ⟷
    (∀ t. neutral t ⟶ regular S t ⟶ erased S t ⟶
      (∀ u. red S t u ⟶ RC S A u) ⟶ RC S A t)

lemma RC-props:
  CR1 S A CR2 S A CR3 S A
proof (induct A)
  case (Hit h)
  show CR1 S (Hit h)
    unfolding CR1-def by (auto intro: hcomp-SN)
  show CR2 S (Hit h)
    unfolding CR2-def
    by (auto intro: regular-red erased-red hcomp-step)
  show CR3 S (Hit h)
    unfolding CR3-def
    by (auto intro: hcomp.hneutral)
next
  case (Fun A B)
  show CR1 S (Fun A B)
    unfolding CR1-def
  proof (intro allI impI)
    fix t
    assume t: RC S (Fun A B) t
    have var: RC S A (Var 0)
    proof −
      have cr3: neutral (Var 0) ⟶ regular S (Var 0) ⟶
          erased S (Var 0) ⟶
          (∀ u. red S (Var 0) u ⟶ RC S A u) ⟶
          RC S A (Var 0)
        using Fun(3) unfolding CR3-def by blast
      show ?thesis
        by (rule cr3[rule-format])
          (auto simp: erased-def elim: red.cases)
    qed
    from t var have RC S B (App t (Var 0)) by simp
```

```
  then have SN S (App t (Var 0))
    using Fun(4) unfolding CR1-def by blast
  then show SN S t by (rule SN-App1)
qed
show CR2 S (Fun A B)
  unfolding CR2-def
proof (intro allI impI)
  fix t u
  assume t: RC S (Fun A B) t and step: red S t u
  have reg: regular S u
    by (rule regular-red[OF step]) (use t in simp)
  have fixed: erased S u
    by (rule erased-red[OF step]) (use t in simp-all)
  have app: ∀ x. RC S A x ⟶ RC S B (App u x)
  proof (intro allI impI)
    fix x
    assume x: RC S A x
    from t x have old: RC S B (App t x) by simp
    have red S (App t x) (App u x) by (rule red.appL[OF step])
    with old Fun(5) show RC S B (App u x)
      unfolding CR2-def by blast
  qed
  from reg fixed app show RC S (Fun A B) u by simp
qed
show CR3 S (Fun A B)
  unfolding CR3-def
proof (intro allI impI)
  fix t
  assume neu: neutral t and reg: regular S t and fixed: erased S t
    and reducts: ∀ u. red S t u ⟶ RC S (Fun A B) u
  have app: ∀ x. RC S A x ⟶ RC S B (App t x)
  proof (intro allI impI)
    fix x
    assume x: RC S A x
    have snx: SN S x
      using Fun(1) x unfolding CR1-def by blast
    from snx x show RC S B (App t x)
    proof (induct rule: SN.induct)
      case (SNI x)
      have regx: regular S x by (rule RC-regular[OF SNI.prems])
      have fixedx: erased S x by (rule RC-erased[OF SNI.prems])
      have cr3: neutral (App t x) ⟶
          regular S (App t x) ⟶ erased S (App t x) ⟶
          (∀ u. red S (App t x) u ⟶ RC S B u) ⟶
          RC S B (App t x)
        using Fun(6) unfolding CR3-def by blast
      have red-app: ⋀u. red S (App t x) u ⟹ RC S B u
      proof −
        fix u
```

```
      assume step: red S (App t x) u
      then show RC S B u
      proof cases
        case beta
        with neu show ?thesis by simp
      next
        case (appL q')
        from reducts appL have RC S (Fun A B) q' by blast
        with SNI.prems appL show ?thesis by simp
      next
        case (appR q')
        have x': RC S A q'
          using Fun(2) SNI.prems appR unfolding CR2-def by blast
        have RC S B (App t q')
          using SNI.hyps(2) appR x' by blast
        with appR show ?thesis by blast
      qed
    qed
    show ?case
    proof (rule cr3[rule-format])
      show neutral (App t x) by simp
      show regular S (App t x) using reg regx by simp
      show erased S (App t x)
        using fixed fixedx red-app unfolding erased-def by simp
      fix u
      assume red S (App t x) u
      then show RC S B u by (rule red-app)
    qed
  qed
  from reg fixed app show RC S (Fun A B) t by simp
 qed
qed

lemma RC-SN:
 RC S A t ⟹ SN S t
 using RC-props(1) unfolding CR1-def by blast

lemma RC-red:
 RC S A t ⟹ red S t u ⟹ RC S A u
 using RC-props(2) unfolding CR2-def by blast

lemma RC-neutral:
 assumes neutral t regular S t erased S t
   ⋀u. red S t u ⟹ RC S A u
 shows RC S A t
 using assms RC-props(3) unfolding CR3-def by blast

definition KR :: sigenv ⇒ nat ⇒ ty ⇒ tm ⇒ bool where
```

```
  KR S n A t ⟷
    (∀ m f. Ren n m f ⟶ RC S A (vrename f t))

lemma KR-core:
  KR S n A t ⟹ RC S A t
proof −
  assume rel: KR S n A t
  from rel have all: ∀ m f. Ren n m f ⟶ RC S A (vrename f t)
    unfolding KR-def by blast
  from all[rule-format, OF Ren-id[of n]] show RC S A t by simp
qed

lemma KR-regular:
  KR S n A t ⟹ regular S t
proof −
  assume rel: KR S n A t
  have RC S A t by (rule KR-core[OF rel])
  then show regular S t by (cases A) auto
qed

lemma KR-erased:
  KR S n A t ⟹ erased S t
proof −
  assume rel: KR S n A t
  have RC S A t by (rule KR-core[OF rel])
  then show erased S t by (cases A) auto
qed

lemma KR-SN:
  KR S n A t ⟹ SN S t
  by (auto intro: RC-SN dest: KR-core)

lemma KR-Hit-hcomp:
  KR S n (Hit h) t ⟹ hcomp S h t
  by (auto dest: KR-core)

lemma KR-App:
  assumes KR S n (Fun A B) t and KR S n A u
  shows KR S n B (App t u)
  unfolding KR-def
proof (intro allI impI)
  fix m f
  assume ren: Ren n m f
  from assms ren have
    RC S (Fun A B) (vrename f t) RC S A (vrename f u)
    unfolding KR-def by blast+
  then show RC S B (vrename f (App t u)) by simp
qed
```

lemma *KR-vrename*:
assumes *KR S n A t* **and** *Ren n m f*
shows *KR S m A* (*vrename f t*)
unfolding *KR-def*
proof (*intro allI impI*)
fix *k g*
assume *Ren m k g*
then have *Ren n k* (*g* ∘ *f*) **by** (*rule Ren-comp*[*OF assms*(*2*)])
from *assms*(*1*) *this* **show** *RC S A* (*vrename g* (*vrename f t*))
unfolding *KR-def* **by** (*simp add*: *vrename-comp*)
qed

lemma *KR-red*:
assumes *rel*: *KR S n A t* **and** *step*: *red S t u*
shows *KR S n A u*
unfolding *KR-def*
proof (*intro allI impI*)
fix *m f*
assume *ren*: *Ren n m f*
from *rel ren* **have** *source*: *RC S A* (*vrename f t*)
unfolding *KR-def* **by** *blast*
have *r*: *red S* (*vrename f t*) (*vrename f u*)
by (*rule red-vrename*[*OF step KR-regular*[*OF rel*]])
show *RC S A* (*vrename f u*) **by** (*rule RC-red*[*OF source r*])
qed

lemma *KR-reds*:
assumes *KR S n A t* **and** *reds S t u*
shows *KR S n A u*
using *assms*(*2*,*1*)
by (*induct rule*: *rtranclp-induct*) (*auto intro*: *KR-red*)

lemma *KR-neutral*:
assumes *reg*: *regular S t* **and** *fixed*: *erased S t* **and** *neu*: *neutral t*
and *reducts*: ⋀*m f u*. *Ren n m f* ⟹
red S (*vrename f t*) *u* ⟹ *RC S A u*
shows *KR S n A t*
unfolding *KR-def*
proof (*intro allI impI*)
fix *m f*
assume *ren*: *Ren n m f*
show *RC S A* (*vrename f t*)
proof (*rule RC-neutral*)
show *neutral* (*vrename f t*) **by** (*rule neutral-vrename*[*OF neu*])
show *regular S* (*vrename f t*) **using** *reg* **by** *simp*
show *erased S* (*vrename f t*)
using *fixed* **unfolding** *erased-def* **by** *simp*
fix *u*
assume *red S* (*vrename f t*) *u*

then show *RC S A u* **by** (*rule reducts*[*OF ren*])
qed
qed

15.5 The hereditary Kripke candidate

primrec *KC* :: *sigenv* ⇒ *nat* ⇒ *ty* ⇒ *tm* ⇒ *bool* **where**
KC S n (*Hit h*) *t* =
(*regular S t* ∧ *erased S t* ∧ *hcomp S h t*)
| *KC S n* (*Fun A B*) *t* =
(*regular S t* ∧ *erased S t* ∧ *SN S t* ∧
(∀ *m f u*. *Ren n m f* ⟶ *KC S m A u* ⟶
KC S m B (*App* (*vrename f t*) *u*)))

lemma *KC-regular*:
KC S n A t ⟹ *regular S t*
by (*cases A*) *auto*

lemma *KC-erased*:
KC S n A t ⟹ *erased S t*
by (*cases A*) *auto*

lemma *KC-SN*:
KC S n A t ⟹ *SN S t*
by (*cases A*) (*auto intro*: *hcomp-SN*)

lemma *KC-Hit-hcomp*:
KC S n (*Hit h*) *t* ⟹ *hcomp S h t*
by *simp*

lemma *erased-vrename*:
erased S t ⟹ *erased S* (*vrename f t*)
unfolding *erased-def* **by** *simp*

lemma *KC-vrename*:
assumes *rel*: *KC S n A t* **and** *ren*: *Ren n m f*
shows *KC S m A* (*vrename f t*)
using *rel ren*
proof (*induct A arbitrary*: *n m f t*)
case (*Hit h*)
have *reg*: *regular S t* **using** *Hit.prems* **by** *simp*
have *comp*: *hcomp S h t* **using** *Hit.prems* **by** *simp*
have *hcomp S h* (*vrename f t*)
by (*rule hcomp-vrename*[*OF comp reg*])
moreover have *erased S* (*vrename f t*)
by (*rule erased-vrename*) (*use Hit.prems* **in** *simp*)
ultimately show *?case* **using** *Hit.prems* **by** *simp*
next
case (*Fun A B*)

```
  have reg: regular S t using Fun.prems by simp
  have fixed: erased S t using Fun.prems by simp
  have sn: SN S t using Fun.prems by simp
  have sn': SN S (vrename f t) by (rule SN-vrename[OF sn reg])
  have fixed': erased S (vrename f t)
    by (rule erased-vrename[OF fixed])
  have app: ∀ k g u. Ren m k g ⟶ KC S k A u ⟶
      KC S k B (App (vrename g (vrename f t)) u)
  proof (intro allI impI)
    fix k g u
    assume g: Ren m k g and u: KC S k A u
    have comp: Ren n k (g ∘ f) by (rule Ren-comp[OF Fun.prems(2) g])
    from Fun.prems(1) comp u
    have KC S k B (App (vrename (g ∘ f) t) u) by simp
    then show KC S k B (App (vrename g (vrename f t)) u)
      by (simp add: vrename-comp)
  qed
  from reg fixed' sn' app show ?case by simp
qed

lemma KC-App:
  assumes KC S n (Fun A B) t and KC S n A u
  shows KC S n B (App t u)
proof −
  from assms(1) have app: ∀ m f u. Ren n m f ⟶
      KC S m A u ⟶ KC S m B (App (vrename f t) u)
    by simp
  from app[rule-format, OF Ren-id[of n] assms(2)]
  show ?thesis by simp
qed

lemma KC-red:
  assumes rel: KC S n A t and step: red S t u
  shows KC S n A u
  using rel step
proof (induct A arbitrary: n t u)
  case (Hit h)
  then show ?case
    by (auto intro: regular-red erased-red hcomp-step)
next
  case (Fun A B)
  have reg: regular S u
    by (rule regular-red[OF Fun.prems(2) KC-regular[OF Fun.prems(1)]])
  have fixed: erased S u
    by (rule erased-red[
        OF Fun.prems(2) KC-regular[OF Fun.prems(1)]
          KC-erased[OF Fun.prems(1)]])
  have sn: SN S u by (rule SN-step[OF KC-SN[OF Fun.prems(1)] Fun.prems(2)])
  have app: ∀ m f x. Ren n m f ⟶ KC S m A x ⟶
```

```
    KC S m B (App (vrename f u) x)
  proof (intro allI impI)
    fix m f x
    assume ren: Ren n m f and x: KC S m A x
    from Fun.prems(1) ren x
    have old: KC S m B (App (vrename f t) x) by simp
    have r: red S (vrename f t) (vrename f u)
      by (rule red-vrename[OF Fun.prems(2) KC-regular[OF Fun.prems(1)]])
    have red S (App (vrename f t) x) (App (vrename f u) x)
      by (rule red.appL[OF r])
    then show KC S m B (App (vrename f u) x)
      by (rule Fun.hyps(2)[OF old])
  qed
  from reg fixed sn app show ?case by simp
qed

lemma KC-reds:
  assumes KC S n A t and reds S t u
  shows KC S n A u
  using assms(2,1)
  by (induct rule: rtranclp-induct) (auto intro: KC-red)

lemma KC-neutral:
  assumes neu: neutral t and reg: regular S t and fixed: erased S t
    and reducts: ⋀u. red S t u ⟹ KC S n A u
  shows KC S n A t
  using neu reg fixed reducts
proof (induct A arbitrary: n t)
  case (Hit h)
  have hcomp S h t
    by (rule hcomp.hneutral[OF Hit.prems(1)])
      (use Hit.prems(4) in auto)
  with Hit.prems show ?case by simp
next
  case (Fun A B)
  have sn: SN S t
  proof (rule SN.SNI)
    fix u
    assume red S t u
    then have KC S n (Fun A B) u by (rule Fun.prems(4))
    then show SN S u by (rule KC-SN)
  qed
  have app: ∀ m f x. Ren n m f ⟶ KC S m A x ⟶
      KC S m B (App (vrename f t) x)
  proof (intro allI impI)
    fix m f x
    assume ren: Ren n m f and x: KC S m A x
    have snx: SN S x by (rule KC-SN[OF x])
    from snx x show KC S m B (App (vrename f t) x)
```

```
    proof (induct rule: SN.induct)
      case (SNI x)
      have head-reg: regular S (vrename f t)
        using Fun.prems(2) by simp
      have head-fixed: erased S (vrename f t)
        using Fun.prems(3) unfolding erased-def by simp
      have app-reg: regular S (App (vrename f t) x)
        using head-reg KC-regular[OF SNI.prems] by simp
      have app-fixed: erased S (App (vrename f t) x)
        using head-fixed KC-erased[OF SNI.prems]
        unfolding erased-def by simp
      show ?case
      proof (rule Fun.hyps(2)[OF - app-reg app-fixed])
        show neutral (App (vrename f t) x) by simp
        fix y
        assume step: red S (App (vrename f t) x) y
        then show KC S m B y
        proof cases
          case beta
          have neutral (vrename f t)
            by (rule neutral-vrename[OF Fun.prems(1)])
          with beta have False by simp
          then show ?thesis by blast
        next
          case (appL q')
          obtain t' where r: red S t t' and q': q' = vrename f t'
            using red-vrenameE[OF Fun.prems(2) appL(2)] by blast
          have rel': KC S n (Fun A B) t' by (rule Fun.prems(4)[OF r])
          have KC S m (Fun A B) (vrename f t')
            by (rule KC-vrename[OF rel' ren])
          with q' have head': KC S m (Fun A B) q' by simp
          have KC S m B (App q' x)
            by (rule KC-App[OF head' SNI.prems])
          with appL show ?thesis by simp
        next
          case (appR q')
          have x': KC S m A q' by (rule KC-red[OF SNI.prems appR(2)])
          have KC S m B (App (vrename f t) q')
            by (rule SNI.hyps(2)[OF appR(2) x'])
          with appR show ?thesis by simp
        qed
      qed
    qed
    from Fun.prems(2,3) sn app show ?case by simp
qed

lemma KC-Var: KC S n A (Var i)
proof (rule KC-neutral)
```

show *neutral* (*Var i*) **by** *simp*
show *regular S* (*Var i*) **by** *simp*
show *erased S* (*Var i*) **unfolding** *erased-def* **by** *simp*
fix *u*
assume *red S* (*Var i*) *u*
then show *KC S n A u* **by** (*cases rule*: *red.cases*)
qed

15.6 Substitution and reducibility

lemma *subst-lift-same* [*simp*]:
subst n s (*lift n t*) = *t*
by (*induct t arbitrary*: *n s*) (*simp-all add*: *map-idI*)

lemma *shift-ren-zero* [*simp*]: *shift-ren 0* = *Suc*
unfolding *shift-ren-def* **by** (*rule ext*) *simp*

lemma *lift-zero-as-vrename*: *lift 0 t* = *vrename Suc t*
by (*simp add*: *lift-as-vrename*)

lemma *psubst-lift0*:
psubst (*up-sub s*) (*lift 0 t*) = *lift 0* (*psubst s t*)
proof −
have *env*:
up-sub s ∘ *shift-ren 0* =
(λ*i*. *vrename* (*shift-ren 0*) (*s i*))
proof
fix *i*
show (*up-sub s* ∘ *shift-ren 0*) *i* =
vrename (*shift-ren 0*) (*s i*)
by (*simp add*: *shift-ren-def lift-zero-as-vrename*)
qed
have *psubst* (*up-sub s*) (*lift 0 t*) =
psubst (*up-sub s*) (*vrename* (*shift-ren 0*) *t*)
by (*simp add*: *lift-as-vrename*)
also have ... = *psubst* (*up-sub s* ∘ *shift-ren 0*) *t*
by (*rule psubst-vrename*)
also have ... =
psubst (λ*i*. *vrename* (*shift-ren 0*) (*s i*)) *t*
by (*simp only*: *env*)
also have ... = *vrename* (*shift-ren 0*) (*psubst s t*)
by (*simp add*: *vrename-psubst*)
also have ... = *lift 0* (*psubst s t*)
by (*simp add*: *lift-as-vrename*)
finally show *?thesis* **.**
qed

lemma *up-sub-psubst-comp*:
(λ*i*. *psubst* (*up-sub r*) (*up-sub s i*)) =

```
    up-sub (λi. psubst r (s i))
proof
  fix i
  show psubst (up-sub r) (up-sub s i) =
      up-sub (λi. psubst r (s i)) i
    by (cases i) (simp-all add: psubst-lift0)
qed

lemma psubst-comp:
  psubst r (psubst s t) =
    psubst (λi. psubst r (s i)) t
  by (induct t arbitrary: r s)
    (simp-all add: up-sub-psubst-comp map-map)

lemma subst0-psubst:
  subst 0 (psubst s u) (psubst (up-sub s) t) =
    psubst s (subst 0 u t)
proof −
  have env:
      (λi. psubst (single-sub 0 (psubst s u)) (up-sub s i)) =
       (λi. psubst s (single-sub 0 u i))
  proof
    fix i
    show psubst (single-sub 0 (psubst s u)) (up-sub s i) =
        psubst s (single-sub 0 u i)
    proof (cases i)
      case 0
      then show ?thesis by (simp add: single-sub-def)
    next
      case (Suc j)
      have psubst (single-sub 0 (psubst s u)) (lift 0 (s j)) = s j
        by (simp add: subst-as-psubst[symmetric])
      with Suc show ?thesis by (simp add: single-sub-def)
    qed
  qed
  show ?thesis
    by (simp add: subst-as-psubst psubst-comp env)
qed

lemma subst0-psubst-up:
  subst 0 x (psubst (up-sub s) t) =
    psubst (ext-sub x s) t
proof −
  have env:
      (λi. psubst (single-sub 0 x) (up-sub s i)) = ext-sub x s
  proof
    fix i
    show psubst (single-sub 0 x) (up-sub s i) = ext-sub x s i
    proof (cases i)
```

```
    case 0
    then show ?thesis by (simp add: single-sub-def ext-sub-def)
  next
    case (Suc j)
    have psubst (single-sub 0 x) (lift 0 (s j)) = s j
      by (simp add: subst-as-psubst[symmetric])
    with Suc show ?thesis by (simp add: ext-sub-def)
  qed
 qed
 show ?thesis
  by (simp add: subst-as-psubst psubst-comp env)
qed

lemma psubst-apps [simp]:
 psubst s (apps t xs) = apps (psubst s t) (map (psubst s) xs)
 by (induct xs arbitrary: t) simp-all

lemma psubst-elimArgs:
 map (psubst s) (elimArgs h C pms lms ss as) =
  elimArgs h C (map (psubst s) pms) (map (psubst s) lms) ss
   (map (psubst s) as)
 by (induct ss arbitrary: as) (auto split: list.split argspec.split)

lemma erased-dsubst-zero:
 assumes erased S t
 shows dsubst S 0 (DVar j) t = t
proof −
 have dsubst S 0 (DVar j) t =
    dsubst S 0 (DVar j) (erase-dims S t)
  using assms unfolding erased-def by simp
 also have ... = erase-dims S t by simp
 also have ... = t using assms unfolding erased-def .
 finally show ?thesis .
qed

lemma erased-up-sub:
 assumes ⋀i. erased S (s i)
 shows erased S (up-sub s i)
 using assms
proof (cases i)
 case 0
 then show ?thesis unfolding erased-def by simp
next
 case (Suc j)
 have erase-dims S (s j) = s j
  using assms[of j] unfolding erased-def .
 with Suc show ?thesis unfolding erased-def by simp
qed
```

lemma *psubst-dsubst-zero*:
  assumes *dims*: *variable-dims t*
    and *env*: $\bigwedge i$. *erased S* (*s i*)
  shows *psubst s* (*dsubst S 0* (*DVar j*) *t*) =
    dsubst S 0 (*DVar j*) (*psubst s t*)
  using *dims env*
proof (*induct t arbitrary*: *s j*)
  case (*Var i*)
  then show *?case* **by** (*simp add*: *erased-dsubst-zero*)
next
  case (*Lam A t*)
  have *up*: $\bigwedge i$. *erased S* (*up-sub s i*)
    by (*rule erased-up-sub*[*OF Lam.prems*(*2*)])
  have *dims*: *variable-dims t* **using** *Lam.prems*(*1*) **by** *simp*
  from *Lam.hyps*[*OF dims up*] **show** *?case* **by** *simp*
next
  case (*LCon h k as d*)
  then obtain *q* **where** *d*: *d* = *DVar q* **by** *auto*
  from *LCon d* **show** *?case*
    by (*cases q* = *0*) (*auto simp*: *list-all-iff map-map*)
qed (*auto simp*: *list-all-iff map-map*)

lemma *dclosed0-psubst*:
  assumes *dims*: *variable-dims t*
    and *closed*: *dclosed0 S t*
    and *env*: $\bigwedge i$. *erased S* (*s i*)
  shows *dclosed0 S* (*psubst s t*)
proof (*unfold dclosed0-def*, *intro allI*)
  fix *j*
  have *dsubst S 0* (*DVar j*) (*psubst s t*) =
      psubst s (*dsubst S 0* (*DVar j*) *t*)
    using *psubst-dsubst-zero*[
      where *S* = *S* **and** *t* = *t* **and** *s* = *s* **and** *j* = *j*, *OF dims env*]
    by *simp*
  also have ... = *psubst s t*
    using *closed* **unfolding** *dclosed0-def* **by** *simp*
  finally show *dsubst S 0* (*DVar j*) (*psubst s t*) = *psubst s t* **.**
qed

lemma *red-psubst*:
  assumes *step*: *red S t u* **and** *reg*: *regular S t*
    and *env*: $\bigwedge i$. *regular S* (*s i*)
    and *fixed*: $\bigwedge i$. *erased S* (*s i*)
  shows *red S* (*psubst s t*) (*psubst s u*)
  using *step reg env fixed*
proof (*induct arbitrary*: *s rule*: *red.induct*)
  case (*beta u A t*)
  have *dimsu*: *variable-dims u*
    by (*rule regular-variable-dims*[**where** *S* = *S*]) (*use beta* **in** *simp*)

```
  have closed: dclosed0 S (psubst s u)
    by (rule dclosed0-psubst[OF dimsu beta.hyps(1) beta.prems(3)])
  have r: red S
      (App (Lam A (psubst (up-sub s) t)) (psubst s u))
      (subst 0 (psubst s u) (psubst (up-sub s) t))
    by (rule red.beta[OF closed])
  from r show ?case by (simp add: subst0-psubst)
next
  case (elimPoint k h C pms lms as)
  have kpm: k < length pms using elimPoint by simp
  have r: red S
      (HElim h C (map (psubst s) pms) (map (psubst s) lms)
        (PCon h k (map (psubst s) as)))
      (apps ((map (psubst s) pms) ! k)
        (elimArgs h C (map (psubst s) pms) (map (psubst s) lms)
          (pcons (S h) ! k) (map (psubst s) as)))
    by (rule red.elimPoint[
          where S = S and h = h and k = k and C = C
            and pms = map (psubst s) pms
            and lms = map (psubst s) lms
            and as = map (psubst s) as,
          OF elimPoint(1)])
  from r kpm show ?case by (simp add: psubst-elimArgs)
next
  case (elimPath k h C pms lms as j)
  have klm: k < length lms using elimPath by simp
  have mreg: regular S (lms ! k)
    using elimPath klm by (auto simp: list-all-iff)
  have mdims: variable-dims (lms ! k)
    by (rule regular-variable-dims[OF mreg])
  have method:
      psubst s (dsubst S 0 (DVar j) (lms ! k)) =
        dsubst S 0 (DVar j) (psubst s (lms ! k))
    by (rule psubst-dsubst-zero[OF mdims elimPath.prems(3)])
  have r: red S
      (HElim h C (map (psubst s) pms) (map (psubst s) lms)
        (LCon h k (map (psubst s) as) (DVar j)))
      (apps (dsubst S 0 (DVar j) ((map (psubst s) lms) ! k))
        (elimArgs h C (map (psubst s) pms) (map (psubst s) lms)
          (lcons (S h) ! k) (map (psubst s) as)))
    by (rule red.elimPath[
          where S = S and h = h and k = k and C = C
            and pms = map (psubst s) pms
            and lms = map (psubst s) lms
            and as = map (psubst s) as and j = j,
          OF elimPath(1)])
  from r klm method show ?case by (simp add: psubst-elimArgs)
next
  case (appL t t' u)
```

```
  then show ?case by (auto intro: red.appL)
next
  case (appR u u' t)
  then show ?case by (auto intro: red.appR)
next
  case (lam t t' A)
  have env': ⋀i. regular S (up-sub s i)
    by (rule regular-up-sub[OF lam.prems(2)])
  have fixed': ⋀i. erased S (up-sub s i)
    by (rule erased-up-sub[OF lam.prems(3)])
  have regt: regular S t using lam.prems(1) by simp
  have red S (psubst (up-sub s) t) (psubst (up-sub s) t')
    by (rule lam.hyps(2)[OF regt env' fixed'])
  then show ?case by (auto intro: red.lam)
next
  case (pconArg a a' h k xs ys)
  have rega: regular S a
    using pconArg.prems(1) by (auto simp: list-all-iff)
  have red S (psubst s a) (psubst s a')
    by (rule pconArg.hyps(2)[OF rega pconArg.prems(2,3)])
  then show ?case by (auto intro: red.pconArg)
next
  case (lconArg a a' h k xs ys d)
  have rega: regular S a
    using lconArg.prems(1) by (auto simp: list-all-iff)
  have red S (psubst s a) (psubst s a')
    by (rule lconArg.hyps(2)[OF rega lconArg.prems(2,3)])
  then show ?case by (auto intro: red.lconArg)
next
  case (elimPm m m' h C xs ys lms t)
  have regm: regular S m
    using elimPm.prems(1) by (auto simp: list-all-iff)
  have red S (psubst s m) (psubst s m')
    by (rule elimPm.hyps(2)[OF regm elimPm.prems(2,3)])
  then show ?case by (auto intro: red.elimPm)
next
  case (elimLm m m' h C pms xs ys t)
  have regm: regular S m
    using elimLm.prems(1) by (auto simp: list-all-iff)
  have red S (psubst s m) (psubst s m')
    by (rule elimLm.hyps(2)[OF regm elimLm.prems(2,3)])
  then show ?case by (auto intro: red.elimLm)
next
  case (elimScrutinee t t' h C pms lms)
  have regt: regular S t using elimScrutinee.prems(1) by simp
  have red S (psubst s t) (psubst s t')
    by (rule elimScrutinee.hyps(2)[
        OF regt elimScrutinee.prems(2,3)])
  then show ?case by (auto intro: red.elimScrutinee)
```

qed

lemma *red-subst0*:
assumes *step*: *red S t t′* **and** *regt*: *regular S t*
and *regs*: *regular S s* **and** *fixeds*: *erased S s*
shows *red S* (*subst 0 s t*) (*subst 0 s t′*)
proof −
have *env-reg*: ⋀*i*. *regular S* (*single-sub 0 s i*)
unfolding *single-sub-def*
using *regs* **by** (*case-tac i*; *simp*)
have *env-fixed*: ⋀*i*. *erased S* (*single-sub 0 s i*)
proof −
fix *i*
show *erased S* (*single-sub 0 s i*)
using *fixeds* **unfolding** *single-sub-def erased-def*
by (*cases i*) *simp-all*
qed
have *red S*
(*psubst* (*single-sub 0 s*) *t*)
(*psubst* (*single-sub 0 s*) *t′*)
by (*rule red-psubst*[*OF step regt env-reg env-fixed*])
then show *?thesis* **by** (*simp add*: *subst-as-psubst*)
qed

lemma *red-LamE*:
assumes *red S* (*Lam A t*) *u*
obtains *t′* **where** *u* = *Lam A t′* **and** *red S t t′*
using *assms* **by** (*cases rule*: *red.cases*) *auto*

lemma *SN-LamI*:
assumes *SN S t*
shows *SN S* (*Lam A t*)
using *assms*
proof (*induct rule*: *SN.induct*)
case (*SNI t*)
show *?case*
proof (*rule SN.SNI*)
fix *u*
assume *red S* (*Lam A t*) *u*
then obtain *t′* **where** *u*: *u* = *Lam A t′* **and** *r*: *red S t t′*
by (*rule red-LamE*)
have *SN S* (*Lam A t′*) **by** (*rule SNI.hyps*(*2*)[*OF r*])
with *u* **show** *SN S u* **by** *simp*
qed
qed

lemma *KC-beta-app*:
assumes *sn*: *SN S b* **and** *reg*: *regular S b* **and** *fixed*: *erased S b*
and *bodies*: ⋀*x*. *KC S n A x* ⟹

```
      KC S n B (subst 0 x b)
    and arg: KC S n A a
  shows KC S n B (App (Lam A b) a)
  using sn reg fixed bodies arg
proof (induct arbitrary: a rule: SN.induct)
  case (SNI b)
  note body-IH = SNI.hyps(2)
  have regb: regular S b by (rule SNI.prems(1))
  have fixedb: erased S b by (rule SNI.prems(2))
  have bodyfun: ⋀x. KC S n A x ⟹
      KC S n B (subst 0 x b)
    by (rule SNI.prems(3))
  have arga: KC S n A a by (rule SNI.prems(4))
  have sna: SN S a by (rule KC-SN[OF arga])
  from sna arga show ?case
  proof (induct rule: SN.induct)
    case (SNI a)
    have xrel: KC S n A a by (rule SNI.prems)
    have app-reg: regular S (App (Lam A b) a)
      using regb KC-regular[OF xrel] by simp
    have app-fixed: erased S (App (Lam A b) a)
      using fixedb KC-erased[OF xrel]
      unfolding erased-def by simp
    show ?case
    proof (rule KC-neutral[OF - app-reg app-fixed])
      show neutral (App (Lam A b) a) by simp
      fix u
      assume step: red S (App (Lam A b) a) u
      then show KC S n B u
      proof cases
        case beta
        with bodyfun xrel show ?thesis by blast
      next
        case (appL q')
        then obtain b' where q': q' = Lam A b' and r: red S b b'
          by (auto elim: red-LamE)
        have regb': regular S b'
          by (rule regular-red[OF r regb])
        have fixedb': erased S b'
          by (rule erased-red[OF r regb fixedb])
        have bodies': ⋀x. KC S n A x ⟹
            KC S n B (subst 0 x b')
        proof −
          fix x
          assume x: KC S n A x
          have old: KC S n B (subst 0 x b)
            by (rule bodyfun[OF x])
          have substep: red S (subst 0 x b) (subst 0 x b')
            by (rule red-subst0[
```

```
            OF r regb KC-regular[OF x] KC-erased[OF x]])
        show KC S n B (subst 0 x b') by (rule KC-red[OF old substep])
      qed
      have KC S n B (App (Lam A b') a)
        by (rule body-IH[OF r regb' fixedb' bodies' xrel])
      with q' appL show ?thesis by simp
    next
      case (appR a')
      have a': KC S n A a' by (rule KC-red[OF xrel appR(2)])
      have KC S n B (App (Lam A b) a')
        by (rule SNI.hyps(2)[OF appR(2) a'])
      with appR show ?thesis by simp
    qed
  qed
qed

lemma KC-Lam:
  assumes reg: regular S b and fixed: erased S b and sn: SN S b
    and bodies: ⋀m f x. Ren n m f ⟹ KC S m A x ⟹
      KC S m B (subst 0 x (vrename (up-ren f) b))
  shows KC S n (Fun A B) (Lam A b)
proof −
  have snlam: SN S (Lam A b) by (rule SN-LamI[OF sn])
  have fixedlam: erased S (Lam A b)
    using fixed unfolding erased-def by simp
  have app: ∀ m f x. Ren n m f ⟶ KC S m A x ⟶
      KC S m B (App (vrename f (Lam A b)) x)
  proof (intro allI impI)
    fix m f x
    assume ren: Ren n m f and x: KC S m A x
    have regbody: regular S (vrename (up-ren f) b) using reg by simp
    have fixedbody: erased S (vrename (up-ren f) b)
      by (rule erased-vrename[OF fixed])
    have snbody: SN S (vrename (up-ren f) b)
      by (rule SN-vrename[OF sn reg])
    have body: ⋀y. KC S m A y ⟹
        KC S m B (subst 0 y (vrename (up-ren f) b))
      by (rule bodies[OF ren])
    have KC S m B
        (App (Lam A (vrename (up-ren f) b)) x)
      by (rule KC-beta-app[OF snbody regbody fixedbody body x])
    then show KC S m B (App (vrename f (Lam A b)) x) by simp
  qed
  from reg fixedlam snlam app show ?thesis by simp
qed
```

15.7 Constructor and method arguments

fun *kc-args* ::
  sigenv ⇒ *nat* ⇒ *nat* ⇒ *argspec list* ⇒ *tm list* ⇒ *bool*
where
  kc-args S n h [] [] = *True*
| *kc-args S n h* [] (*a* # *as*) = *False*
| *kc-args S n h* (*s* # *ss*) [] = *False*
| *kc-args S n h* (*s* # *ss*) (*a* # *as*) =
    (*KC S n* (*argTy h s*) *a* ∧ *kc-args S n h ss as*)

lemma *kc-args-length*:
  kc-args S n h ss as ⟹ *length as* = *length ss*
  by (*induct ss arbitrary*: *as*) (*case-tac as*; *auto*)+

lemma *kc-args-nth*:
  assumes *kc-args S n h ss as* **and** *i* < *length as*
  shows *KC S n* (*argTy h* (*ss* ! *i*)) (*as* ! *i*)
  using *assms*
proof (*induct ss arbitrary*: *as i*)
  case *Nil*
  then show *?case* **by** (*cases as*) *auto*
next
  case (*Cons s ss*)
  then obtain *a as′* **where** *as*: *as* = *a* # *as′* **by** (*cases as*) *auto*
  show *?case*
  proof (*cases i*)
    case *0*
    with *Cons.prems as* **show** *?thesis* **by** *simp*
  next
    case (*Suc j*)
    have *tail*: *kc-args S n h ss as′* **using** *Cons.prems as* **by** *simp*
    have *bound*: *j* < *length as′* **using** *Cons.prems as Suc* **by** *simp*
    have *KC S n* (*argTy h* (*ss* ! *j*)) (*as′* ! *j*)
      by (*rule Cons.hyps*[*OF tail bound*])
    with *as Suc* **show** *?thesis* **by** *simp*
  qed
qed

lemma *kc-argsI*:
  assumes *length as* = *length ss*
    and ⋀*i*. *i* < *length as* ⟹
      KC S n (*argTy h* (*ss* ! *i*)) (*as* ! *i*)
  shows *kc-args S n h ss as*
  using *assms*
proof (*induct ss arbitrary*: *as*)
  case *Nil*
  then show *?case* **by** (*cases as*) *auto*
next
  case (*Cons s ss*)

```
  then obtain a as' where as: as = a # as' by (cases as) auto
  have head: KC S n (argTy h s) a
    using Cons.prems(2)[of 0] as by simp
  have tail: kc-args S n h ss as'
  proof (rule Cons.hyps)
    show length as' = length ss using Cons.prems(1) as by simp
    fix i
    assume i < length as'
    then have Suc i < length as using as by simp
    from Cons.prems(2)[OF this] as
    show KC S n (argTy h (ss ! i)) (as' ! i) by simp
  qed
  from head tail as show ?case by simp
qed

lemma kc-args-regular:
  assumes kc-args S n h ss as
  shows list-all (regular S) as
  using assms
  by (induct ss arbitrary: as)
    (case-tac as; auto dest: KC-regular)+

lemma kc-args-erased:
  assumes kc-args S n h ss as
  shows list-all (erased S) as
  using assms
  by (induct ss arbitrary: as)
    (case-tac as; auto dest: KC-erased)+

lemma kc-args-hargs:
  assumes kc-args S n h ss as
  shows hargs S h ss as
  using assms
proof (induct ss arbitrary: as)
  case Nil
  then show ?case by (cases as) auto
next
  case (Cons s ss)
  then obtain a as' where as: as = a # as' by (cases as) auto
  have rel: KC S n (argTy h s) a using Cons.prems as by simp
  have hcomp S (argHit h s) a
    using rel by (cases s) (auto dest: KC-Hit-hcomp)
  with Cons as show ?case by simp
qed

lemma KC-PCon:
  assumes k: k < length (pcons (S h))
    and args: kc-args S n h (pcons (S h) ! k) as
  shows KC S n (Hit h) (PCon h k as)
```

```
proof −
  have len: length as = length (pcons (S h) ! k)
    by (rule kc-args-length[OF args])
  have reg: regular S (PCon h k as)
    using k len kc-args-regular[OF args] by simp
  have fixed: erased S (PCon h k as)
    using kc-args-erased[OF args]
    unfolding erased-def
    by (simp add: list-all-iff map-eq-self-iff)
  have comp: hcomp S h (PCon h k as)
    by (rule hcomp-PCon[OF k kc-args-hargs[OF args]])
  from reg fixed comp show ?thesis by simp
qed

lemma KC-LCon:
  assumes k: k < length (lcons (S h))
    and args: kc-args S n h (lcons (S h) ! k) as
  shows KC S n (Hit h) (LCon h k as (DVar 1))
proof −
  have len: length as = length (lcons (S h) ! k)
    by (rule kc-args-length[OF args])
  have reg: regular S (LCon h k as (DVar 1))
    using k len kc-args-regular[OF args] by simp
  have fixed: erased S (LCon h k as (DVar 1))
    using kc-args-erased[OF args]
    unfolding erased-def
    by (simp add: list-all-iff map-eq-self-iff)
  have comp: hcomp S h (LCon h k as (DVar 1))
    by (rule hcomp-LCon[OF k kc-args-hargs[OF args]])
  from reg fixed comp show ?thesis by simp
qed

fun k-methods ::
  sigenv ⇒ nat ⇒ nat ⇒ ty ⇒
    argspec list list ⇒ tm list ⇒ bool
where
  k-methods S n h C [] [] = True
| k-methods S n h C [] (m # ms) = False
| k-methods S n h C (ss # css) [] = False
| k-methods S n h C (ss # css) (m # ms) =
    (KC S n (methodTy h C ss) m ∧ k-methods S n h C css ms)

lemma k-methods-length:
  k-methods S n h C css ms ⟹ length ms = length css
  by (induct css arbitrary: ms) (case-tac ms; auto)+

lemma k-methods-nth:
  assumes k-methods S n h C css ms and i < length ms
  shows KC S n (methodTy h C (css ! i)) (ms ! i)
```

```
  using assms
proof (induct css arbitrary: ms i)
  case Nil
  then show ?case by (cases ms) auto
next
  case (Cons ss css)
  then obtain m ms' where ms: ms = m # ms' by (cases ms) auto
  show ?case
  proof (cases i)
    case 0
    with Cons.prems ms show ?thesis by simp
  next
    case (Suc j)
    have tail: k-methods S n h C css ms' using Cons.prems ms by simp
    have bound: j < length ms' using Cons.prems ms Suc by simp
    have KC S n (methodTy h C (css ! j)) (ms' ! j)
      by (rule Cons.hyps[OF tail bound])
    with ms Suc show ?thesis by simp
  qed
qed

lemma k-methodsI:
  assumes length ms = length css
    and ⋀i. i < length ms ⟹
      KC S n (methodTy h C (css ! i)) (ms ! i)
  shows k-methods S n h C css ms
  using assms
proof (induct css arbitrary: ms)
  case Nil
  then show ?case by (cases ms) auto
next
  case (Cons ss css)
  then obtain m ms' where ms: ms = m # ms' by (cases ms) auto
  have head: KC S n (methodTy h C ss) m
    using Cons.prems(2)[of 0] ms by simp
  have tail: k-methods S n h C css ms'
  proof (rule Cons.hyps)
    show length ms' = length css using Cons.prems(1) ms by simp
    fix i
    assume i < length ms'
    then have Suc i < length ms using ms by simp
    from Cons.prems(2)[OF this] ms
    show KC S n (methodTy h C (css ! i)) (ms' ! i) by simp
  qed
  from head tail ms show ?case by simp
qed

lemma k-methods-regular:
  k-methods S n h C css ms ⟹ list-all (regular S) ms
```

```
  by (induct css arbitrary: ms)
    (case-tac ms; auto dest: KC-regular)+

lemma k-methods-erased:
  k-methods S n h C css ms ⟹ list-all (erased S) ms
  by (induct css arbitrary: ms)
    (case-tac ms; auto dest: KC-erased)+

lemma k-methods-SN:
  k-methods S n h C css ms ⟹ list-all (SN S) ms
  by (induct css arbitrary: ms)
    (case-tac ms; auto dest: KC-SN)+

lemma k-methods-list-red:
  assumes list-red S ms ms' and k-methods S n h C css ms
  shows k-methods S n h C css ms'
  using assms
proof (induct arbitrary: css rule: list-red.induct)
  case (lr-head m m' ms)
  then show ?case by (cases css) (auto intro: KC-red)
next
  case (lr-tail ms ms' m)
  then show ?case by (cases css) auto
qed

lemma KC-apps-elimArgs:
  assumes args: kc-args S n h ss as
    and method: KC S n (methodTy h C ss) m
    and recursive: ⋀a. KC S n (Hit h) a ⟹
      KC S n C (HElim h C pms lms a)
  shows KC S n C (apps m (elimArgs h C pms lms ss as))
  using args method
proof (induct ss arbitrary: as m)
  case Nil
  then have as = [] by (cases as) auto
  with Nil show ?case by simp
next
  case (Cons s ss)
  then obtain a as' where as: as = a # as' by (cases as) auto
  have tail: kc-args S n h ss as' using Cons.prems(1) as by simp
  show ?case
  proof (cases s)
    case (Par h')
    have a: KC S n (Hit h') a using Cons.prems(1) as Par by simp
    have m: KC S n (Fun (Hit h') (methodTy h C ss)) m
      using Cons.prems(2) Par by simp
    have KC S n (methodTy h C ss) (App m a)
      by (rule KC-App[OF m a])
    from Cons.hyps[OF tail this] as Par show ?thesis by simp
```

```
  next
    case Rec
    have a: KC S n (Hit h) a using Cons.prems(1) as Rec by simp
    have m: KC S n (Fun (Hit h) (Fun C (methodTy h C ss))) m
      using Cons.prems(2) Rec by simp
    have first: KC S n (Fun C (methodTy h C ss)) (App m a)
      by (rule KC-App[OF m a])
    have ea: KC S n C (HElim h C pms lms a)
      by (rule recursive[OF a])
    have KC S n (methodTy h C ss) (App (App m a) (HElim h C pms lms a))
      by (rule KC-App[OF first ea])
    from Cons.hyps[OF tail this] as Rec show ?thesis by simp
  qed
qed

lemma KC-apps-elimArgs-indexed:
  assumes args: kc-args S n h ss as
    and method: KC S n (methodTy h C ss) m
    and recursive: ⋀i. i < length as ⟹ ss ! i = Rec ⟹
      KC S n C (HElim h C pms lms (as ! i))
  shows KC S n C (apps m (elimArgs h C pms lms ss as))
  using args method recursive
proof (induct ss arbitrary: as m)
  case Nil
  then have as = [] by (cases as) auto
  with Nil show ?case by simp
next
  case (Cons s ss)
  then obtain a as' where as: as = a # as' by (cases as) auto
  have tail: kc-args S n h ss as' using Cons.prems(1) as by simp
  have rec-tail: ⋀i. i < length as' ⟹ ss ! i = Rec ⟹
     KC S n C (HElim h C pms lms (as' ! i))
  proof −
    fix i
    assume i: i < length as' and rec: ss ! i = Rec
    have Suc i < length as using i as by simp
    moreover have (s # ss) ! Suc i = Rec using rec by simp
    ultimately have KC S n C
       (HElim h C pms lms (as ! Suc i))
      by (rule Cons.prems(3))
    with as show KC S n C (HElim h C pms lms (as' ! i)) by simp
  qed
  show ?case
  proof (cases s)
    case (Par h')
    have a: KC S n (Hit h') a using Cons.prems(1) as Par by simp
    have m: KC S n (Fun (Hit h') (methodTy h C ss)) m
      using Cons.prems(2) Par by simp
    have KC S n (methodTy h C ss) (App m a)
```

```
    by (rule KC-App[OF m a])
  from Cons.hyps[OF tail this rec-tail] as Par show ?thesis by simp
 next
  case Rec
  have a: KC S n (Hit h) a using Cons.prems(1) as Rec by simp
  have m: KC S n (Fun (Hit h) (Fun C (methodTy h C ss))) m
   using Cons.prems(2) Rec by simp
  have ea: KC S n C (HElim h C pms lms a)
  proof −
   have 0 < length as using as by simp
   moreover have (s # ss) ! 0 = Rec using Rec by simp
   ultimately have KC S n C (HElim h C pms lms (as ! 0))
    by (rule Cons.prems(3))
   with as show ?thesis by simp
  qed
  have first: KC S n (Fun C (methodTy h C ss)) (App m a)
   by (rule KC-App[OF m a])
  have KC S n (methodTy h C ss) (App (App m a) (HElim h C pms lms a))
   by (rule KC-App[OF first ea])
  from Cons.hyps[OF tail this rec-tail] as Rec show ?thesis by simp
 qed
qed

lemma regular-PCon-arg:
 regular S (PCon h k as) ⟹ i < length as ⟹
  regular S (as ! i)
 by (auto simp: list-all-iff)

lemma regular-LCon-arg:
 regular S (LCon h k as d) ⟹ i < length as ⟹
  regular S (as ! i)
 by (auto simp: list-all-iff)

lemma erased-PCon-arg:
 assumes erased S (PCon h k as) and i < length as
 shows erased S (as ! i)
 using assms unfolding erased-def
 by (auto simp: map-eq-self-iff)

lemma erased-LCon-arg:
 assumes erased S (LCon h k as d) and i < length as
 shows erased S (as ! i)
 using assms unfolding erased-def
 by (auto simp: map-eq-self-iff)

lemma regular-HElim:
 assumes regular S t
  and k-methods S n h C (pcons (S h)) pms
  and k-methods S n h C (lcons (S h)) lms
```

```
  shows regular S (HElim h C pms lms t)
  using assms k-methods-length k-methods-regular by simp

lemma erased-HElim:
  assumes erased S t
    and k-methods S n h C (pcons (S h)) pms
    and k-methods S n h C (lcons (S h)) lms
  shows erased S (HElim h C pms lms t)
proof −
  have p: map (erase-dims S) pms = pms
    using k-methods-erased[OF assms(2)]
    unfolding erased-def
    by (simp add: list-all-iff map-eq-self-iff)
  have l: map (erase-dims S) lms = lms
    using k-methods-erased[OF assms(3)]
    unfolding erased-def
    by (simp add: list-all-iff map-eq-self-iff)
  from assms(1) p l show ?thesis unfolding erased-def by simp
qed

lemma KC-HElim-method-access:
  assumes pms: k-methods S n h C (pcons (S h)) pms
    and lms: k-methods S n h C (lcons (S h)) lms
    and close: ⋀pms lms.
      k-methods S n h C (pcons (S h)) pms ⟹
      k-methods S n h C (lcons (S h)) lms ⟹
      (⋀pms'. list-red S pms pms' ⟹
        KC S n C (HElim h C pms' lms t)) ⟹
      (⋀lms'. list-red S lms lms' ⟹
        KC S n C (HElim h C pms lms' t)) ⟹
      KC S n C (HElim h C pms lms t)
  shows KC S n C (HElim h C pms lms t)
proof −
  have snp: SN-list S pms
    by (rule SN-list-all[OF k-methods-SN[OF pms]])
  have snl: SN-list S lms
    by (rule SN-list-all[OF k-methods-SN[OF lms]])
  from snp snl pms lms show ?thesis
  proof (induct rule: SN-lists-induct)
    case (step pms lms)
    show ?case
    proof (rule close[OF step.prems(1,2)])
      fix pms'
      assume lr: list-red S pms pms'
      have kp: k-methods S n h C (pcons (S h)) pms'
        by (rule k-methods-list-red[OF lr step.prems(1)])
      show KC S n C (HElim h C pms' lms t)
        by (rule step.hyps(1)[OF lr kp step.prems(2)])
    next
```

```
    fix lms'
    assume lr: list-red S lms lms'
    have kl: k-methods S n h C (lcons (S h)) lms'
      by (rule k-methods-list-red[OF lr step.prems(2)])
    show KC S n C (HElim h C pms lms' t)
      by (rule step.hyps(2)[OF lr step.prems(1) kl])
  qed
 qed
qed

lemma red-HElim-PConE [case-names head pm lm scrut]:
 assumes step: red S (HElim h C pms lms (PCon h k as)) u
 obtains
  (head) k < length (pcons (S h))
    u = apps (pms ! k)
      (elimArgs h C pms lms (pcons (S h) ! k) as)
 | (pm) xs m m' ys where
    pms = xs @ m # ys red S m m'
    u = HElim h C (xs @ m' # ys) lms (PCon h k as)
 | (lm) xs m m' ys where
    lms = xs @ m # ys red S m m'
    u = HElim h C pms (xs @ m' # ys) (PCon h k as)
 | (scrut) t' where red S (PCon h k as) t'
    u = HElim h C pms lms t'
 using step
 by (cases rule: red.cases) (auto intro: head pm lm scrut)

lemma red-HElim-LConE [case-names head pm lm scrut]:
 assumes step: red S (HElim h C pms lms (LCon h k as (DVar j))) u
 obtains
  (head) k < length (lcons (S h))
    u = apps (dsubst S 0 (DVar j) (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as)
 | (pm) xs m m' ys where
    pms = xs @ m # ys red S m m'
    u = HElim h C (xs @ m' # ys) lms (LCon h k as (DVar j))
 | (lm) xs m m' ys where
    lms = xs @ m # ys red S m m'
    u = HElim h C pms (xs @ m' # ys) (LCon h k as (DVar j))
 | (scrut) t' where red S (LCon h k as (DVar j)) t'
    u = HElim h C pms lms t'
 using step
 by (cases rule: red.cases) (auto intro: head pm lm scrut)

lemma KC-HElim-aux:
   assumes comp: hcomp S h t and reg: regular S t and fixed: erased S t
     and pms: k-methods S n h C (pcons (S h)) pms
     and lms: k-methods S n h C (lcons (S h)) lms
   shows KC S n C (HElim h C pms lms t)
```

```
  using comp reg fixed pms lms
proof (induct arbitrary: n C pms lms rule: hcomp.induct)
  case (hneutral t h)
  show ?case
  proof (rule KC-HElim-method-access[
        OF hneutral.prems(3,4)])
    fix pms lms
    assume kp: k-methods S n h C (pcons (S h)) pms
      and kl: k-methods S n h C (lcons (S h)) lms
      and pstep: ⋀pms'. list-red S pms pms' ⟹
        KC S n C (HElim h C pms' lms t)
      and lstep: ⋀lms'. list-red S lms lms' ⟹
        KC S n C (HElim h C pms lms' t)
    have rh: regular S (HElim h C pms lms t)
      by (rule regular-HElim[OF hneutral.prems(1) kp kl])
    have eh: erased S (HElim h C pms lms t)
      by (rule erased-HElim[OF hneutral.prems(2) kp kl])
    show KC S n C (HElim h C pms lms t)
    proof (rule KC-neutral[OF - rh eh])
      show neutral (HElim h C pms lms t) by simp
      fix u
      assume red: red S (HElim h C pms lms t) u
      then show KC S n C u
      proof cases
        case elimPoint
        with hneutral.hyps(1) show ?thesis by simp
      next
        case elimPath
        with hneutral.hyps(1) show ?thesis by simp
      next
        case elimPm
        from pstep elimPm list-red-append show ?thesis by metis
      next
        case elimLm
        from lstep elimLm list-red-append show ?thesis by metis
      next
        case (elimScrutinee t')
        have rt: regular S t'
          by (rule regular-red[OF elimScrutinee(2) hneutral.prems(1)])
        have et: erased S t'
          by (rule erased-red[
                OF elimScrutinee(2) hneutral.prems(1,2)])
        have KC S n C (HElim h C pms lms t')
          by (rule hneutral.hyps(3)[
                OF elimScrutinee(2) rt et kp kl])
        with elimScrutinee show ?thesis by simp
      qed
    qed
  qed
```

```
next
  case (hpcon k h as)
  have args: kc-args S n h (pcons (S h) ! k) as
  proof (rule kc-argsI[OF hpcon.hyps(2)])
    fix i
    assume i: i < length as
    have ri: regular S (as ! i)
      by (rule regular-PCon-arg[OF hpcon.prems(1) i])
    have ei: erased S (as ! i)
      by (rule erased-PCon-arg[OF hpcon.prems(2) i])
    have hi: hcomp S (argHit h (pcons (S h) ! k ! i)) (as ! i)
      by (rule hpcon.hyps(3)[OF i])
    from ri ei hi show
      KC S n (argTy h (pcons (S h) ! k ! i)) (as ! i)
      by (cases pcons (S h) ! k ! i) simp-all
  qed
  show ?case
  proof (rule KC-HElim-method-access[
        OF hpcon.prems(3,4)])
    fix pms lms
    assume kp: k-methods S n h C (pcons (S h)) pms
      and kl: k-methods S n h C (lcons (S h)) lms
      and pstep: ⋀pms'. list-red S pms pms' ⟹
        KC S n C (HElim h C pms' lms (PCon h k as))
      and lstep: ⋀lms'. list-red S lms lms' ⟹
        KC S n C (HElim h C pms lms' (PCon h k as))
    have rh: regular S (HElim h C pms lms (PCon h k as))
      by (rule regular-HElim[OF hpcon.prems(1) kp kl])
    have eh: erased S (HElim h C pms lms (PCon h k as))
      by (rule erased-HElim[OF hpcon.prems(2) kp kl])
    show KC S n C (HElim h C pms lms (PCon h k as))
    proof (rule KC-neutral[OF - rh eh])
      show neutral (HElim h C pms lms (PCon h k as)) by simp
      fix u
      assume red: red S (HElim h C pms lms (PCon h k as)) u
      show KC S n C u
      proof (rule red-HElim-PConE[OF red])
        assume head-guard: k < length (pcons (S h))
          and head-eq: u = apps (pms ! k)
            (elimArgs h C pms lms (pcons (S h) ! k) as)
        have kpm: k < length pms
          using hpcon.hyps(1) k-methods-length[OF kp] by simp
        have method: KC S n (methodTy h C (pcons (S h) ! k)) (pms ! k)
          by (rule k-methods-nth[OF kp kpm])
        have recursive: ⋀i. i < length as ⟹
            pcons (S h) ! k ! i = Rec ⟹
            KC S n C (HElim h C pms lms (as ! i))
        proof −
          fix i
```

```
    assume i: i < length as
      and rec: pcons (S h) ! k ! i = Rec
    have ri: regular S (as ! i)
      by (rule regular-PCon-arg[OF hpcon.prems(1) i])
    have ei: erased S (as ! i)
      by (rule erased-PCon-arg[OF hpcon.prems(2) i])
    from hpcon.hyps(4)[OF i, of n C pms lms] ri ei kp kl rec
    show KC S n C (HElim h C pms lms (as ! i)) by simp
  qed
  have head: KC S n C
      (apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k) as))
    by (rule KC-apps-elimArgs-indexed[
          OF args method recursive])
  with head-eq show ?thesis by simp
next
  fix xs m m' ys
  assume pms-eq: pms = xs @ m # ys and mred: red S m m'
    and u-eq: u =
      HElim h C (xs @ m' # ys) lms (PCon h k as)
  have lr: list-red S pms (xs @ m' # ys)
    using pms-eq mred by (auto intro: list-red-append)
  have KC S n C
      (HElim h C (xs @ m' # ys) lms (PCon h k as))
    by (rule pstep[OF lr])
  with u-eq show ?thesis by simp
next
  fix xs m m' ys
  assume lms-eq: lms = xs @ m # ys and mred: red S m m'
    and u-eq: u =
      HElim h C pms (xs @ m' # ys) (PCon h k as)
  have lr: list-red S lms (xs @ m' # ys)
    using lms-eq mred by (auto intro: list-red-append)
  have KC S n C
      (HElim h C pms (xs @ m' # ys) (PCon h k as))
    by (rule lstep[OF lr])
  with u-eq show ?thesis by simp
next
  fix t'
  assume tred: red S (PCon h k as) t'
    and u-eq: u = HElim h C pms lms t'
  have rt: regular S t'
    by (rule regular-red[OF tred hpcon.prems(1)])
  have et: erased S t'
    by (rule erased-red[
          OF tred hpcon.prems(1,2)])
  have KC S n C (HElim h C pms lms t')
    by (rule hpcon.hyps(6)[
          OF tred rt et kp kl])
```

```
      with u-eq show ?thesis by simp
    qed
  qed
qed
next
  case (hlcon k h as j)
  have args: kc-args S n h (lcons (S h) ! k) as
  proof (rule kc-argsI[OF hlcon.hyps(2)])
    fix i
    assume i: i < length as
    have ri: regular S (as ! i)
      by (rule regular-LCon-arg[OF hlcon.prems(1) i])
    have ei: erased S (as ! i)
      by (rule erased-LCon-arg[OF hlcon.prems(2) i])
    have hi: hcomp S (argHit h (lcons (S h) ! k ! i)) (as ! i)
      by (rule hlcon.hyps(3)[OF i])
    from ri ei hi show
      KC S n (argTy h (lcons (S h) ! k ! i)) (as ! i)
      by (cases lcons (S h) ! k ! i) simp-all
  qed
  show ?case
  proof (rule KC-HElim-method-access[
      OF hlcon.prems(3,4)])
    fix pms lms
    assume kp: k-methods S n h C (pcons (S h)) pms
      and kl: k-methods S n h C (lcons (S h)) lms
      and pstep: ⋀pms'. list-red S pms pms' ⟹
        KC S n C (HElim h C pms' lms (LCon h k as (DVar j)))
      and lstep: ⋀lms'. list-red S lms lms' ⟹
        KC S n C (HElim h C pms lms' (LCon h k as (DVar j)))
    have rh: regular S (HElim h C pms lms (LCon h k as (DVar j)))
      by (rule regular-HElim[OF hlcon.prems(1) kp kl])
    have eh: erased S (HElim h C pms lms (LCon h k as (DVar j)))
      by (rule erased-HElim[OF hlcon.prems(2) kp kl])
    show KC S n C (HElim h C pms lms (LCon h k as (DVar j)))
    proof (rule KC-neutral[OF - rh eh])
      show neutral (HElim h C pms lms (LCon h k as (DVar j))) by simp
      fix u
      assume red: red S (HElim h C pms lms (LCon h k as (DVar j))) u
      show KC S n C u
      proof (rule red-HElim-LConE[OF red])
        assume head-guard: k < length (lcons (S h))
          and head-eq: u =
            apps (dsubst S 0 (DVar j) (lms ! k))
              (elimArgs h C pms lms (lcons (S h) ! k) as)
        have klm: k < length lms
          using hlcon.hyps(1) k-methods-length[OF kl] by simp
        have method: KC S n (methodTy h C (lcons (S h) ! k)) (lms ! k)
          by (rule k-methods-nth[OF kl klm])
```

```
  have same: dsubst S 0 (DVar j) (lms ! k) = lms ! k
    by (rule erased-dsubst-zero[OF KC-erased[OF method]])
  have recursive: ⋀i. i < length as ⟹
      lcons (S h) ! k ! i = Rec ⟹
      KC S n C (HElim h C pms lms (as ! i))
  proof −
    fix i
    assume i: i < length as
      and rec: lcons (S h) ! k ! i = Rec
    have ri: regular S (as ! i)
      by (rule regular-LCon-arg[OF hlcon.prems(1) i])
    have ei: erased S (as ! i)
      by (rule erased-LCon-arg[OF hlcon.prems(2) i])
    from hlcon.hyps(4)[OF i, of n C pms lms] ri ei kp kl rec
    show KC S n C (HElim h C pms lms (as ! i)) by simp
  qed
  have head: KC S n C
      (apps (lms ! k)
        (elimArgs h C pms lms (lcons (S h) ! k) as))
    by (rule KC-apps-elimArgs-indexed[
          OF args method recursive])
  with head-eq same show ?thesis by simp
next
  fix xs m m′ ys
  assume pms-eq: pms = xs @ m # ys and mred: red S m m′
    and u-eq: u =
      HElim h C (xs @ m′ # ys) lms (LCon h k as (DVar j))
  have lr: list-red S pms (xs @ m′ # ys)
    using pms-eq mred by (auto intro: list-red-append)
  have KC S n C
      (HElim h C (xs @ m′ # ys) lms (LCon h k as (DVar j)))
    by (rule pstep[OF lr])
  with u-eq show ?thesis by simp
next
  fix xs m m′ ys
  assume lms-eq: lms = xs @ m # ys and mred: red S m m′
    and u-eq: u =
      HElim h C pms (xs @ m′ # ys) (LCon h k as (DVar j))
  have lr: list-red S lms (xs @ m′ # ys)
    using lms-eq mred by (auto intro: list-red-append)
  have KC S n C
      (HElim h C pms (xs @ m′ # ys) (LCon h k as (DVar j)))
    by (rule lstep[OF lr])
  with u-eq show ?thesis by simp
next
  fix t′
  assume tred: red S (LCon h k as (DVar j)) t′
    and u-eq: u = HElim h C pms lms t′
  have rt: regular S t′
```

```
      by (rule regular-red[OF tred hlcon.prems(1)])
    have et: erased S t'
      by (rule erased-red[
          OF tred hlcon.prems(1,2)])
    have KC S n C (HElim h C pms lms t')
      by (rule hlcon.hyps(6)[
          OF tred rt et kp kl])
    with u-eq show ?thesis by simp
   qed
  qed
 qed
 qed

lemma KC-HElim:
 assumes KC S n (Hit h) t
  and k-methods S n h C (pcons (S h)) pms
  and k-methods S n h C (lcons (S h)) lms
 shows KC S n C (HElim h C pms lms t)
 by (rule KC-HElim-aux[
     OF KC-Hit-hcomp[OF assms(1)] KC-regular[OF assms(1)]
      KC-erased[OF assms(1)] assms(2,3)])
```

15.8 The fundamental theorem

```
definition k-env ::
 sigenv ⇒ nat ⇒ ty list ⇒ (nat ⇒ tm) ⇒ bool
where
 k-env S n G s ⟷
  (∀ i < length G. KC S n (G ! i) (s i))

lemma k-env-up:
 assumes k-env S n G s
 shows k-env S (Suc n) (A # G) (up-sub s)
 unfolding k-env-def
proof (intro allI impI)
 fix i
 assume i: i < length (A # G)
 show KC S (Suc n) ((A # G) ! i) (up-sub s i)
 proof (cases i)
  case 0
  then show ?thesis by (simp add: KC-Var)
 next
  case (Suc j)
  from assms i Suc have rel: KC S n (G ! j) (s j)
   unfolding k-env-def by simp
  have KC S (Suc n) (G ! j) (vrename Suc (s j))
   by (rule KC-vrename[OF rel Ren-Suc])
  with Suc show ?thesis by (simp add: lift-zero-as-vrename)
 qed
```

qed

lemma *k-env-vrename*:
assumes *env*: *k-env S n G s* **and** *ren*: *Ren n m f*
shows *k-env S m G* (*λi. vrename f* (*s i*))
using *env* **unfolding** *k-env-def*
by (*auto intro*: *KC-vrename*[*OF - ren*])

lemma *k-env-ext*:
assumes *env*: *k-env S n G s* **and** *x*: *KC S n A x*
shows *k-env S n* (*A # G*) (*ext-sub x s*)
using *env x* **unfolding** *k-env-def ext-sub-def*
by (*intro allI impI*) (*case-tac i*; *auto*)

theorem *fundamental-erased*:
assumes *deriv*: *wt S D G t A* **and** *env*: *k-env S n G s*
shows *KC S n A* (*psubst s* (*erase-dims S t*))
using *deriv env*
proof (*induct arbitrary*: *n s rule*: *wt.induct*)
case (*wtVar i G D*)
then show *?case* **unfolding** *k-env-def* **by** *simp*
next
case (*wtLam D A G t B*)
have *up*: *k-env S* (*Suc n*) (*A # G*) (*up-sub s*)
by (*rule k-env-up*[*OF wtLam.prems*])
have *body*: *KC S* (*Suc n*) *B*
(*psubst* (*up-sub s*) (*erase-dims S t*))
by (*rule wtLam.hyps*(*2*)[*OF up*])
let *?b* = *psubst* (*up-sub s*) (*erase-dims S t*)
have *reg*: *regular S ?b* **by** (*rule KC-regular*[*OF body*])
have *fixed*: *erased S ?b* **by** (*rule KC-erased*[*OF body*])
have *sn*: *SN S ?b* **by** (*rule KC-SN*[*OF body*])
have *bodies*: ⋀*m f x*. *Ren n m f* ⟹ *KC S m A x* ⟹
KC S m B (*subst 0 x* (*vrename* (*up-ren f*) *?b*))
proof −
fix *m f x*
assume *ren*: *Ren n m f* **and** *x*: *KC S m A x*
have *renamed*: *k-env S m G* (*λi. vrename f* (*s i*))
by (*rule k-env-vrename*[*OF wtLam.prems ren*])
have *extended*:
k-env S m (*A # G*)
(*ext-sub x* (*λi. vrename f* (*s i*)))
by (*rule k-env-ext*[*OF renamed x*])
have *ih*: *KC S m B*
(*psubst* (*ext-sub x* (*λi. vrename f* (*s i*)))
(*erase-dims S t*))
by (*rule wtLam.hyps*(*2*)[*OF extended*])
have *eq*: *subst 0 x* (*vrename* (*up-ren f*) *?b*) =
psubst (*ext-sub x* (*λi. vrename f* (*s i*)))

```
      (erase-dims S t)
    proof −
      have renamed-body:
        vrename (up-ren f) ?b =
          psubst (up-sub (λi. vrename f (s i)))
            (erase-dims S t)
        by (simp add: vrename-psubst up-sub-vrename)
      show ?thesis
        by (simp add: renamed-body subst0-psubst-up)
    qed
    from ih eq show
      KC S m B (subst 0 x (vrename (up-ren f) ?b))
      by simp
  qed
  have KC S n (Fun A B) (Lam A ?b)
    by (rule KC-Lam[OF reg fixed sn bodies])
  then show ?case by simp
next
  case (wtApp D G t A B u)
  have fnrel: KC S n (Fun A B) (psubst s (erase-dims S t))
    by (rule wtApp.hyps(2)[OF wtApp.prems])
  have arg: KC S n A (psubst s (erase-dims S u))
    by (rule wtApp.hyps(4)[OF wtApp.prems])
  from KC-App[OF fnrel arg] show ?case by simp
next
  case (wtPCon k h as D G)
  have args: kc-args S n h (pcons (S h) ! k)
      (map (psubst s ∘ erase-dims S) as)
  proof (rule kc-argsI)
    show length (map (psubst s ∘ erase-dims S) as) =
        length (pcons (S h) ! k)
      using wtPCon.hyps(2) by simp
    fix i
    assume i: i < length (map (psubst s ∘ erase-dims S) as)
    have ia: i < length as using i by simp
    have ih: ∀ x env.
        k-env S x G env ⟶
        KC S x (argTy h (pcons (S h) ! k ! i))
          (psubst env (erase-dims S (as ! i)))
      using wtPCon.hyps(3)[rule-format, OF ia] by blast
    have KC S n (argTy h (pcons (S h) ! k ! i))
        (psubst s (erase-dims S (as ! i)))
      using ih wtPCon.prems by blast
    with i show KC S n (argTy h (pcons (S h) ! k ! i))
        ((map (psubst s ∘ erase-dims S) as) ! i)
      by simp
  qed
  have KC S n (Hit h)
      (PCon h k (map (psubst s ∘ erase-dims S) as))
```

```
    by (rule KC-PCon[OF wtPCon.hyps(1) args])
  then show ?case by simp
next
  case (wtLCon j D k h as G)
  have args: kc-args S n h (lcons (S h) ! k)
      (map (psubst s ∘ erase-dims S) as)
  proof (rule kc-argsI)
    show length (map (psubst s ∘ erase-dims S) as) =
        length (lcons (S h) ! k)
      using wtLCon.hyps(3) by simp
    fix i
    assume i: i < length (map (psubst s ∘ erase-dims S) as)
    have ia: i < length as using i by simp
    have ih: ∀ x env.
        k-env S x G env ⟶
        KC S x (argTy h (lcons (S h) ! k ! i))
          (psubst env (erase-dims S (as ! i)))
      using wtLCon.hyps(4)[rule-format, OF ia] by blast
    have KC S n (argTy h (lcons (S h) ! k ! i))
        (psubst s (erase-dims S (as ! i)))
      using ih wtLCon.prems by blast
    with i show KC S n (argTy h (lcons (S h) ! k ! i))
        ((map (psubst s ∘ erase-dims S) as) ! i)
      by simp
  qed
  have KC S n (Hit h)
      (LCon h k (map (psubst s ∘ erase-dims S) as) (DVar 1))
    by (rule KC-LCon[OF wtLCon.hyps(2) args])
  then show ?case by simp
next
  case (wtElim D G t h pms lms C)
  have scrut: KC S n (Hit h) (psubst s (erase-dims S t))
    by (rule wtElim.hyps(2)[OF wtElim.prems])
  have pmethods: k-methods S n h C (pcons (S h))
      (map (psubst s ∘ erase-dims S) pms)
  proof (rule k-methodsI)
    show length (map (psubst s ∘ erase-dims S) pms) =
        length (pcons (S h))
      using wtElim.hyps(3) by simp
    fix i
    assume i: i < length (map (psubst s ∘ erase-dims S) pms)
    have ip: i < length pms using i by simp
    have ih: ∀ x env.
        k-env S x G env ⟶
        KC S x (methodTy h C (pcons (S h) ! i))
          (psubst env (erase-dims S (pms ! i)))
      using wtElim.hyps(5)[rule-format, OF ip] by blast
    have KC S n (methodTy h C (pcons (S h) ! i))
        (psubst s (erase-dims S (pms ! i)))
```

```
    using ih wtElim.prems by blast
  with i show KC S n (methodTy h C (pcons (S h) ! i))
      ((map (psubst s ∘ erase-dims S) pms) ! i)
    by simp
 qed
 have lmethods: k-methods S n h C (lcons (S h))
    (map (psubst s ∘ erase-dims S) lms)
 proof (rule k-methodsI)
  show length (map (psubst s ∘ erase-dims S) lms) =
      length (lcons (S h))
    using wtElim.hyps(4) by simp
  fix i
  assume i: i < length (map (psubst s ∘ erase-dims S) lms)
  have il: i < length lms using i by simp
  have ih: ∀ x env.
      k-env S x G env ⟶
      KC S x (methodTy h C (lcons (S h) ! i))
        (psubst env (erase-dims S (lms ! i)))
    using wtElim.hyps(6)[rule-format, OF il] by blast
  have KC S n (methodTy h C (lcons (S h) ! i))
      (psubst s (erase-dims S (lms ! i)))
    using ih wtElim.prems by blast
  with i show KC S n (methodTy h C (lcons (S h) ! i))
      ((map (psubst s ∘ erase-dims S) lms) ! i)
    by simp
 qed
 have KC S n C
    (HElim h C (map (psubst s ∘ erase-dims S) pms)
      (map (psubst s ∘ erase-dims S) lms)
      (psubst s (erase-dims S t)))
  by (rule KC-HElim[OF scrut pmethods lmethods])
 then show ?case by simp
qed

lemma k-env-Var: k-env S (length G) G Var
 unfolding k-env-def by (auto intro: KC-Var)

lemma SN-erase-dims-reflect-regular-aux:
 assumes sn: SN S e
 shows ⋀t. e = erase-dims S t ⟹ regular S t ⟹ SN S t
 using sn
proof (induct rule: SN.induct)
 case (SNI e)
 show ?case
 proof (rule SN.SNI)
  fix u
  assume step: red S t u
  have erased-step: red S e (erase-dims S u)
    using red-erase-dims[OF step SNI.prems(2)] SNI.prems(1) by simp
```

```
    have regu: regular S u
      by (rule regular-red[OF step SNI.prems(2)])
    show SN S u
      by (rule SNI.hyps(2)[OF erased-step refl regu])
  qed
qed

lemma SN-erase-dims-reflect-regular:
  assumes regular S t and SN S (erase-dims S t)
  shows SN S t
  by (rule SN-erase-dims-reflect-regular-aux[OF assms(2) refl assms(1)])

theorem wt-strong-normalization:
  assumes typed: wt S D G t A
  shows SN S t
proof −
  have rel: KC S (length G) A (erase-dims S t)
    using fundamental-erased[OF typed k-env-Var] by simp
  have erased-sn: SN S (erase-dims S t) by (rule KC-SN[OF rel])
  show ?thesis
    by (rule SN-erase-dims-reflect-regular[
          OF regular-wt[OF typed] erased-sn])
qed

theorem cwt-strong-normalization:
  assumes wf-sigenv S and cwt S D G t A
  shows SN S t
  by (rule wt-strong-normalization[OF cwt-wt[OF assms(2)]])

corollary strong-normalization:
  assumes wf-sigenv S and cwt S D G t A
  shows SN S t
  by (rule cwt-strong-normalization[OF assms])

theorem SN-reaches-normal-form:
  assumes SN S t
  shows ∃ u. reds S t u ∧ nform S u
  using assms
proof (induct rule: SN.induct)
  case (SNI t)
  show ?case
  proof (cases nform S t)
    case True
    then show ?thesis by blast
  next
    case False
    then obtain u where step: red S t u
      unfolding nform-def by blast
    from SNI.hyps(2)[OF step] obtain v where
```

uv: *reds S u v* **and** *normal*: *nform S v*
by *blast*
have *tu*: *reds S t u* **using** *step* **by** (*rule r-into-rtranclp*)
have *reds S t v* **by** (*rule rtranclp-trans*[*OF tu uv*])
with *normal* **show** *?thesis* **by** *blast*
qed
qed

corollary *SN-normalizes*:
assumes *SN S t*
shows *normalizes S t*
using *SN-reaches-normal-form*[*OF assms*]
unfolding *normalizes-def* **by** *blast*

corollary *wt-reaches-normal-form*:
assumes *wt S D G t A*
shows ∃ *u*. *reds S t u* ∧ *nform S u*
by (*rule SN-reaches-normal-form*[*OF wt-strong-normalization*[*OF assms*]])

corollary *cwt-reaches-normal-form*:
assumes *wf-sigenv S* **and** *cwt S D G t A*
shows ∃ *u*. *reds S t u* ∧ *nform S u*
by (*rule SN-reaches-normal-form*[
OF cwt-strong-normalization[*OF assms*]])

corollary *cwt-normalizes*:
assumes *wf-sigenv S* **and** *cwt S D G t A*
shows *normalizes S t*
by (*rule SN-normalizes*[*OF cwt-strong-normalization*[*OF assms*]])

16 Unconditional canonical results and consistency

theorem *checked-term-has-canonical-normal-form*:
assumes *wf*: *wf-sigenv S* **and** *typed*: *cwt S* {} [] *t A*
shows ∃ *u*. *reds S t u* ∧ *nform S u* ∧ *canonAt S A u*
proof −
from *cwt-reaches-normal-form*[*OF wf typed*]
obtain *u* **where** *reduces*: *reds S t u* **and** *normal*: *nform S u*
by *blast*
have *target*: *cwt S* {} [] *u A*
by (*rule checked-source-subject-reduction-steps*[*OF wf typed reduces*])
have *canonAt S A u* **by** (*rule canonical-forms*[*OF target normal*])
with *reduces normal* **show** *?thesis* **by** *blast*
qed

corollary *checked-hit-has-canonical-normal-form*:
assumes *wf*: *wf-sigenv S* **and** *typed*: *cwt S* {} [] *t* (*Hit h*)
shows ∃ *u*. *reds S t u* ∧ *nform S u* ∧ *canonical S h u*
using *checked-term-has-canonical-normal-form*[*OF wf typed*] **by** *auto*

corollary *empty-signature-consistency*:
assumes *wf*: *wf-sigenv S* **and** *empty*: *pcons* (*S h*) = []
shows ¬ (∃ *t*. *cwt S* {} [] *t* (*Hit h*))
proof
assume ∃ *t*. *cwt S* {} [] *t* (*Hit h*)
then obtain *t* **where** *typed*: *cwt S* {} [] *t* (*Hit h*) **by** *blast*
from *checked-hit-has-canonical-normal-form*[*OF wf typed*]
obtain *u* **where** *canonical S h u* **by** *blast*
with *no-canonical-without-constructors*[
where *S* = *S* **and** *h* = *h* **and** *t* = *u*, *OF empty*]
show *False* **by** *blast*
qed

end
theory *HIT-Confluence*
imports *HIT-Normalization*
begin

17 Confluence

17.1 A dimension-capture critical pair

definition *capture-arg* :: *tm* **where**
capture-arg = *LCon 2 0* [] (*DVar 0*)

definition *capture-body* :: *tm* **where**
capture-body =
HElim 1 (*Hit 2*) [] [*Var 0*] (*Var 1*)

definition *capture-method* :: *tm* **where**
capture-method =
App (*Lam* (*Hit 2*) *capture-body*) *capture-arg*

definition *capture-source* :: *tm* **where**
capture-source =
HElim 0 (*Hit 2*) [] [*capture-method*]
(*LCon 0 0* [] (*DVar 1*))

definition *capture-left* :: *tm* **where**
capture-left =
HElim 1 (*Hit 2*) [] [*LCon 2 0* [] (*DVar 1*)] (*Var 0*)

definition *capture-right* :: *tm* **where**
capture-right =
HElim 1 (*Hit 2*) [] [*LCon 2 0* [] (*DVar 0*)] (*Var 0*)

lemma *capture-argument-not-dclosed*:
¬ *dclosed0 S capture-arg*

unfolding *dclosed0-def capture-arg-def* **by** *auto*

lemma *capture-substitute-then-instantiate*:
dsubst S 0 (DVar 1)
(subst 0 capture-arg capture-body) = capture-right
unfolding *capture-arg-def capture-body-def capture-right-def* **by** *simp*

lemma *capture-instantiate-then-substitute*:
subst 0 (dsubst S 0 (DVar 1) capture-arg)
(dsubst S 0 (DVar 1) capture-body) = capture-left
unfolding *capture-arg-def capture-body-def capture-left-def* **by** *simp*

lemma *capture-critical-pair-distinct*:
dsubst S 0 (DVar 1)
(subst 0 capture-arg capture-body) ≠
subst 0 (dsubst S 0 (DVar 1) capture-arg)
(dsubst S 0 (DVar 1) capture-body)
using *capture-substitute-then-instantiate*
capture-instantiate-then-substitute
unfolding *capture-left-def capture-right-def* **by** *simp*

The critical pair above is the reason that rule *red.beta* requires its argument to be stable under instantiation of the reserved dimension. A dimension-dependent beta redex is merely postponed until the surrounding path method has been instantiated. This is the smallest restriction that makes beta substitution commute with path computation; ordinary ambient terms satisfy it by dimension freshness.

17.2 Multi-step compatibility

lemma *reds-context*:
assumes *mono*: ⋀*x y*. *red S x y* ⟹ *red S* (*f x*) (*f y*)
and *steps*: *reds S x y*
shows *reds S* (*f x*) (*f y*)
using *steps*
proof (*induct rule*: *rtranclp-induct*)
case *base*
show *?case* **by** *simp*
next
case (*step y z*)
have *red S* (*f y*) (*f z*) **by** (*rule mono*[*OF step.hyps*(*2*)])
with *step.hyps*(*3*) **show** *?case* **by** (*rule rtranclp.rtrancl-into-rtrancl*)
qed

lemma *reds-App*:
assumes *reds S t t′* **and** *reds S u u′*
shows *reds S* (*App t u*) (*App t′ u′*)
proof −
have *left*: *reds S* (*App t u*) (*App t′ u*)

```
    by (rule reds-context[OF red.appL assms(1)])
  have right: reds S (App t' u) (App t' u')
    by (rule reds-context[OF red.appR assms(2)])
  show ?thesis by (rule rtranclp-trans[OF left right])
qed

lemma reds-Lam:
  reds S t t' ⟹ reds S (Lam A t) (Lam A t')
  by (rule reds-context[OF red.lam])

abbreviation list-reds ::
  sigenv ⇒ tm list ⇒ tm list ⇒ bool
where
  list-reds S xs ys ≡ list-all2 (reds S) xs ys

lemma list-reds-refl [simp]: list-reds S xs xs
  by (induct xs) auto

lemma list-reds-length:
  list-reds S xs ys ⟹ length xs = length ys
  by (rule list-all2-lengthD)

lemma list-reds-nth:
  assumes list-reds S xs ys and i < length xs
  shows reds S (xs ! i) (ys ! i)
  using assms by (metis list-all2-nthD)

lemma reds-PCon-lists:
  assumes list-reds S xs ys
  shows reds S (PCon h k (pre @ xs)) (PCon h k (pre @ ys))
  using assms
proof (induct arbitrary: pre)
  case Nil
  show ?case by simp
next
  case (Cons x xs y ys)
  have head: reds S
      (PCon h k (pre @ x # xs))
      (PCon h k (pre @ y # xs))
    by (rule reds-context[
          where f = λz. PCon h k (pre @ z # xs),
          OF - Cons.hyps(1)])
      (auto intro: red.pconArg)
  have tail: reds S
      (PCon h k ((pre @ [y]) @ xs))
      (PCon h k ((pre @ [y]) @ ys))
    by (rule Cons.hyps(3))
  have tail': reds S
      (PCon h k (pre @ y # xs))
```

```
    (PCon h k (pre @ y # ys))
   using tail by simp
 show ?case by (rule rtranclp-trans[OF head tail'])
qed

lemma reds-PCon:
 list-reds S xs ys ⟹
   reds S (PCon h k xs) (PCon h k ys)
 using reds-PCon-lists[where pre = []] by simp

lemma reds-LCon-lists:
 assumes list-reds S xs ys
 shows reds S (LCon h k (pre @ xs) d) (LCon h k (pre @ ys) d)
 using assms
proof (induct arbitrary: pre)
 case Nil
 show ?case by simp
next
 case (Cons x xs y ys)
 have head: reds S
    (LCon h k (pre @ x # xs) d)
    (LCon h k (pre @ y # xs) d)
   by (rule reds-context[
        where f = λz. LCon h k (pre @ z # xs) d,
        OF - Cons.hyps(1)])
    (auto intro: red.lconArg)
 have tail: reds S
    (LCon h k ((pre @ [y]) @ xs) d)
    (LCon h k ((pre @ [y]) @ ys) d)
   by (rule Cons.hyps(3))
 have tail': reds S
    (LCon h k (pre @ y # xs) d)
    (LCon h k (pre @ y # ys) d)
   using tail by simp
 show ?case by (rule rtranclp-trans[OF head tail'])
qed

lemma reds-LCon:
 list-reds S xs ys ⟹
   reds S (LCon h k xs d) (LCon h k ys d)
 using reds-LCon-lists[where pre = []] by simp

lemma reds-HElim-pms-lists:
 assumes list-reds S xs ys
 shows reds S
   (HElim h C (pre @ xs) lms t)
   (HElim h C (pre @ ys) lms t)
 using assms
proof (induct arbitrary: pre)
```

```
  case Nil
  show ?case by simp
next
  case (Cons x xs y ys)
  have head: reds S
      (HElim h C (pre @ x # xs) lms t)
      (HElim h C (pre @ y # xs) lms t)
    by (rule reds-context[
          where f = λz. HElim h C (pre @ z # xs) lms t,
          OF - Cons.hyps(1)])
      (auto intro: red.elimPm)
  have tail: reds S
      (HElim h C ((pre @ [y]) @ xs) lms t)
      (HElim h C ((pre @ [y]) @ ys) lms t)
    by (rule Cons.hyps(3))
  have tail': reds S
      (HElim h C (pre @ y # xs) lms t)
      (HElim h C (pre @ y # ys) lms t)
    using tail by simp
  show ?case by (rule rtranclp-trans[OF head tail'])
qed

lemma reds-HElim-pms:
  list-reds S pms pms' ⟹
    reds S (HElim h C pms lms t) (HElim h C pms' lms t)
  using reds-HElim-pms-lists[where pre = []] by simp

lemma reds-HElim-lms-lists:
  assumes list-reds S xs ys
  shows reds S
    (HElim h C pms (pre @ xs) t)
    (HElim h C pms (pre @ ys) t)
  using assms
proof (induct arbitrary: pre)
  case Nil
  show ?case by simp
next
  case (Cons x xs y ys)
  have head: reds S
      (HElim h C pms (pre @ x # xs) t)
      (HElim h C pms (pre @ y # xs) t)
    by (rule reds-context[
          where f = λz. HElim h C pms (pre @ z # xs) t,
          OF - Cons.hyps(1)])
      (auto intro: red.elimLm)
  have tail: reds S
      (HElim h C pms ((pre @ [y]) @ xs) t)
      (HElim h C pms ((pre @ [y]) @ ys) t)
    by (rule Cons.hyps(3))
```

```
  have tail': reds S
      (HElim h C pms (pre @ y # xs) t)
      (HElim h C pms (pre @ y # ys) t)
    using tail by simp
  show ?case by (rule rtranclp-trans[OF head tail'])
qed

lemma reds-HElim-lms:
  list-reds S lms lms' ⟹
    reds S (HElim h C pms lms t) (HElim h C pms lms' t)
  using reds-HElim-lms-lists[where pre = []] by simp

lemma reds-HElim-scrutinee:
  reds S t t' ⟹
    reds S (HElim h C pms lms t) (HElim h C pms lms t')
  by (rule reds-context[OF red.elimScrutinee])

lemma reds-HElim:
  assumes list-reds S pms pms' and list-reds S lms lms'
    and reds S t t'
  shows reds S
    (HElim h C pms lms t) (HElim h C pms' lms' t')
proof −
  have p: reds S
      (HElim h C pms lms t) (HElim h C pms' lms t)
    by (rule reds-HElim-pms[OF assms(1)])
  have l: reds S
      (HElim h C pms' lms t) (HElim h C pms' lms' t)
    by (rule reds-HElim-lms[OF assms(2)])
  have s: reds S
      (HElim h C pms' lms' t) (HElim h C pms' lms' t')
    by (rule reds-HElim-scrutinee[OF assms(3)])
  show ?thesis
    by (rule rtranclp-trans[OF rtranclp-trans[OF p l] s])
qed

lemma reds-apps:
  assumes reds S t t' and list-reds S xs ys
  shows reds S (apps t xs) (apps t' ys)
  using assms(2) assms(1)
proof (induct arbitrary: t t')
  case Nil
  then show ?case by simp
next
  case (Cons x xs y ys)
  have reds S (App t x) (App t' y)
    by (rule reds-App[OF Cons.prems Cons.hyps(1)])
  from Cons.hyps(3)[OF this] show ?case by simp
qed
```

```
lemma list-reds-trans:
  assumes list-reds S xs ys and list-reds S ys zs
  shows list-reds S xs zs
  using assms
proof (induct arbitrary: zs rule: list-all2-induct)
  case Nil
  then show ?case by (cases zs) auto
next
  case (Cons x xs y ys)
  then obtain z zs' where zs: zs = z # zs'
    by (cases zs) auto
  have yz: reds S y z and tails: list-reds S ys zs'
    using Cons.prems zs by auto
  have reds S x z
    by (rule rtranclp-trans[OF Cons.hyps(1) yz])
  moreover have list-reds S xs zs'
    by (rule Cons.hyps(3)[OF tails])
  ultimately show ?case using zs by auto
qed

lemma elimArgs-list-reds:
  assumes pms: list-reds S pms pms'
    and lms: list-reds S lms lms'
    and args: list-reds S as as'
  shows list-reds S
    (elimArgs h C pms lms ss as)
    (elimArgs h C pms' lms' ss as')
  using args
proof (induct ss arbitrary: as as')
  case Nil
  show ?case by simp
next
  case (Cons s ss)
  show ?case
  proof (cases as)
    case Nil
    then have as' = [] using Cons.prems by (cases as') auto
    with Nil show ?thesis by simp
  next
    case (Cons a xs)
    then obtain b ys where
      as': as' = b # ys and ab: reds S a b
      and tails: list-reds S xs ys
      using Cons.prems by (cases as') auto
    have rest: list-reds S
        (elimArgs h C pms lms ss xs)
        (elimArgs h C pms' lms' ss ys)
      by (rule Cons.hyps[OF tails])
```

```
    show ?thesis
    proof (cases s)
      case (Par h')
      with Cons as' ab rest show ?thesis by simp
    next
      case Rec
      have rec: reds S
          (HElim h C pms lms a)
          (HElim h C pms' lms' b)
        by (rule reds-HElim[OF pms lms ab])
      with Rec Cons as' ab rest show ?thesis by simp
    qed
  qed
qed

lemma reds-subst-argument:
  assumes step: red S s s' and reg: regular S s
  shows reds S (subst n s t) (subst n s' t)
  using step reg
proof (induct t arbitrary: n s s')
  case (Var i)
  show ?case
  proof (cases i < n)
    case True
    then show ?thesis by simp
  next
    case False
    show ?thesis
    proof (cases i = n)
      case True
      have reds S s s'
        by (rule r-into-rtranclp[where r = red S, OF Var.prems(1)])
      with True show ?thesis by simp
    next
      case False
      with ‹¬ i < n› show ?thesis by simp
    qed
  qed
next
  case (Lam A t)
  have lifted: red S (lift 0 s) (lift 0 s')
    using red-vrename[OF Lam.prems(1,2), of shift-ren 0]
    by (simp add: lift-as-vrename)
  have reg-lift: regular S (lift 0 s)
    by (rule regular-lift[OF Lam.prems(2)])
  have reds S
      (subst (Suc n) (lift 0 s) t)
      (subst (Suc n) (lift 0 s') t)
    by (rule Lam.hyps[OF lifted reg-lift])
```

then show *?case* **by** (*auto intro*: *reds-Lam*)
next
case (*App t u*)
have *left*: *reds S* (*subst n s t*) (*subst n s′ t*)
by (*rule App.hyps*(*1*)[*OF App.prems*])
have *right*: *reds S* (*subst n s u*) (*subst n s′ u*)
by (*rule App.hyps*(*2*)[*OF App.prems*])
show *?case* **by** (*simp add*: *reds-App*[*OF left right*])
next
case (*PCon h k as*)
have *lr*: *list-reds S*
(*map* (*subst n s*) *as*) (*map* (*subst n s′*) *as*)
using *PCon.hyps PCon.prems*
by (*induct as*) *auto*
show *?case* **by** (*simp add*: *reds-PCon*[*OF lr*])
next
case (*LCon h k as d*)
have *lr*: *list-reds S*
(*map* (*subst n s*) *as*) (*map* (*subst n s′*) *as*)
using *LCon.hyps LCon.prems*
by (*induct as*) *auto*
show *?case* **by** (*simp add*: *reds-LCon*[*OF lr*])
next
case (*HElim h C pms lms t*)
have *p*: *list-reds S*
(*map* (*subst n s*) *pms*) (*map* (*subst n s′*) *pms*)
using *HElim.hyps*(*1*) *HElim.prems* **by** (*induct pms*) *auto*
have *l*: *list-reds S*
(*map* (*subst n s*) *lms*) (*map* (*subst n s′*) *lms*)
using *HElim.hyps*(*2*) *HElim.prems* **by** (*induct lms*) *auto*
have *q*: *reds S* (*subst n s t*) (*subst n s′ t*)
by (*rule HElim.hyps*(*3*)[*OF HElim.prems*])
show *?case* **by** (*simp add*: *reds-HElim*[*OF p l q*])
qed

17.3 Dimension-safe reduction

lemma *dclosed0-lift*:
assumes *variable-dims t* **and** *dclosed0 S t*
shows *dclosed0 S* (*lift n t*)
using *dclosed0-vrename*[*OF assms*(*1*), *of S shift-ren n*] *assms*(*2*)
by (*simp add*: *lift-as-vrename*)

lemma *dclosed0-up-sub*:
assumes ⋀*i*. *dclosed0 S* (*s i*)
and ⋀*i*. *variable-dims* (*s i*)
shows *dclosed0 S* (*up-sub s i*)
using *assms*
by (*cases i*) (*auto intro*: *dclosed0-lift*)

lemma *psubst-dsubst-zero-dclosed*:
  assumes *dims*: *variable-dims t*
    and *env-closed*: $\bigwedge i$. *dclosed0 S* (*s i*)
    and *env-dims*: $\bigwedge i$. *variable-dims* (*s i*)
  shows *psubst s* (*dsubst S 0* (*DVar j*) *t*) =
    dsubst S 0 (*DVar j*) (*psubst s t*)
  using *dims env-closed env-dims*
proof (*induct t arbitrary*: *s j*)
  case (*Var i*)
  then show *?case*
    using *Var.prems*(*2*)[*of i*] **unfolding** *dclosed0-def* **by** *simp*
next
  case (*Lam A t*)
  have *closed*: $\bigwedge i$. *dclosed0 S* (*up-sub s i*)
    by (*rule dclosed0-up-sub*[*OF Lam.prems*(*2*,*3*)])
  have *dims*: $\bigwedge i$. *variable-dims* (*up-sub s i*)
    by (*rule variable-dims-up-sub*[*OF Lam.prems*(*3*)])
  from *Lam.hyps*[*OF - closed dims*] *Lam.prems* **show** *?case* **by** *simp*
next
  case (*LCon h k as d*)
  then obtain *q* **where** *d*: *d* = *DVar q* **by** *auto*
  from *LCon d* **show** *?case*
    by (*cases q* = *0*) (*auto simp*: *list-all-iff map-map*)
qed (*auto simp*: *list-all-iff map-map*)

lemma *dclosed0-psubst-closed*:
  assumes *dims*: *variable-dims t* **and** *closed*: *dclosed0 S t*
    and *env-closed*: $\bigwedge i$. *dclosed0 S* (*s i*)
    and *env-dims*: $\bigwedge i$. *variable-dims* (*s i*)
  shows *dclosed0 S* (*psubst s t*)
proof (*unfold dclosed0-def*, *intro allI*)
  fix *j*
  have *dsubst S 0* (*DVar j*) (*psubst s t*) =
      psubst s (*dsubst S 0* (*DVar j*) *t*)
    using *psubst-dsubst-zero-dclosed*[
      where *S* = *S* **and** *t* = *t* **and** *s* = *s* **and** *j* = *j*,
      OF dims env-closed env-dims]
    by *simp*
  also have ... = *psubst s t*
    using *closed* **unfolding** *dclosed0-def* **by** *simp*
  finally show *dsubst S 0* (*DVar j*) (*psubst s t*) = *psubst s t* **.**
qed

lemma *dclosed0-subst*:
  assumes *dims-t*: *variable-dims t* **and** *closed-t*: *dclosed0 S t*
    and *dims-s*: *variable-dims s* **and** *closed-s*: *dclosed0 S s*
  shows *dclosed0 S* (*subst n s t*)
proof −

```
  have env-closed: ⋀i. dclosed0 S (single-sub n s i)
    unfolding single-sub-def
    using closed-s by (auto intro: dclosed0-lift)
  have env-dims: ⋀i. variable-dims (single-sub n s i)
    unfolding single-sub-def
    using dims-s by (auto intro: variable-dims-lift)
  have dclosed0 S (psubst (single-sub n s) t)
    by (rule dclosed0-psubst-closed[
          OF dims-t closed-t env-closed env-dims])
  then show ?thesis by (simp add: subst-as-psubst)
qed

lemma red-psubst-dclosed:
  assumes step: red S t u and reg: regular S t
    and env-reg: ⋀i. regular S (s i)
    and env-closed: ⋀i. dclosed0 S (s i)
  shows red S (psubst s t) (psubst s u)
  using step reg env-reg env-closed
proof (induct arbitrary: s rule: red.induct)
  case (beta u A t)
  have regt: regular S t and regu: regular S u
    using beta.prems(1) by simp-all
  have dimst: variable-dims t
    by (rule regular-variable-dims[OF regt])
  have dimsu: variable-dims u
    by (rule regular-variable-dims[OF regu])
  have env-dims: ⋀i. variable-dims (s i)
    by (rule regular-variable-dims[OF beta.prems(2)])
  have closed-arg: dclosed0 S (psubst s u)
    by (rule dclosed0-psubst-closed[
          OF dimsu beta.hyps(1) beta.prems(3) env-dims])
  have raw: red S
      (App (Lam A (psubst (up-sub s) t)) (psubst s u))
      (subst 0 (psubst s u) (psubst (up-sub s) t))
    by (rule red.beta[OF closed-arg])
  from raw show ?case by (simp add: subst0-psubst)
next
  case (elimPoint k h C pms lms as)
  have raw: red S
      (HElim h C (map (psubst s) pms) (map (psubst s) lms)
        (PCon h k (map (psubst s) as)))
      (apps ((map (psubst s) pms) ! k)
        (elimArgs h C (map (psubst s) pms) (map (psubst s) lms)
          (pcons (S h) ! k) (map (psubst s) as)))
    by (rule red.elimPoint[
          where S = S and h = h and k = k and C = C
            and pms = map (psubst s) pms
            and lms = map (psubst s) lms
            and as = map (psubst s) as,
```

```
        OF elimPoint.hyps(1)])
  from raw show ?case
    using elimPoint by (simp add: psubst-elimArgs)
next
  case (elimPath k h C pms lms as j)
  have mreg: regular S (lms ! k)
    using elimPath by (auto simp: list-all-iff)
  have mdims: variable-dims (lms ! k)
    by (rule regular-variable-dims[OF mreg])
  have env-dims: ⋀i. variable-dims (s i)
    by (rule regular-variable-dims[OF elimPath.prems(2)])
  have method:
      psubst s (dsubst S 0 (DVar j) (lms ! k)) =
        dsubst S 0 (DVar j) (psubst s (lms ! k))
    by (rule psubst-dsubst-zero-dclosed[
        OF mdims elimPath.prems(3) env-dims])
  have raw: red S
      (HElim h C (map (psubst s) pms) (map (psubst s) lms)
        (LCon h k (map (psubst s) as) (DVar j)))
      (apps (dsubst S 0 (DVar j) ((map (psubst s) lms) ! k))
        (elimArgs h C (map (psubst s) pms) (map (psubst s) lms)
          (lcons (S h) ! k) (map (psubst s) as)))
    by (rule red.elimPath[
        where S = S and h = h and k = k and C = C
          and pms = map (psubst s) pms
          and lms = map (psubst s) lms
          and as = map (psubst s) as and j = j,
        OF elimPath.hyps(1)])
  from raw method show ?case
    using elimPath by (simp add: psubst-elimArgs)
next
  case (appL t t' u)
  then show ?case by (auto intro: red.appL)
next
  case (appR u u' t)
  then show ?case by (auto intro: red.appR)
next
  case (lam t t' A)
  have up-reg: ⋀i. regular S (up-sub s i)
    by (rule regular-up-sub[OF lam.prems(2)])
  have env-dims: ⋀i. variable-dims (s i)
    by (rule regular-variable-dims[OF lam.prems(2)])
  have up-closed: ⋀i. dclosed0 S (up-sub s i)
    by (rule dclosed0-up-sub[OF lam.prems(3) env-dims])
  have red S (psubst (up-sub s) t) (psubst (up-sub s) t')
    by (rule lam.hyps(2)[OF - up-reg up-closed])
      (use lam.prems(1) in simp)
  then show ?case by (auto intro: red.lam)
next
```

```
  case (pconArg a a' h k xs ys)
  have rega: regular S a
    using pconArg.prems(1) by (auto simp: list-all-iff)
  have red S (psubst s a) (psubst s a')
    by (rule pconArg.hyps(2)[OF rega pconArg.prems(2,3)])
  then show ?case by (auto intro: red.pconArg)
next
  case (lconArg a a' h k xs ys d)
  have rega: regular S a
    using lconArg.prems(1) by (auto simp: list-all-iff)
  have red S (psubst s a) (psubst s a')
    by (rule lconArg.hyps(2)[OF rega lconArg.prems(2,3)])
  then show ?case by (auto intro: red.lconArg)
next
  case (elimPm m m' h C xs ys lms t)
  have regm: regular S m
    using elimPm.prems(1) by (auto simp: list-all-iff)
  have red S (psubst s m) (psubst s m')
    by (rule elimPm.hyps(2)[OF regm elimPm.prems(2,3)])
  then show ?case by (auto intro: red.elimPm)
next
  case (elimLm m m' h C pms xs ys t)
  have regm: regular S m
    using elimLm.prems(1) by (auto simp: list-all-iff)
  have red S (psubst s m) (psubst s m')
    by (rule elimLm.hyps(2)[OF regm elimLm.prems(2,3)])
  then show ?case by (auto intro: red.elimLm)
next
  case (elimScrutinee t t' h C pms lms)
  have regt: regular S t using elimScrutinee.prems(1) by simp
  have red S (psubst s t) (psubst s t')
    by (rule elimScrutinee.hyps(2)[
          OF regt elimScrutinee.prems(2,3)])
  then show ?case by (auto intro: red.elimScrutinee)
qed

lemma red-subst-dclosed:
  assumes step: red S t t' and regt: regular S t
    and regs: regular S s and closeds: dclosed0 S s
  shows red S (subst n s t) (subst n s t')
proof −
  have env-reg: ⋀i. regular S (single-sub n s i)
    unfolding single-sub-def
    using regs by (auto intro: regular-lift)
  have env-closed: ⋀i. dclosed0 S (single-sub n s i)
    unfolding single-sub-def
    using regs closeds regular-variable-dims
    by (auto intro: dclosed0-lift)
  have red S
```

```
    (psubst (single-sub n s) t)
    (psubst (single-sub n s) t')
  by (rule red-psubst-dclosed[
      OF step regt env-reg env-closed])
 then show ?thesis by (simp add: subst-as-psubst)
qed

lemma dclosed0-elimArgs:
 assumes pms: list-all (dclosed0 S) pms
   and args: list-all (dclosed0 S) as
 shows list-all (dclosed0 S) (elimArgs h C pms lms ss as)
 using args
proof (induct ss arbitrary: as)
 case Nil
 show ?case by simp
next
 case (Cons s ss)
 then show ?case
   using pms by (cases as; cases s) auto
qed

lemma dclosed0-apps:
 assumes dclosed0 S t and list-all (dclosed0 S) xs
 shows dclosed0 S (apps t xs)
 using assms by (induct xs arbitrary: t) auto

lemma dsubst-variable-closed:
 assumes reg: regular S t and j ≠ 0
 shows dclosed0 S (dsubst S 0 (DVar j) t)
 using reg assms(2)
 by (induct t) (auto simp: list-all-iff)

lemma dclosed0-red:
 assumes step: red S t u and reg: regular S t
   and closed: dclosed0 S t
 shows dclosed0 S u
 using step reg closed
proof (induct rule: red.induct)
 case (beta u A t)
 have regt: regular S t and regu: regular S u
   using beta.prems(1) by simp-all
 have closet: dclosed0 S t and closeu: dclosed0 S u
   using beta.prems(2) by simp-all
 show ?case
   by (rule dclosed0-subst[
       OF regular-variable-dims[OF regt] closet
         regular-variable-dims[OF regu] closeu])
next
 case (elimPoint k h C pms lms as)
```

```
  have cpms: list-all (dclosed0 S) pms
    and cas: list-all (dclosed0 S) as
    using elimPoint.prems(2) by simp-all
  have kpm: k < length pms using elimPoint by simp
  have method: dclosed0 S (pms ! k)
    using cpms kpm by (simp add: list-all-iff)
  have eargs: list-all (dclosed0 S)
      (elimArgs h C pms lms (pcons (S h) ! k) as)
    by (rule dclosed0-elimArgs[OF cpms cas])
  show ?case by (rule dclosed0-apps[OF method eargs])
next
  case (elimPath k h C pms lms as j)
  have cpms: list-all (dclosed0 S) pms
    and cas: list-all (dclosed0 S) as and j0: j ≠ 0
    using elimPath.prems(2) by simp-all
  have k': k < length lms using elimPath by simp
  have mreg: regular S (lms ! k)
    using elimPath k' by (auto simp: list-all-iff)
  have method: dclosed0 S (dsubst S 0 (DVar j) (lms ! k))
    by (rule dsubst-variable-closed[OF mreg j0])
  have eargs: list-all (dclosed0 S)
      (elimArgs h C pms lms (lcons (S h) ! k) as)
    by (rule dclosed0-elimArgs[OF cpms cas])
  show ?case by (rule dclosed0-apps[OF method eargs])
next
  case (appL t t' u)
  then show ?case by simp
next
  case (appR u u' t)
  then show ?case by simp
next
  case (lam t t' A)
  then show ?case by simp
next
  case (pconArg a a' h k xs ys)
  then show ?case by (auto simp: list-all-iff)
next
  case (lconArg a a' h k xs ys d)
  then show ?case by (auto simp: list-all-iff)
next
  case (elimPm m m' h C xs ys lms t)
  then show ?case by (auto simp: list-all-iff)
next
  case (elimLm m m' h C pms xs ys t)
  then show ?case by simp
next
  case (elimScrutinee t t' h C pms lms)
  then show ?case by simp
qed
```

lemma *dsubst-same-regular*:
assumes *regular S t*
shows *dsubst S 0 (DVar 0) t = t*
using *assms*
by (*induct t*) (*auto simp*: *list-all-iff map-idI*)

lemma *dsubst-variable-comp*:
assumes *regular S t*
shows *dsubst S 0 (DVar j) (dsubst S 0 (DVar q) t) =*
dsubst S 0 (DVar (if q = 0 then j else q)) t
proof (*cases q = 0*)
case *True*
then show *?thesis* **using** *dsubst-same-regular*[*OF assms*] **by** *simp*
next
case *False*
have *closed*: *dclosed0 S (dsubst S 0 (DVar q) t)*
by (*rule dsubst-variable-closed*[*OF assms False*])
then have *dsubst S 0 (DVar j) (dsubst S 0 (DVar q) t) =*
dsubst S 0 (DVar q) t
unfolding *dclosed0-def* **by** *simp*
with *False* **show** *?thesis* **by** *simp*
qed

lemma *dsubst-apps* [*simp*]:
dsubst S 0 (DVar j) (apps t xs) =
apps (dsubst S 0 (DVar j) t)
(map (dsubst S 0 (DVar j)) xs)
by (*induct xs arbitrary*: *t*) *simp-all*

lemma *dsubst-elimArgs*:
map (dsubst S 0 (DVar j))
(elimArgs h C pms lms ss as) =
elimArgs h C (map (dsubst S 0 (DVar j)) pms) lms ss
(map (dsubst S 0 (DVar j)) as)
by (*induct ss arbitrary*: *as*)
(*auto split*: *list.split argspec.split*)

lemma *red-dsubst-variable*:
assumes *step*: *red S t u* **and** *reg*: *regular S t*
shows *red S*
(*dsubst S 0 (DVar j) t*) (*dsubst S 0 (DVar j) u*)
using *step reg*
proof (*induct rule*: *red.induct*)
case (*beta u A t*)
have *regt*: *regular S t* **and** *regu*: *regular S u*
using *beta.prems* **by** *simp-all*
have *dt*: *variable-dims t* **by** (*rule regular-variable-dims*[*OF regt*])
have *du*: *variable-dims u* **by** (*rule regular-variable-dims*[*OF regu*])

```
  have closed: dclosed0 S u by (rule beta.hyps(1))
  have fixed: dsubst S 0 (DVar j) u = u
    using closed unfolding dclosed0-def by simp
  have commute:
      dsubst S 0 (DVar j) (subst 0 u t) =
        subst 0 u (dsubst S 0 (DVar j) t)
  proof −
    have env-closed: ⋀i. dclosed0 S (single-sub 0 u i)
      unfolding single-sub-def
      using closed by (auto intro: dclosed0-lift)
    have env-dims: ⋀i. variable-dims (single-sub 0 u i)
      unfolding single-sub-def
      using du by (auto intro: variable-dims-lift)
    have psubst (single-sub 0 u) (dsubst S 0 (DVar j) t) =
        dsubst S 0 (DVar j) (psubst (single-sub 0 u) t)
      by (rule psubst-dsubst-zero-dclosed[
          OF dt env-closed env-dims])
    then show ?thesis by (simp add: subst-as-psubst)
  qed
  have red S
      (App (Lam A (dsubst S 0 (DVar j) t)) u)
      (subst 0 u (dsubst S 0 (DVar j) t))
    by (rule red.beta[OF closed])
  with fixed commute show ?case by simp
next
  case (elimPoint k h C pms lms as)
  have raw: red S
      (HElim h C (map (dsubst S 0 (DVar j)) pms) lms
        (PCon h k (map (dsubst S 0 (DVar j)) as)))
      (apps ((map (dsubst S 0 (DVar j)) pms) ! k)
        (elimArgs h C (map (dsubst S 0 (DVar j)) pms) lms
          (pcons (S h) ! k)
          (map (dsubst S 0 (DVar j)) as)))
    by (rule red.elimPoint[
          where S = S and h = h and k = k and C = C
            and pms = map (dsubst S 0 (DVar j)) pms
            and lms = lms
            and as = map (dsubst S 0 (DVar j)) as,
          OF elimPoint.hyps(1)])
  from raw show ?case
    using elimPoint.hyps(1) elimPoint.prems
    by (simp add: dsubst-elimArgs)
next
  case (elimPath k h C pms lms as q)
  have mreg: regular S (lms ! k)
    using elimPath by (auto simp: list-all-iff)
  let ?q = if q = 0 then j else q
  have raw: red S
      (HElim h C (map (dsubst S 0 (DVar j)) pms) lms
```

```
      (LCon h k (map (dsubst S 0 (DVar j)) as) (DVar ?q)))
    (apps (dsubst S 0 (DVar ?q) (lms ! k))
      (elimArgs h C (map (dsubst S 0 (DVar j)) pms) lms
        (lcons (S h) ! k)
        (map (dsubst S 0 (DVar j)) as)))
  by (rule red.elimPath[
       where S = S and h = h and k = k and C = C
         and pms = map (dsubst S 0 (DVar j)) pms
         and lms = lms
         and as = map (dsubst S 0 (DVar j)) as
         and j = ?q,
       OF elimPath.hyps(1)])
 have method:
    dsubst S 0 (DVar j)
      (dsubst S 0 (DVar q) (lms ! k)) =
    dsubst S 0 (DVar ?q) (lms ! k)
  by (rule dsubst-variable-comp[OF mreg])
 show ?case
 proof (cases q = 0)
  case True
  from raw method True show ?thesis
   by (simp add: dsubst-elimArgs)
 next
  case False
  from raw method False show ?thesis
   by (simp add: dsubst-elimArgs)
 qed
next
 case (appL t t' u)
 then show ?case by (auto intro: red.appL)
next
 case (appR u u' t)
 then show ?case by (auto intro: red.appR)
next
 case (lam t t' A)
 then show ?case by (auto intro: red.lam)
next
 case (pconArg a a' h k xs ys)
 have red S
    (PCon h k
      (map (dsubst S 0 (DVar j)) xs @
        dsubst S 0 (DVar j) a #
        map (dsubst S 0 (DVar j)) ys))
    (PCon h k
      (map (dsubst S 0 (DVar j)) xs @
        dsubst S 0 (DVar j) a' #
        map (dsubst S 0 (DVar j)) ys))
  by (rule red.pconArg) (use pconArg in auto)
 then show ?case by simp
```

```
next
  case (lconArg a a' h k xs ys d)
  then obtain q where d: d = DVar q by auto
  have red S
      (LCon h k
        (map (dsubst S 0 (DVar j)) xs @
          dsubst S 0 (DVar j) a #
          map (dsubst S 0 (DVar j)) ys)
        (DVar (if q = 0 then j else q)))
      (LCon h k
        (map (dsubst S 0 (DVar j)) xs @
          dsubst S 0 (DVar j) a' #
          map (dsubst S 0 (DVar j)) ys)
        (DVar (if q = 0 then j else q)))
    by (rule red.lconArg) (use lconArg d in auto)
  show ?case
  proof (cases q = 0)
    case True
    with d ‹red S
      (LCon h k
        (map (dsubst S 0 (DVar j)) xs @
          dsubst S 0 (DVar j) a #
          map (dsubst S 0 (DVar j)) ys)
        (DVar (if q = 0 then j else q)))
      (LCon h k
        (map (dsubst S 0 (DVar j)) xs @
          dsubst S 0 (DVar j) a' #
          map (dsubst S 0 (DVar j)) ys)
        (DVar (if q = 0 then j else q)))›
    show ?thesis by simp
  next
    case False
    with d ‹red S
      (LCon h k
        (map (dsubst S 0 (DVar j)) xs @
          dsubst S 0 (DVar j) a #
          map (dsubst S 0 (DVar j)) ys)
        (DVar (if q = 0 then j else q)))
      (LCon h k
        (map (dsubst S 0 (DVar j)) xs @
          dsubst S 0 (DVar j) a' #
          map (dsubst S 0 (DVar j)) ys)
        (DVar (if q = 0 then j else q)))›
    show ?thesis by simp
  qed
next
  case (elimPm m m' h C xs ys lms t)
  have red S
      (HElim h C
```

```
      (map (dsubst S 0 (DVar j)) xs @
        dsubst S 0 (DVar j) m #
        map (dsubst S 0 (DVar j)) ys)
      lms (dsubst S 0 (DVar j) t))
    (HElim h C
      (map (dsubst S 0 (DVar j)) xs @
        dsubst S 0 (DVar j) m' #
        map (dsubst S 0 (DVar j)) ys)
      lms (dsubst S 0 (DVar j) t))
  by (rule red.elimPm) (use elimPm in auto)
 then show ?case by simp
next
 case (elimLm m m' h C pms xs ys t)
 have red S
    (HElim h C (map (dsubst S 0 (DVar j)) pms)
      (xs @ m # ys) (dsubst S 0 (DVar j) t))
    (HElim h C (map (dsubst S 0 (DVar j)) pms)
      (xs @ m' # ys) (dsubst S 0 (DVar j) t))
  by (rule red.elimLm[OF elimLm.hyps(1)])
 then show ?case by simp
next
 case (elimScrutinee t t' h C pms lms)
 then show ?case by (auto intro: red.elimScrutinee)
qed

lemma reds-dsubst-variable:
 assumes reds S t u and regular S t
 shows reds S
  (dsubst S 0 (DVar j) t) (dsubst S 0 (DVar j) u)
proof −
 have invariant: regular S u ∧ reds S
    (dsubst S 0 (DVar j) t) (dsubst S 0 (DVar j) u)
  using assms(1)
 proof (induct rule: rtranclp-induct)
  case base
  show ?case using assms(2) by simp
 next
  case (step y z)
  have regy: regular S y using step.hyps(3) by blast
  have regz: regular S z
   by (rule regular-red[OF step.hyps(2) regy])
  have mapped: red S
      (dsubst S 0 (DVar j) y) (dsubst S 0 (DVar j) z)
   by (rule red-dsubst-variable[OF step.hyps(2) regy])
  have path: reds S
      (dsubst S 0 (DVar j) t) (dsubst S 0 (DVar j) y)
   using step.hyps(3) by blast
  have reds S
      (dsubst S 0 (DVar j) t) (dsubst S 0 (DVar j) z)
```

```
    by (rule rtranclp.rtrancl-into-rtrancl[OF path mapped])
  with regz show ?case by blast
 qed
 then show ?thesis by blast
qed
```

17.4 Local peaks

```
lemma red-as-reds:
 red S t u ⟹ reds S t u
 by (rule r-into-rtranclp[where r = red S])

lemma list-red-imp-list-reds:
 assumes list-red S xs ys
 shows list-reds S xs ys
 using assms
proof induct
 case (lr-head a b xs)
 have reds S a b
  by (rule r-into-rtranclp[where r = red S, OF lr-head.hyps])
 then show ?case by auto
next
 case (lr-tail xs ys a)
 then show ?case by auto
qed

lemma list-red-ConsE:
 assumes list-red S (a # xs) ys
 obtains
  (head) b where red S a b ys = b # xs
 | (tail) zs where list-red S xs zs ys = a # zs
 using assms by (cases rule: list-red.cases) auto

lemma list-red-local-confluence:
 assumes local: ⋀a b c. a ∈ set xs ⟹
    red S a b ⟹ red S a c ⟹ joinable S b c
  and left: list-red S xs ys
  and right: list-red S xs zs
 shows ∃ws. list-reds S ys ws ∧ list-reds S zs ws
 using left right local
proof (induct arbitrary: zs)
 case (lr-head a b xs)
 note ab-step = lr-head.hyps
 note local-all = lr-head.prems(2)
 show ?case
 proof (rule list-red-ConsE[OF lr-head.prems(1)])
  fix c
  assume ac: red S a c and zs: zs = c # xs
  have mem: a ∈ set (a # xs) by simp
```

```
    have joinable S b c
      by (rule local-all[OF mem ab-step ac])
    then obtain d where bd: reds S b d and cd: reds S c d
      unfolding joinable-def by blast
    have list-reds S (b # xs) (d # xs) using bd by auto
    moreover have list-reds S (c # xs) (d # xs) using cd by auto
    ultimately show ?thesis using zs by blast
  next
    fix xs'
    assume tail-step: list-red S xs xs' and zs: zs = a # xs'
    have tail: list-reds S xs xs'
      by (rule list-red-imp-list-reds[OF tail-step])
    have first: list-reds S (b # xs) (b # xs') using tail by auto
    have ab: reds S a b
      by (rule r-into-rtranclp[where r = red S, OF ab-step])
    have second: list-reds S (a # xs') (b # xs') using ab by auto
    from first second zs show ?thesis by blast
  qed
next
  case (lr-tail xs ys a)
  note tail-step = lr-tail.hyps(1)
  note tail-IH = lr-tail.hyps(2)
  note local-all = lr-tail.prems(2)
  show ?case
  proof (rule list-red-ConsE[OF lr-tail.prems(1)])
    fix b
    assume ab-step: red S a b and zs: zs = b # xs
    have tail: list-reds S xs ys
      by (rule list-red-imp-list-reds[OF tail-step])
    have first: list-reds S (a # ys) (b # ys)
    proof −
      have reds S a b
        by (rule r-into-rtranclp[where r = red S, OF ab-step])
      then show ?thesis by auto
    qed
    have second: list-reds S (b # xs) (b # ys) using tail by auto
    from first second zs show ?thesis by blast
  next
    fix zs'
    assume right-tail: list-red S xs zs'
      and zs: zs = a # zs'
    have local-tail: ⋀x b c. x ∈ set xs ⟹
        red S x b ⟹ red S x c ⟹ joinable S b c
      using local-all by auto
    have ∃ ws. list-reds S ys ws ∧ list-reds S zs' ws
      by (rule tail-IH[OF right-tail local-tail])
    then obtain ws where yw: list-reds S ys ws
      and zw: list-reds S zs' ws
      by blast
```

```
    have list-reds S (a # ys) (a # ws) using yw by auto
    moreover have list-reds S (a # zs') (a # ws) using zw by auto
    ultimately show ?thesis using zs by blast
  qed
qed

lemma point-contractum-reds:
  assumes k: k < length pms
    and pms: list-reds S pms pms'
    and lms: list-reds S lms lms'
    and args: list-reds S as as'
  shows reds S
    (apps (pms ! k)
      (elimArgs h C pms lms ss as))
    (apps (pms' ! k)
      (elimArgs h C pms' lms' ss as'))
proof −
  have method: reds S (pms ! k) (pms' ! k)
    by (rule list-reds-nth[OF pms k])
  have eargs: list-reds S
      (elimArgs h C pms lms ss as)
      (elimArgs h C pms' lms' ss as')
    by (rule elimArgs-list-reds[OF pms lms args])
  show ?thesis by (rule reds-apps[OF method eargs])
qed

lemma path-contractum-reds:
  assumes k: k < length lms
    and reg: list-all (regular S) lms
    and pms: list-reds S pms pms'
    and lms: list-reds S lms lms'
    and args: list-reds S as as'
  shows reds S
    (apps (dsubst S 0 (DVar j) (lms ! k))
      (elimArgs h C pms lms ss as))
    (apps (dsubst S 0 (DVar j) (lms' ! k))
      (elimArgs h C pms' lms' ss as'))
proof −
  have method0: reds S (lms ! k) (lms' ! k)
    by (rule list-reds-nth[OF lms k])
  have mreg: regular S (lms ! k)
    using reg k by (simp add: list-all-iff)
  have method: reds S
      (dsubst S 0 (DVar j) (lms ! k))
      (dsubst S 0 (DVar j) (lms' ! k))
    by (rule reds-dsubst-variable[OF method0 mreg])
  have eargs: list-reds S
      (elimArgs h C pms lms ss as)
      (elimArgs h C pms' lms' ss as')
```

```
    by (rule elimArgs-list-reds[OF pms lms args])
  show ?thesis by (rule reds-apps[OF method eargs])
qed

lemma red-AppE [case-names beta left right]:
  assumes red S (App t u) v
  obtains
    (beta) A body where t = Lam A body dclosed0 S u
      v = subst 0 u body
  | (left) t' where red S t t' v = App t' u
  | (right) u' where red S u u' v = App t u'
  using assms by (cases rule: red.cases) auto

lemma beta-body-peak:
  assumes body: red S t t' and regt: regular S t
    and regu: regular S u and closed: dclosed0 S u
  shows joinable S
    (subst 0 u t) (App (Lam A t') u)
proof −
  have left: reds S (subst 0 u t) (subst 0 u t')
    by (rule red-as-reds[
          OF red-subst-dclosed[OF body regt regu closed]])
  have right-step: red S (App (Lam A t') u) (subst 0 u t')
    by (rule red.beta[OF closed])
  have right: reds S (App (Lam A t') u) (subst 0 u t')
    by (rule red-as-reds[OF right-step])
  show ?thesis by (rule joinableI[OF left right])
qed

lemma beta-argument-peak:
  assumes arg: red S u u' and regu: regular S u
    and closed: dclosed0 S u
  shows joinable S
    (subst 0 u t) (App (Lam A t) u')
proof −
  have left: reds S (subst 0 u t) (subst 0 u' t)
    by (rule reds-subst-argument[OF arg regu])
  have regu': regular S u' by (rule regular-red[OF arg regu])
  have closed': dclosed0 S u'
    by (rule dclosed0-red[OF arg regu closed])
  have right-step: red S (App (Lam A t) u') (subst 0 u' t)
    by (rule red.beta[OF closed'])
  have right: reds S (App (Lam A t) u') (subst 0 u' t)
    by (rule red-as-reds[OF right-step])
  show ?thesis by (rule joinableI[OF left right])
qed

lemma point-root-peak:
  assumes reg: regular S (HElim h C pms lms (PCon h k as))
```

```
    and k: k < length (pcons (S h))
    and other: red S (HElim h C pms lms (PCon h k as)) v
  shows joinable S
    (apps (pms ! k)
      (elimArgs h C pms lms (pcons (S h) ! k) as)) v
  using other
proof (rule red-HElim-PConE)
  assume k < length (pcons (S h))
    and v: v = apps (pms ! k)
      (elimArgs h C pms lms (pcons (S h) ! k) as)
  then show ?thesis by simp
next
  fix xs m m' ys
  assume pms: pms = xs @ m # ys and mm': red S m m'
    and v: v = HElim h C (xs @ m' # ys) lms (PCon h k as)
  let ?pms' = xs @ m' # ys
  have lr: list-reds S pms ?pms'
    using pms list-red-imp-list-reds[
      OF list-red-append[OF mm', of xs ys]]
    by simp
  have kp: k < length pms using reg k by simp
  have left: reds S
      (apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k) as))
      (apps (?pms' ! k)
        (elimArgs h C ?pms' lms (pcons (S h) ! k) as))
    by (rule point-contractum-reds[OF kp lr])
      simp-all
  have right-step: red S
      (HElim h C ?pms' lms (PCon h k as))
      (apps (?pms' ! k)
        (elimArgs h C ?pms' lms (pcons (S h) ! k) as))
    by (rule red.elimPoint[
          where S = S and h = h and k = k and C = C
            and pms = ?pms' and lms = lms and as = as,
          OF k])
  have right: reds S v
      (apps (?pms' ! k)
        (elimArgs h C ?pms' lms (pcons (S h) ! k) as))
    using v red-as-reds[OF right-step] by simp
  show ?thesis by (rule joinableI[OF left right])
next
  fix xs m m' ys
  assume lms: lms = xs @ m # ys and mm': red S m m'
    and v: v = HElim h C pms (xs @ m' # ys) (PCon h k as)
  let ?lms' = xs @ m' # ys
  have lr: list-reds S lms ?lms'
    using lms list-red-imp-list-reds[
      OF list-red-append[OF mm', of xs ys]]
```

```
    by simp
  have kp: k < length pms using reg k by simp
  have left: reds S
      (apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k) as))
      (apps (pms ! k)
        (elimArgs h C pms ?lms' (pcons (S h) ! k) as))
    by (rule point-contractum-reds[OF kp])
      (use lr in simp-all)
  have right-step: red S
      (HElim h C pms ?lms' (PCon h k as))
      (apps (pms ! k)
        (elimArgs h C pms ?lms' (pcons (S h) ! k) as))
    by (rule red.elimPoint[
          where S = S and h = h and k = k and C = C
            and pms = pms and lms = ?lms' and as = as,
          OF k])
  have right: reds S v
      (apps (pms ! k)
        (elimArgs h C pms ?lms' (pcons (S h) ! k) as))
    using v red-as-reds[OF right-step] by simp
  show ?thesis by (rule joinableI[OF left right])
next
  fix t'
  assume step: red S (PCon h k as) t'
    and v: v = HElim h C pms lms t'
  obtain as' where t': t' = PCon h k as'
    and lr0: list-red S as as'
    by (rule red-PCon-list[OF step])
  have lr: list-reds S as as'
    by (rule list-red-imp-list-reds[OF lr0])
  have kp: k < length pms using reg k by simp
  have left: reds S
      (apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k) as))
      (apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k) as'))
    by (rule point-contractum-reds[OF kp])
      (use lr in simp-all)
  have right-step: red S
      (HElim h C pms lms (PCon h k as'))
      (apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k) as'))
    by (rule red.elimPoint[
          where S = S and h = h and k = k and C = C
            and pms = pms and lms = lms and as = as',
          OF k])
  have right: reds S v
      (apps (pms ! k)
```

```
      (elimArgs h C pms lms (pcons (S h) ! k) as'))
    using v t' red-as-reds[OF right-step] by simp
  show ?thesis by (rule joinableI[OF left right])
qed

lemma path-root-peak:
  assumes reg: regular S
      (HElim h C pms lms (LCon h k as (DVar j)))
    and k: k < length (lcons (S h))
    and other: red S
      (HElim h C pms lms (LCon h k as (DVar j))) v
  shows joinable S
    (apps (dsubst S 0 (DVar j) (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as)) v
  using other
proof (rule red-HElim-LConE)
  assume k < length (lcons (S h))
    and v: v = apps (dsubst S 0 (DVar j) (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as)
  then show ?thesis by simp
next
  fix xs m m' ys
  assume pms: pms = xs @ m # ys and mm': red S m m'
    and v: v = HElim h C (xs @ m' # ys) lms
      (LCon h k as (DVar j))
  let ?pms' = xs @ m' # ys
  have lr: list-reds S pms ?pms'
    using pms list-red-imp-list-reds[
      OF list-red-append[OF mm', of xs ys]]
    by simp
  have kl: k < length lms using reg k by simp
  have left: reds S
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as))
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C ?pms' lms (lcons (S h) ! k) as))
    by (rule path-contractum-reds[OF kl])
      (use reg lr in simp-all)
  have right-step: red S
      (HElim h C ?pms' lms (LCon h k as (DVar j)))
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C ?pms' lms (lcons (S h) ! k) as))
    by (rule red.elimPath[
          where S = S and h = h and k = k and C = C
            and pms = ?pms' and lms = lms and as = as and j = j,
          OF k])
  have right: reds S v
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C ?pms' lms (lcons (S h) ! k) as))
```

```
    using v red-as-reds[OF right-step] by simp
  show ?thesis by (rule joinableI[OF left right])
next
  fix xs m m' ys
  assume lms: lms = xs @ m # ys and mm': red S m m'
    and v: v = HElim h C pms (xs @ m' # ys)
      (LCon h k as (DVar j))
  let ?lms' = xs @ m' # ys
  have lr: list-reds S lms ?lms'
    using lms list-red-imp-list-reds[
      OF list-red-append[OF mm', of xs ys]]
    by simp
  have kl: k < length lms using reg k by simp
  have left: reds S
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as))
      (apps (dsubst S 0 (DVar j) (?lms' ! k))
        (elimArgs h C pms ?lms' (lcons (S h) ! k) as))
    by (rule path-contractum-reds[OF kl])
      (use reg lr in simp-all)
  have right-step: red S
      (HElim h C pms ?lms' (LCon h k as (DVar j)))
      (apps (dsubst S 0 (DVar j) (?lms' ! k))
        (elimArgs h C pms ?lms' (lcons (S h) ! k) as))
    by (rule red.elimPath[
          where S = S and h = h and k = k and C = C
            and pms = pms and lms = ?lms' and as = as and j = j,
          OF k])
  have right: reds S v
      (apps (dsubst S 0 (DVar j) (?lms' ! k))
        (elimArgs h C pms ?lms' (lcons (S h) ! k) as))
    using v red-as-reds[OF right-step] by simp
  show ?thesis by (rule joinableI[OF left right])
next
  fix t'
  assume step: red S (LCon h k as (DVar j)) t'
    and v: v = HElim h C pms lms t'
  obtain as' where t': t' = LCon h k as' (DVar j)
    and lr0: list-red S as as'
    by (rule red-LCon-list[OF step])
  have lr: list-reds S as as'
    by (rule list-red-imp-list-reds[OF lr0])
  have kl: k < length lms using reg k by simp
  have left: reds S
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as))
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as'))
    by (rule path-contractum-reds[OF kl])
```

```
    (use reg lr in simp-all)
  have right-step: red S
    (HElim h C pms lms (LCon h k as' (DVar j)))
    (apps (dsubst S 0 (DVar j) (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as'))
   by (rule red.elimPath[
       where S = S and h = h and k = k and C = C
        and pms = pms and lms = lms and as = as' and j = j,
       OF k])
  have right: reds S v
    (apps (dsubst S 0 (DVar j) (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as'))
   using v t' red-as-reds[OF right-step] by simp
  show ?thesis by (rule joinableI[OF left right])
qed

lemma red-HElimE [case-names point path pm lm scrut]:
  assumes red S (HElim h C pms lms t) u
  obtains
    (point) k as where t = PCon h k as
      k < length (pcons (S h))
      u = apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k) as)
  | (path) k as j where t = LCon h k as (DVar j)
      k < length (lcons (S h))
      u = apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as)
  | (pm) xs m m' ys where pms = xs @ m # ys red S m m'
      u = HElim h C (xs @ m' # ys) lms t
  | (lm) xs m m' ys where lms = xs @ m # ys red S m m'
      u = HElim h C pms (xs @ m' # ys) t
  | (scrut) t' where red S t t'
      u = HElim h C pms lms t'
  using assms by (cases rule: red.cases) auto

lemma joinable-App-left:
  assumes joinable S t u
  shows joinable S (App t a) (App u a)
proof −
  from assms obtain v where tv: reds S t v and uv: reds S u v
    unfolding joinable-def by blast
  have reds S (App t a) (App v a) by (rule reds-App[OF tv]) simp
  moreover have reds S (App u a) (App v a) by (rule reds-App[OF uv]) simp
  ultimately show ?thesis by (rule joinableI)
qed

lemma joinable-App-right:
  assumes joinable S t u
  shows joinable S (App a t) (App a u)
```

proof −
  from *assms* **obtain** *v* **where** *tv*: *reds S t v* **and** *uv*: *reds S u v*
    unfolding *joinable-def* **by** *blast*
  have *reds S* (*App a t*) (*App a v*) **by** (*rule reds-App*) (*use tv* **in** *simp-all*)
  moreover have *reds S* (*App a u*) (*App a v*)
    by (*rule reds-App*) (*use uv* **in** *simp-all*)
  ultimately show *?thesis* **by** (*rule joinableI*)
qed

lemma *joinable-Lam*:
  assumes *joinable S t u*
  shows *joinable S* (*Lam A t*) (*Lam A u*)
proof −
  from *assms* **obtain** *v* **where** *tv*: *reds S t v* **and** *uv*: *reds S u v*
    unfolding *joinable-def* **by** *blast*
  have *reds S* (*Lam A t*) (*Lam A v*) **by** (*rule reds-Lam*[*OF tv*])
  moreover have *reds S* (*Lam A u*) (*Lam A v*) **by** (*rule reds-Lam*[*OF uv*])
  ultimately show *?thesis* **by** (*rule joinableI*)
qed

lemma *red-App-local-confluence*:
  assumes *regt*: *regular S t* **and** *regu*: *regular S u*
    and *local-t*: $\bigwedge$*t1 t2*. *red S t t1* $\Longrightarrow$ *red S t t2* $\Longrightarrow$
      joinable S t1 t2
    and *local-u*: $\bigwedge$*u1 u2*. *red S u u1* $\Longrightarrow$ *red S u u2* $\Longrightarrow$
      joinable S u1 u2
    and *left*: *red S* (*App t u*) *x*
    and *right*: *red S* (*App t u*) *y*
  shows *joinable S x y*
proof (*rule red-AppE*[*OF left*])
  fix *A body*
  assume *t*: *t* = *Lam A body* **and** *close*: *dclosed0 S u*
    and *x*: *x* = *subst 0 u body*
  have *regbody*: *regular S body* **using** *regt t* **by** *simp*
  show *?thesis*
  proof (*rule red-AppE*[*OF right*])
    fix *B body′*
    assume *t′*: *t* = *Lam B body′* **and** *dclosed0 S u*
      and *y*: *y* = *subst 0 u body′*
    from *t t′ x y* **show** *?thesis* **by** *simp*
  next
    fix *f′*
    assume *tf′*: *red S t f′* **and** *y*: *y* = *App f′ u*
    from *tf′ t* **obtain** *body′* **where** *f′*: *f′* = *Lam A body′*
      and *step*: *red S body body′*
      by (*auto elim*: *red-LamE*)
    have *joinable S* (*subst 0 u body*) (*App* (*Lam A body′*) *u*)
      by (*rule beta-body-peak*[*OF step regbody regu close*])
    with *x y f′* **show** *?thesis* **by** *simp*

```
  next
    fix u'
    assume uu': red S u u' and y: y = App t u'
    have joinable S (subst 0 u body) (App (Lam A body) u')
      by (rule beta-argument-peak[OF uu' regu close])
    with t x y show ?thesis by simp
  qed
next
  fix f'
  assume tf': red S t f' and x: x = App f' u
  show ?thesis
  proof (rule red-AppE[OF right])
    fix A body
    assume t: t = Lam A body and close: dclosed0 S u
      and y: y = subst 0 u body
    from tf' t obtain body' where f': f' = Lam A body'
      and step: red S body body'
      by (auto elim: red-LamE)
    have regbody: regular S body using regt t by simp
    have joinable S (subst 0 u body) (App (Lam A body') u)
      by (rule beta-body-peak[OF step regbody regu close])
    with x y f' show ?thesis by (simp add: joinable-sym)
  next
    fix f''
    assume tf'': red S t f'' and y: y = App f'' u
    have joinable S f' f'' by (rule local-t[OF tf' tf''])
    then have joinable S (App f' u) (App f'' u)
      by (rule joinable-App-left)
    with x y show ?thesis by simp
  next
    fix u'
    assume uu': red S u u' and y: y = App t u'
    have first: reds S (App f' u) (App f' u')
      by (rule red-as-reds[OF red.appR[OF uu']])
    have second: reds S (App t u') (App f' u')
      by (rule red-as-reds[OF red.appL[OF tf']])
    have joinable S (App f' u) (App t u')
      by (rule joinableI[OF first second])
    with x y show ?thesis by simp
  qed
next
  fix u'
  assume uu': red S u u' and x: x = App t u'
  show ?thesis
  proof (rule red-AppE[OF right])
    fix A body
    assume t: t = Lam A body and close: dclosed0 S u
      and y: y = subst 0 u body
    have joinable S (subst 0 u body) (App (Lam A body) u')
```

```
    by (rule beta-argument-peak[OF uu' regu close])
   with t x y show ?thesis by (simp add: joinable-sym)
  next
   fix f'
   assume tf': red S t f' and y: y = App f' u
   have first: reds S (App t u') (App f' u')
    by (rule red-as-reds[OF red.appL[OF tf']])
   have second: reds S (App f' u) (App f' u')
    by (rule red-as-reds[OF red.appR[OF uu']])
   have joinable S (App t u') (App f' u)
    by (rule joinableI[OF first second])
   with x y show ?thesis by simp
  next
   fix u''
   assume uu'': red S u u'' and y: y = App t u''
   have joinable S u' u'' by (rule local-u[OF uu' uu''])
   then have joinable S (App t u') (App t u'')
    by (rule joinable-App-right)
   with x y show ?thesis by simp
  qed
qed

lemma red-HElim-local-confluence:
   assumes reg: regular S (HElim h C pms lms t)
    and local-pms: ⋀m u v. m ∈ set pms ⟹
     red S m u ⟹ red S m v ⟹ joinable S u v
    and local-lms: ⋀m u v. m ∈ set lms ⟹
     red S m u ⟹ red S m v ⟹ joinable S u v
    and local-t: ⋀u v. red S t u ⟹ red S t v ⟹
     joinable S u v
    and left: red S (HElim h C pms lms t) x
    and right: red S (HElim h C pms lms t) y
   shows joinable S x y
  proof (rule red-HElimE[OF left])
   fix k as
   assume t: t = PCon h k as and k: k < length (pcons (S h))
    and x: x = apps (pms ! k)
     (elimArgs h C pms lms (pcons (S h) ! k) as)
   have regroot: regular S (HElim h C pms lms (PCon h k as))
    using reg t by simp
   have other: red S (HElim h C pms lms (PCon h k as)) y
    using right t by simp
   have joinable S
     (apps (pms ! k)
      (elimArgs h C pms lms (pcons (S h) ! k) as)) y
    by (rule point-root-peak[OF regroot k other])
   with x show ?thesis by simp
  next
   fix k as j
```

```
  assume t: t = LCon h k as (DVar j)
    and k: k < length (lcons (S h))
    and x: x = apps (dsubst S 0 (DVar j) (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as)
  have regroot: regular S
      (HElim h C pms lms (LCon h k as (DVar j)))
    using reg t by simp
  have other: red S
      (HElim h C pms lms (LCon h k as (DVar j))) y
    using right t by simp
  have joinable S
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as)) y
    by (rule path-root-peak[OF regroot k other])
  with x show ?thesis by simp
next
  fix xs m m' ys
  assume pms: pms = xs @ m # ys and mm': red S m m'
    and x: x = HElim h C (xs @ m' # ys) lms t
  let ?pms1 = xs @ m' # ys
  have lp1: list-red S pms ?pms1
    using pms list-red-append[OF mm', of xs ys] by simp
  have rp1: list-reds S pms ?pms1
    by (rule list-red-imp-list-reds[OF lp1])
  show ?thesis
  proof (rule red-HElimE[OF right])
    fix k as
    assume t': t = PCon h k as
      and k: k < length (pcons (S h))
      and y: y = apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k) as)
    have regroot: regular S (HElim h C pms lms (PCon h k as))
      using reg t' by simp
    have other: red S (HElim h C pms lms (PCon h k as)) x
      using left t' by simp
    have peak: joinable S
        (apps (pms ! k)
          (elimArgs h C pms lms (pcons (S h) ! k) as)) x
      by (rule point-root-peak[OF regroot k other])
    from joinable-sym[OF peak] x y show ?thesis by simp
  next
    fix k as j
    assume t': t = LCon h k as (DVar j)
      and k: k < length (lcons (S h))
      and y: y = apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as)
    have regroot: regular S
        (HElim h C pms lms (LCon h k as (DVar j)))
      using reg t' by simp
```

```
  have other: red S
      (HElim h C pms lms (LCon h k as (DVar j))) x
    using left t' by simp
  have peak: joinable S
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as)) x
    by (rule path-root-peak[OF regroot k other])
  from joinable-sym[OF peak] x y show ?thesis by simp
next
  fix us n n' vs
  assume pms2: pms = us @ n # vs and nn': red S n n'
    and y: y = HElim h C (us @ n' # vs) lms t
  let ?pms2 = us @ n' # vs
  have lp2: list-red S pms ?pms2
    using pms2 list-red-append[OF nn', of us vs] by simp
  have ∃ ws. list-reds S ?pms1 ws ∧ list-reds S ?pms2 ws
    by (rule list-red-local-confluence[
          OF local-pms lp1 lp2])
  then obtain ws where one: list-reds S ?pms1 ws
    and two: list-reds S ?pms2 ws
    by blast
  have left-path: reds S
      (HElim h C ?pms1 lms t) (HElim h C ws lms t)
    by (rule reds-HElim[OF one]) simp-all
  have right-path: reds S
      (HElim h C ?pms2 lms t) (HElim h C ws lms t)
    by (rule reds-HElim[OF two]) simp-all
  have joinable S
      (HElim h C ?pms1 lms t) (HElim h C ?pms2 lms t)
    by (rule joinableI[OF left-path right-path])
  with x y show ?thesis by simp
next
  fix us n n' vs
  assume lms2: lms = us @ n # vs and nn': red S n n'
    and y: y = HElim h C pms (us @ n' # vs) t
  let ?lms2 = us @ n' # vs
  have rl2: list-reds S lms ?lms2
    using lms2 list-red-imp-list-reds[
      OF list-red-append[OF nn', of us vs]]
    by simp
  have left-path: reds S
      (HElim h C ?pms1 lms t) (HElim h C ?pms1 ?lms2 t)
    by (rule reds-HElim) (use rl2 in simp-all)
  have right-path: reds S
      (HElim h C pms ?lms2 t) (HElim h C ?pms1 ?lms2 t)
    by (rule reds-HElim) (use rp1 in simp-all)
  have joinable S
      (HElim h C ?pms1 lms t) (HElim h C pms ?lms2 t)
    by (rule joinableI[OF left-path right-path])
```

```
    with x y show ?thesis by simp
  next
    fix t'
    assume tt': red S t t' and y: y = HElim h C pms lms t'
    have left-path: reds S
        (HElim h C ?pms1 lms t) (HElim h C ?pms1 lms t')
      by (rule reds-HElim-scrutinee[OF red-as-reds[OF tt']])
    have right-path: reds S
        (HElim h C pms lms t') (HElim h C ?pms1 lms t')
      by (rule reds-HElim) (use rp1 in simp-all)
    have joinable S
        (HElim h C ?pms1 lms t) (HElim h C pms lms t')
      by (rule joinableI[OF left-path right-path])
    with x y show ?thesis by simp
  qed

next
  fix xs m m' ys
  assume lms: lms = xs @ m # ys and mm': red S m m'
    and x: x = HElim h C pms (xs @ m' # ys) t
  let ?lms1 = xs @ m' # ys
  have ll1: list-red S lms ?lms1
    using lms list-red-append[OF mm', of xs ys] by simp
  have rl1: list-reds S lms ?lms1
    by (rule list-red-imp-list-reds[OF ll1])
  show ?thesis
  proof (rule red-HElimE[OF right])
    fix k as
    assume t': t = PCon h k as
      and k: k < length (pcons (S h))
      and y: y = apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k) as)
    have regroot: regular S (HElim h C pms lms (PCon h k as))
      using reg t' by simp
    have other: red S (HElim h C pms lms (PCon h k as)) x
      using left t' by simp
    have peak: joinable S
        (apps (pms ! k)
          (elimArgs h C pms lms (pcons (S h) ! k) as)) x
      by (rule point-root-peak[OF regroot k other])
    from joinable-sym[OF peak] x y show ?thesis by simp
  next
    fix k as j
    assume t': t = LCon h k as (DVar j)
      and k: k < length (lcons (S h))
      and y: y = apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as)
    have regroot: regular S
        (HElim h C pms lms (LCon h k as (DVar j)))
```

```
    using reg t' by simp
  have other: red S
      (HElim h C pms lms (LCon h k as (DVar j))) x
    using left t' by simp
  have peak: joinable S
      (apps (dsubst S 0 (DVar j) (lms ! k))
        (elimArgs h C pms lms (lcons (S h) ! k) as)) x
    by (rule path-root-peak[OF regroot k other])
  from joinable-sym[OF peak] x y show ?thesis by simp
next
  fix us n n' vs
  assume pms2: pms = us @ n # vs and nn': red S n n'
    and y: y = HElim h C (us @ n' # vs) lms t
  let ?pms2 = us @ n' # vs
  have rp2: list-reds S pms ?pms2
    using pms2 list-red-imp-list-reds[
      OF list-red-append[OF nn', of us vs]]
    by simp
  have left-path: reds S
      (HElim h C pms ?lms1 t) (HElim h C ?pms2 ?lms1 t)
    by (rule reds-HElim) (use rp2 in simp-all)
  have right-path: reds S
      (HElim h C ?pms2 lms t) (HElim h C ?pms2 ?lms1 t)
    by (rule reds-HElim) (use rl1 in simp-all)
  have joinable S
      (HElim h C pms ?lms1 t) (HElim h C ?pms2 lms t)
    by (rule joinableI[OF left-path right-path])
  with x y show ?thesis by simp
next
  fix us n n' vs
  assume lms2: lms = us @ n # vs and nn': red S n n'
    and y: y = HElim h C pms (us @ n' # vs) t
  let ?lms2 = us @ n' # vs
  have ll2: list-red S lms ?lms2
    using lms2 list-red-append[OF nn', of us vs] by simp
  have ∃ ws. list-reds S ?lms1 ws ∧ list-reds S ?lms2 ws
    by (rule list-red-local-confluence[
          OF local-lms ll1 ll2])
  then obtain ws where one: list-reds S ?lms1 ws
    and two: list-reds S ?lms2 ws
    by blast
  have left-path: reds S
      (HElim h C pms ?lms1 t) (HElim h C pms ws t)
    by (rule reds-HElim) (use one in simp-all)
  have right-path: reds S
      (HElim h C pms ?lms2 t) (HElim h C pms ws t)
    by (rule reds-HElim) (use two in simp-all)
  have joinable S
      (HElim h C pms ?lms1 t) (HElim h C pms ?lms2 t)
```

```
    by (rule joinableI[OF left-path right-path])
   with x y show ?thesis by simp
 next
   fix t'
   assume tt': red S t t' and y: y = HElim h C pms lms t'
   have left-path: reds S
      (HElim h C pms ?lms1 t) (HElim h C pms ?lms1 t')
    by (rule reds-HElim-scrutinee[OF red-as-reds[OF tt']])
   have right-path: reds S
      (HElim h C pms lms t') (HElim h C pms ?lms1 t')
    by (rule reds-HElim) (use rl1 in simp-all)
   have joinable S
      (HElim h C pms ?lms1 t) (HElim h C pms lms t')
    by (rule joinableI[OF left-path right-path])
   with x y show ?thesis by simp
 qed
next
 fix t'
 assume tt': red S t t' and x: x = HElim h C pms lms t'
 show ?thesis
 proof (rule red-HElimE[OF right])
   fix k as
   assume t0: t = PCon h k as
    and k: k < length (pcons (S h))
    and y: y = apps (pms ! k)
      (elimArgs h C pms lms (pcons (S h) ! k) as)
   have regroot: regular S (HElim h C pms lms (PCon h k as))
    using reg t0 by simp
   have other: red S (HElim h C pms lms (PCon h k as)) x
    using left t0 by simp
   have peak: joinable S
      (apps (pms ! k)
        (elimArgs h C pms lms (pcons (S h) ! k) as)) x
    by (rule point-root-peak[OF regroot k other])
   from joinable-sym[OF peak] x y show ?thesis by simp
 next
   fix k as j
   assume t0: t = LCon h k as (DVar j)
    and k: k < length (lcons (S h))
    and y: y = apps (dsubst S 0 (DVar j) (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as)
   have regroot: regular S
      (HElim h C pms lms (LCon h k as (DVar j)))
    using reg t0 by simp
   have other: red S
      (HElim h C pms lms (LCon h k as (DVar j))) x
    using left t0 by simp
   have peak: joinable S
      (apps (dsubst S 0 (DVar j) (lms ! k))
```

```
      (elimArgs h C pms lms (lcons (S h) ! k) as)) x
   by (rule path-root-peak[OF regroot k other])
 from joinable-sym[OF peak] x y show ?thesis by simp
next
 fix xs m m' ys
 assume pms2: pms = xs @ m # ys and mm': red S m m'
   and y: y = HElim h C (xs @ m' # ys) lms t
 let ?pms2 = xs @ m' # ys
 have rp2: list-reds S pms ?pms2
   using pms2 list-red-imp-list-reds[
     OF list-red-append[OF mm', of xs ys]]
   by simp
 have left-path: reds S
     (HElim h C pms lms t') (HElim h C ?pms2 lms t')
   by (rule reds-HElim) (use rp2 in simp-all)
 have right-path: reds S
     (HElim h C ?pms2 lms t) (HElim h C ?pms2 lms t')
   by (rule reds-HElim-scrutinee[OF red-as-reds[OF tt']])
 have joinable S
     (HElim h C pms lms t') (HElim h C ?pms2 lms t)
   by (rule joinableI[OF left-path right-path])
 with x y show ?thesis by simp
next
 fix xs m m' ys
 assume lms2: lms = xs @ m # ys and mm': red S m m'
   and y: y = HElim h C pms (xs @ m' # ys) t
 let ?lms2 = xs @ m' # ys
 have rl2: list-reds S lms ?lms2
   using lms2 list-red-imp-list-reds[
     OF list-red-append[OF mm', of xs ys]]
   by simp
 have left-path: reds S
     (HElim h C pms lms t') (HElim h C pms ?lms2 t')
   by (rule reds-HElim) (use rl2 in simp-all)
 have right-path: reds S
     (HElim h C pms ?lms2 t) (HElim h C pms ?lms2 t')
   by (rule reds-HElim-scrutinee[OF red-as-reds[OF tt']])
 have joinable S
     (HElim h C pms lms t') (HElim h C pms ?lms2 t)
   by (rule joinableI[OF left-path right-path])
 with x y show ?thesis by simp
next
 fix t''
 assume tt'': red S t t'' and y: y = HElim h C pms lms t''
 from local-t[OF tt' tt''] obtain w where tw: reds S t' w
   and ttw: reds S t'' w
   unfolding joinable-def by blast
 have left-path: reds S
     (HElim h C pms lms t') (HElim h C pms lms w)
```

```
      by (rule reds-HElim) (use tw in simp-all)
    have right-path: reds S
      (HElim h C pms lms t'') (HElim h C pms lms w)
      by (rule reds-HElim) (use ttw in simp-all)
    have joinable S
      (HElim h C pms lms t') (HElim h C pms lms t'')
      by (rule joinableI[OF left-path right-path])
    with x y show ?thesis by simp
  qed
qed

theorem red-local-confluence:
  assumes reg: regular S t
    and left: red S t u and right: red S t v
  shows joinable S u v
  using assms
proof (induct t arbitrary: u v)
  case (Var i)
  from Var.prems(2) show ?case
    by (cases rule: red.cases)
next
  case (Lam A t)
  from Lam.prems(2) obtain t1 where u: u = Lam A t1
    and one: red S t t1
    by (rule red-LamE)
  from Lam.prems(3) obtain t2 where v: v = Lam A t2
    and two: red S t t2
    by (rule red-LamE)
  have jt: joinable S t1 t2
    by (rule Lam.hyps[OF - one two])
      (use Lam.prems in simp-all)
  from joinable-Lam[OF jt, of A] u v show ?case by simp
next
  case (App t w)
  have regt: regular S t and regw: regular S w
    using App.prems by simp-all
  have local-t: ⋀x y. red S t x ⟹ red S t y ⟹
      joinable S x y
    by (rule App.hyps(1)[OF regt])
  have local-w: ⋀x y. red S w x ⟹ red S w y ⟹
      joinable S x y
    by (rule App.hyps(2)[OF regw])
  show ?case
    by (rule red-App-local-confluence[
        OF regt regw local-t local-w App.prems(2,3)])
next
  case (PCon h k as)
  from PCon.prems(2) obtain xs where u: u = PCon h k xs
    and one: list-red S as xs
```

```
    by (rule red-PCon-list)
  from PCon.prems(3) obtain ys where v: v = PCon h k ys
    and two: list-red S as ys
    by (rule red-PCon-list)
  have local: ⋀a b c. a ∈ set as ⟹
      red S a b ⟹ red S a c ⟹ joinable S b c
  proof −
    fix a b c
    assume mem: a ∈ set as and ab: red S a b and ac: red S a c
    have rega: regular S a
      using PCon.prems(1) mem by (auto simp: list-all-iff)
    show joinable S b c
      by (rule PCon.hyps[OF mem rega ab ac])
  qed
  have ∃ zs. list-reds S xs zs ∧ list-reds S ys zs
    by (rule list-red-local-confluence[OF local one two])
  then obtain zs where xz: list-reds S xs zs
    and yz: list-reds S ys zs by blast
  have left-path: reds S (PCon h k xs) (PCon h k zs)
    by (rule reds-PCon[OF xz])
  have right-path: reds S (PCon h k ys) (PCon h k zs)
    by (rule reds-PCon[OF yz])
  have joinable S (PCon h k xs) (PCon h k ys)
    by (rule joinableI[OF left-path right-path])
  with u v show ?case by simp
next
  case (LCon h k as d)
  from LCon.prems(2) obtain xs where u: u = LCon h k xs d
    and one: list-red S as xs
    by (rule red-LCon-list)
  from LCon.prems(3) obtain ys where v: v = LCon h k ys d
    and two: list-red S as ys
    by (rule red-LCon-list)
  have local: ⋀a b c. a ∈ set as ⟹
      red S a b ⟹ red S a c ⟹ joinable S b c
  proof −
    fix a b c
    assume mem: a ∈ set as and ab: red S a b and ac: red S a c
    have rega: regular S a
      using LCon.prems(1) mem by (auto simp: list-all-iff)
    show joinable S b c
      by (rule LCon.hyps[OF mem rega ab ac])
  qed
  have ∃ zs. list-reds S xs zs ∧ list-reds S ys zs
    by (rule list-red-local-confluence[OF local one two])
  then obtain zs where xz: list-reds S xs zs
    and yz: list-reds S ys zs by blast
  have left-path: reds S (LCon h k xs d) (LCon h k zs d)
    by (rule reds-LCon[OF xz])
```

```
  have right-path: reds S (LCon h k ys d) (LCon h k zs d)
    by (rule reds-LCon[OF yz])
  have joinable S (LCon h k xs d) (LCon h k ys d)
    by (rule joinableI[OF left-path right-path])
  with u v show ?case by simp
next
  case (HElim h C pms lms t)
  have local-pms: ⋀m x y. m ∈ set pms ⟹
      red S m x ⟹ red S m y ⟹ joinable S x y
  proof −
    fix m x y
    assume mem: m ∈ set pms and mx: red S m x and my: red S m y
    have regm: regular S m
      using HElim.prems(1) mem by (auto simp: list-all-iff)
    show joinable S x y
      by (rule HElim.hyps(1)[OF mem regm mx my])
  qed
  have local-lms: ⋀m x y. m ∈ set lms ⟹
      red S m x ⟹ red S m y ⟹ joinable S x y
  proof −
    fix m x y
    assume mem: m ∈ set lms and mx: red S m x and my: red S m y
    have regm: regular S m
      using HElim.prems(1) mem by (auto simp: list-all-iff)
    show joinable S x y
      by (rule HElim.hyps(2)[OF mem regm mx my])
  qed
  have local-t: ⋀x y. red S t x ⟹ red S t y ⟹
      joinable S x y
    by (rule HElim.hyps(3))
      (use HElim.prems in simp-all)
  show ?case
    by (rule red-HElim-local-confluence[
        OF HElim.prems(1) local-pms local-lms local-t
          HElim.prems(2,3)])
qed

lemma reds-first-step:
  assumes reds S t u
  shows t = u ∨ (∃ v. red S t v ∧ reds S v u)
  using assms
proof (induct rule: rtranclp-induct)
  case base
  show ?case by simp
next
  case (step y z)
  show ?case
  proof (cases t = y)
    case True
```

```
    have red S t z using step.hyps(2) True by simp
    moreover have reds S z z by simp
    ultimately show ?thesis by blast
  next
    case False
    from step.hyps(3) False obtain v where tv: red S t v
      and vy: reds S v y by blast
    have reds S v z
      by (rule rtranclp.rtrancl-into-rtrancl[OF vy step.hyps(2)])
    with tv show ?thesis by blast
  qed
qed

lemma nform-reds-eq:
  assumes normal: nform S t and steps: reds S t u
  shows u = t
proof −
  from reds-first-step[OF steps] show ?thesis
  proof
    assume t = u
    then show ?thesis by simp
  next
    assume ∃ v. red S t v ∧ reds S v u
    then obtain v where red S t v by blast
    with normal show ?thesis unfolding nform-def by blast
  qed
qed

theorem SN-unique-normal-form:
  assumes sn: SN S t and reg: regular S t
    and tu: reds S t u and un: nform S u
    and tv: reds S t v and vn: nform S v
  shows u = v
  using sn reg tu un tv vn
proof (induct arbitrary: u v rule: SN.induct)
  case (SNI t)
  show ?case
  proof (cases t = u)
    case True
    have v = t by (rule nform-reds-eq[OF - SNI.prems(4)])
      (use SNI.prems True in simp)
    with True show ?thesis by simp
  next
    case not-u: False
    from reds-first-step[OF SNI.prems(2)] not-u obtain t1 where
      step1: red S t t1 and path1: reds S t1 u
      by blast
    show ?thesis
    proof (cases t = v)
```

```
    case True
    have u = t by (rule nform-reds-eq[OF - SNI.prems(2)])
      (use SNI.prems True in simp)
    with True show ?thesis by simp
  next
    case not-v: False
    from reds-first-step[OF SNI.prems(4)] not-v obtain t2 where
      step2: red S t t2 and path2: reds S t2 v
      by blast
    have peak: joinable S t1 t2
      by (rule red-local-confluence[OF SNI.prems(1) step1 step2])
    then obtain z where one: reds S t1 z and two: reds S t2 z
      unfolding joinable-def by blast
    have sn1: SN S t1 by (rule SNI.hyps(1)[OF step1])
    have snz: SN S z by (rule SN-reds[OF one sn1])
    from SN-reaches-normal-form[OF snz] obtain n where
      zn: reds S z n and nn: nform S n by blast
    have reg1: regular S t1
      by (rule regular-red[OF step1 SNI.prems(1)])
    have reg2: regular S t2
      by (rule regular-red[OF step2 SNI.prems(1)])
    have t1n: reds S t1 n
      by (rule rtranclp-trans[OF one zn])
    have t2n: reds S t2 n
      by (rule rtranclp-trans[OF two zn])
    have un-eq: u = n
      by (rule SNI.hyps(2)[OF step1 reg1 path1
          SNI.prems(3) t1n nn])
    have vn-eq: v = n
      by (rule SNI.hyps(2)[OF step2 reg2 path2
          SNI.prems(5) t2n nn])
    from un-eq vn-eq show ?thesis by simp
  qed
 qed
qed

theorem SN-church-rosser:
 assumes sn: SN S t and reg: regular S t
   and tu: reds S t u and tv: reds S t v
 shows ∃ w. reds S u w ∧ reds S v w
proof −
 have snu: SN S u by (rule SN-reds[OF tu sn])
 have snv: SN S v by (rule SN-reds[OF tv sn])
 from SN-reaches-normal-form[OF snu] obtain un where
   uun: reds S u un and unn: nform S un by blast
 from SN-reaches-normal-form[OF snv] obtain vn where
   vvn: reds S v vn and vnn: nform S vn by blast
 have tun: reds S t un by (rule rtranclp-trans[OF tu uun])
 have tvn: reds S t vn by (rule rtranclp-trans[OF tv vvn])
```

```
  have un = vn
    by (rule SN-unique-normal-form[OF sn reg tun unn tvn vnn])
  with uun vvn show ?thesis by blast
qed

theorem wt-church-rosser:
  assumes typed: wt S D G t A
    and reds S t u and reds S t v
  shows ∃ w. reds S u w ∧ reds S v w
  by (rule SN-church-rosser[
        OF wt-strong-normalization[OF typed]
          regular-wt[OF typed] assms(2,3)])

theorem church-rosser:
  assumes typed: cwt S D G t A
    and reds S t u and reds S t v
  shows ∃ w. reds S u w ∧ reds S v w
  by (rule wt-church-rosser[OF cwt-wt[OF typed] assms(2,3)])

corollary wt-unique-normal-form:
  assumes typed: wt S D G t A
    and reds S t u and nform S u
    and reds S t v and nform S v
  shows u = v
  by (rule SN-unique-normal-form[
        OF wt-strong-normalization[OF typed]
          regular-wt[OF typed] assms(2,3,4,5)])

corollary unique-normal-form:
  assumes typed: cwt S D G t A
    and reds S t u and nform S u
    and reds S t v and nform S v
  shows u = v
  by (rule wt-unique-normal-form[
        OF cwt-wt[OF typed] assms(2,3,4,5)])

end
theory HIT-Canonical-Normal-Forms
  imports HIT-Confluence
begin
```

18 Canonical normal-form comparison

Strong normalization gives existence of a normal form and confluence gives uniqueness. We package that mathematical result as a canonical function. The definition uses Hilbert choice and is therefore not an executable normalizer. Its purpose is to replace semantic joinability obligations by equality of unique normal forms at the theorem interface.

definition *canonical-nf* :: *sigenv* ⇒ *tm* ⇒ *tm* **where**
canonical-nf S t =
(*SOME u*. *reds S t u* ∧ *nform S u*)

theorem *canonical-nf-spec*:
assumes *typed*: *wt S D G t A*
shows *reds S t* (*canonical-nf S t*) ∧
nform S (*canonical-nf S t*)
proof −
from *SN-reaches-normal-form*[*OF wt-strong-normalization*[*OF typed*]]
have ∃ *u*. *reds S t u* ∧ *nform S u* **.**
then show *?thesis*
unfolding *canonical-nf-def* **by** (*rule someI-ex*)
qed

corollary *canonical-nf-reachable*:
wt S D G t A ⟹ *reds S t* (*canonical-nf S t*)
using *canonical-nf-spec* **by** *blast*

corollary *canonical-nf-normal*:
wt S D G t A ⟹ *nform S* (*canonical-nf S t*)
using *canonical-nf-spec* **by** *blast*

theorem *canonical-nf-unique*:
assumes *typed*: *wt S D G t A*
and *steps*: *reds S t u*
and *normal*: *nform S u*
shows *canonical-nf S t* = *u*
by (*rule wt-unique-normal-form*[
OF typed canonical-nf-reachable[*OF typed*]
canonical-nf-normal[*OF typed*] *steps normal*])

corollary *canonical-nf-of-normal*:
assumes *typed*: *wt S D G t A* **and** *normal*: *nform S t*
shows *canonical-nf S t* = *t*
by (*rule canonical-nf-unique*[*OF typed - normal*]) *simp*

lemma *regular-reds*:
assumes *reds S t u* **and** *regular S t*
shows *regular S u*
using *assms*
by (*induct rule*: *rtranclp-induct*) (*auto intro*: *regular-red*)

theorem *typed-joinable-iff-canonical-nf-eq*:
assumes *typed-t*: *wt S D G t A*
and *typed-u*: *wt S E H u B*
shows *joinable S t u* ⟷
canonical-nf S t = *canonical-nf S u*
proof

```
  assume join: joinable S t u
  then obtain v where tv: reds S t v and uv: reds S u v
    unfolding joinable-def by blast
  have tnf: reds S t (canonical-nf S t)
    by (rule canonical-nf-reachable[OF typed-t])
  have unf: reds S u (canonical-nf S u)
    by (rule canonical-nf-reachable[OF typed-u])
  have nt: nform S (canonical-nf S t)
    by (rule canonical-nf-normal[OF typed-t])
  have nu: nform S (canonical-nf S u)
    by (rule canonical-nf-normal[OF typed-u])
  from wt-church-rosser[OF typed-t tv tnf]
  obtain x where vx: reds S v x
    and ntx: reds S (canonical-nf S t) x by blast
  have x-t: x = canonical-nf S t
    by (rule nform-reds-eq[OF nt ntx])
  have vnt: reds S v (canonical-nf S t)
    using vx x-t by simp
  from wt-church-rosser[OF typed-u uv unf]
  obtain y where vy: reds S v y
    and nuy: reds S (canonical-nf S u) y by blast
  have y-u: y = canonical-nf S u
    by (rule nform-reds-eq[OF nu nuy])
  have vnu: reds S v (canonical-nf S u)
    using vy y-u by simp
  have snv: SN S v
    by (rule SN-reds[OF tv wt-strong-normalization[OF typed-t]])
  have regv: regular S v
    by (rule regular-reds[OF tv regular-wt[OF typed-t]])
  show canonical-nf S t = canonical-nf S u
    by (rule SN-unique-normal-form[OF snv regv vnt nt vnu nu])
next
  assume eq: canonical-nf S t = canonical-nf S u
  have reds S t (canonical-nf S t)
    by (rule canonical-nf-reachable[OF typed-t])
  moreover have reds S u (canonical-nf S t)
    using canonical-nf-reachable[OF typed-u] eq by simp
  ultimately show joinable S t u
    unfolding joinable-def by blast
qed

end
theory HIT-Open-Boundary-Check
  imports HIT-Canonical-Normal-Forms
begin
```

19 Finite open-term boundary checking

The semantic predicate *typed-coherent-methods* ranges over every dimension context, term context, and typed constructor instance. Its boundary obligation can instead be discharged once per path constructor. We append fresh variables for that constructor's arguments to the ambient term context and check the two open faces there. Simultaneous substitution then transports each open check to every typed instance.

The resulting predicate is a finite family of typing and joinability obligations. A second formulation compares the canonical normal forms of each pair; the two formulations are equivalent for the typed open terms. The canonical normalizer uses Hilbert choice, so neither formulation is presented as executable code. The theorem below establishes the soundness bridge from the finite open obligations to the extensional safety predicate used by the rest of the development.

definition *open-args* :: *ty list* ⇒ *argspec list* ⇒ *tm list* **where**
  open-args G ss =
    map (λ*i*. *Var* (*length G* + *i*)) [*0*..<*length ss*]

definition *open-inst* :: *ty list* ⇒ *tm list* ⇒ *nat* ⇒ *tm* **where**
  open-inst G as i =
    (*if i* < *length G then Var i*
     else if i − *length G* < *length as then as* ! (*i* − *length G*)
     else Var i)

definition *open-face* ::
  sigenv ⇒ *ty list* ⇒ *nat* ⇒ *ty* ⇒
    tm list ⇒ *tm list* ⇒ *nat* ⇒ *dim* ⇒ *tm*
where
  open-face S G h C pms lms k d =
    apps (*dsubst S 0 d* (*lms* ! *k*))
      (*elimArgs h C pms lms* (*lcons* (*S h*) ! *k*)
        (*open-args G* (*lcons* (*S h*) ! *k*)))

definition *open-endpoint* ::
  sigenv ⇒ *ty list* ⇒ *nat* ⇒ *ty* ⇒
    tm list ⇒ *tm list* ⇒ *nat* ⇒ *endpt* ⇒ *tm*
where
  open-endpoint S G h C pms lms k e =
    elimEnd S h C pms lms (*open-args G* (*lcons* (*S h*) ! *k*)) *e*

definition *boundary-check-base* ::
  sigenv ⇒ *nat set* ⇒ *ty list* ⇒ *nat* ⇒ *ty* ⇒
    tm list ⇒ *tm list* ⇒ *bool*
where
  boundary-check-base S D G h C pms lms ⟷
    wf-sigenv S ∧

```
    0 ∉ D ∧
    length pms = length (pcons (S h)) ∧
    length lms = length (lcons (S h)) ∧
    (∀ k < length pms.
      wt S D G (pms ! k) (methodTy h C (pcons (S h) ! k))) ∧
    (∀ k < length lms.
      wt S (insert 0 D) G (lms ! k)
        (methodTy h C (lcons (S h) ! k)))

definition open-boundary-check ::
  sigenv ⇒ nat set ⇒ ty list ⇒ nat ⇒ ty ⇒
    tm list ⇒ tm list ⇒ bool
where
  open-boundary-check S D G h C pms lms ⟷
    wf-sigenv S ∧
    0 ∉ D ∧
    length pms = length (pcons (S h)) ∧
    length lms = length (lcons (S h)) ∧
    (∀ k < length pms.
      wt S D G (pms ! k) (methodTy h C (pcons (S h) ! k))) ∧
    (∀ k < length lms.
      wt S (insert 0 D) G (lms ! k)
        (methodTy h C (lcons (S h) ! k))) ∧
    (∀ k < length (lcons (S h)).
      joinable S
        (open-face S G h C pms lms k D0)
        (open-endpoint S G h C pms lms k
          (fst (lends (S h) ! k))) ∧
      joinable S
        (open-face S G h C pms lms k D1)
        (open-endpoint S G h C pms lms k
          (snd (lends (S h) ! k))))

definition normal-boundary-check ::
  sigenv ⇒ nat set ⇒ ty list ⇒ nat ⇒ ty ⇒
    tm list ⇒ tm list ⇒ bool
where
  normal-boundary-check S D G h C pms lms ⟷
    boundary-check-base S D G h C pms lms ∧
    (∀ k < length (lcons (S h)).
      canonical-nf S (open-face S G h C pms lms k D0) =
        canonical-nf S
          (open-endpoint S G h C pms lms k
            (fst (lends (S h) ! k))) ∧
      canonical-nf S (open-face S G h C pms lms k D1) =
        canonical-nf S
          (open-endpoint S G h C pms lms k
            (snd (lends (S h) ! k))))
```

definition *open-context* ::
ty list ⇒ *nat* ⇒ *argspec list* ⇒ *ty list*
where
open-context G h ss = *G* @ *map* (*argTy h*) *ss*

lemma *open-args-length* [*simp*]:
length (*open-args G ss*) = *length ss*
by (*simp add*: *open-args-def*)

lemma *open-args-nth*:
assumes *i* < *length ss*
shows *open-args G ss* ! *i* = *Var* (*length G* + *i*)
using *assms* **by** (*simp add*: *open-args-def*)

lemma *open-inst-prefix*:
assumes *i* < *length G*
shows *open-inst G as i* = *Var i*
using *assms* **by** (*simp add*: *open-inst-def*)

lemma *psubst-open-args*:
assumes *length as* = *length ss*
shows *map* (*psubst* (*open-inst G as*)) (*open-args G ss*) = *as*
proof (*rule nth-equalityI*)
show *length* (*map* (*psubst* (*open-inst G as*)) (*open-args G ss*)) =
length as
using *assms* **by** *simp*
next
fix *i*
assume *i*: *i* < *length*
(*map* (*psubst* (*open-inst G as*)) (*open-args G ss*))
then have *iss*: *i* < *length ss* **by** *simp*
then have *ias*: *i* < *length as* **using** *assms* **by** *simp*
show (*map* (*psubst* (*open-inst G as*)) (*open-args G ss*)) ! *i* = *as* ! *i*
using *iss ias* **by** (*simp add*: *open-args-nth open-inst-def*)
qed

lemma *up-sub-prefix*:
assumes ∀ *i* < *length G*. *s i* = *Var i*
shows ∀ *i* < *length* (*A* # *G*). *up-sub s i* = *Var i*
using *assms*
by (*intro allI impI*) (*case-tac i*; *simp*)

theorem *wt-context-append*:
assumes *wt S D G t A*
shows *wt S D* (*G* @ *H*) *t A*
using *assms*
proof (*induct rule*: *wt.induct*)
case (*wtVar i G D*)
have *i*: *i* < *length* (*G* @ *H*) **using** *wtVar* **by** *simp*

```
  have nth: (G @ H) ! i = G ! i
    using wtVar by (simp add: nth-append)
  from wt.wtVar[OF i] nth show ?case by simp
next
  case wtLam
  then show ?case by (auto intro: wt.wtLam)
next
  case wtApp
  then show ?case by (auto intro: wt.wtApp)
next
  case wtPCon
  then show ?case by (auto intro!: wt.wtPCon)
next
  case wtLCon
  then show ?case by (auto intro!: wt.wtLCon)
next
  case wtElim
  then show ?case by (auto intro!: wt.wtElim)
qed

lemma open-args-typed:
  assumes i < length ss
  shows wt S D (open-context G h ss)
    (open-args G ss ! i) (argTy h (ss ! i))
proof −
  have index: length G + i < length (open-context G h ss)
    using assms by (simp add: open-context-def)
  have nth: open-context G h ss ! (length G + i) =
      argTy h (ss ! i)
    using assms by (simp add: open-context-def nth-append)
  have wt S D (open-context G h ss)
      (Var (length G + i))
      (open-context G h ss ! (length G + i))
    by (rule wt.wtVar[OF index])
  with nth open-args-nth[OF assms] show ?thesis by simp
qed

lemma psubst-identity-wt:
  assumes typed: wt S D G t A
    and fixed: ∀ i < length G. s i = Var i
  shows psubst s t = t
  using typed fixed
proof (induct arbitrary: s rule: wt.induct)
  case (wtVar i G D)
  then show ?case by simp
next
  case (wtLam D A G t B)
  have prefix: ∀ i < length (A # G). up-sub s i = Var i
    by (rule up-sub-prefix[OF wtLam.prems])
```

```
  from wtLam(2)[OF prefix] show ?case by simp
next
  case (wtApp D G t A B u)
  then show ?case by simp
next
  case (wtPCon k h as D G)
  have args: map (psubst s) as = as
  proof (rule map-idI)
    fix x
    assume x ∈ set as
    then obtain i where i: i < length as and x: x = as ! i
      by (auto simp: in-set-conv-nth)
    have psubst s (as ! i) = as ! i
      using wtPCon i by auto
    with x show psubst s x = x by simp
  qed
  then show ?case by simp
next
  case (wtLCon j D k h as G)
  have args: map (psubst s) as = as
  proof (rule map-idI)
    fix x
    assume x ∈ set as
    then obtain i where i: i < length as and x: x = as ! i
      by (auto simp: in-set-conv-nth)
    have psubst s (as ! i) = as ! i
      using wtLCon i by auto
    with x show psubst s x = x by simp
  qed
  then show ?case by simp
next
  case (wtElim D G t h pms lms C)
  have scrutinee: psubst s t = t
    by (rule wtElim(2)[OF wtElim.prems])
  have point-methods: map (psubst s) pms = pms
  proof (rule map-idI)
    fix x
    assume x ∈ set pms
    then obtain i where i: i < length pms and x: x = pms ! i
      by (auto simp: in-set-conv-nth)
    have psubst s (pms ! i) = pms ! i
      using wtElim i by auto
    with x show psubst s x = x by simp
  qed
  have path-methods: map (psubst s) lms = lms
  proof (rule map-idI)
    fix x
    assume x ∈ set lms
    then obtain i where i: i < length lms and x: x = lms ! i
```

```
    by (auto simp: in-set-conv-nth)
   have psubst s (lms ! i) = lms ! i
    using wtElim i by auto
   with x show psubst s x = x by simp
  qed
  show ?case using point-methods path-methods scrutinee by simp
qed

lemma open-inst-regular:
  assumes args: ∀ i < length as. ∃ A. wt S D H (as ! i) A
  shows regular S (open-inst G as i)
proof (cases i < length G)
  case True
  then show ?thesis by (simp add: open-inst-def)
next
  assume not-prefix: ¬ i < length G
  show ?thesis
  proof (cases i − length G < length as)
   assume in-args: i − length G < length as
   then obtain A where wt S D H (as ! (i − length G)) A
    using args by blast
   then show ?thesis
    using not-prefix in-args regular-wt by (simp add: open-inst-def)
  next
   assume not-args: ¬ i − length G < length as
   with not-prefix show ?thesis by (simp add: open-inst-def)
  qed
qed

lemma open-inst-dclosed0:
  assumes zero: 0 ∉ D
   and args: ∀ i < length as. ∃ A. wt S D H (as ! i) A
  shows dclosed0 S (open-inst G as i)
proof (cases i < length G)
  case True
  then show ?thesis by (simp add: open-inst-def)
next
  assume not-prefix: ¬ i < length G
  show ?thesis
  proof (cases i − length G < length as)
   assume in-args: i − length G < length as
   then obtain A where typed:
      wt S D H (as ! (i − length G)) A
    using args by blast
   have ∀ j. dsubst S 0 (DVar j) (as ! (i − length G)) =
      as ! (i − length G)
    using wt-dsubst-fresh[OF typed zero] by blast
   then show ?thesis
    using not-prefix in-args
```

```
    unfolding dclosed0-def open-inst-def by simp
  next
    assume not-args: ¬ i − length G < length as
    with not-prefix show ?thesis by (simp add: open-inst-def)
  qed
qed

lemma reds-psubst-dclosed:
  assumes steps: reds S t u
    and reg: regular S t
    and env-reg: ⋀i. regular S (s i)
    and env-closed: ⋀i. dclosed0 S (s i)
  shows reds S (psubst s t) (psubst s u)
  using steps reg
proof (induct rule: rtranclp-induct)
  case base
  show ?case by simp
next
  case (step y z)
  have yreg: regular S y
    by (rule regular-reds[OF step.hyps(1) step.prems])
  have yz: red S (psubst s y) (psubst s z)
    by (rule red-psubst-dclosed[
        OF step.hyps(2) yreg env-reg env-closed])
  from step.hyps(3)[OF step.prems] yz show ?case
    by (rule rtranclp.rtrancl-into-rtrancl)
qed

lemma joinable-psubst-dclosed:
  assumes join: joinable S t u
    and treg: regular S t
    and ureg: regular S u
    and env-reg: ⋀i. regular S (s i)
    and env-closed: ⋀i. dclosed0 S (s i)
  shows joinable S (psubst s t) (psubst s u)
proof −
  from join obtain v where tv: reds S t v and uv: reds S u v
    unfolding joinable-def by blast
  have reds S (psubst s t) (psubst s v)
    by (rule reds-psubst-dclosed[OF tv treg env-reg env-closed])
  moreover have reds S (psubst s u) (psubst s v)
    by (rule reds-psubst-dclosed[OF uv ureg env-reg env-closed])
  ultimately show ?thesis unfolding joinable-def by blast
qed

lemma psubst-instE:
  assumes endpoint: wf-endpt S h ss e A
    and len: length xs = length ss
  shows psubst s (instE h xs e) =
```

```
    instE h (map (psubst s) xs) e
  using endpoint len
proof (induct rule: wf-endpt.induct)
  case (wfEArg i)
  then show ?case by simp
next
  case (wfECon k es)
  have inst:
      map (psubst s ∘ instE h xs) es =
       map (instE h (map (psubst s) xs)) es
  proof (rule nth-equalityI)
    show length (map (psubst s ∘ instE h xs) es) =
        length (map (instE h (map (psubst s) xs)) es)
      by simp
  next
    fix i
    assume i: i < length (map (psubst s ∘ instE h xs) es)
    then have i < length es by simp
    with wfECon show (map (psubst s ∘ instE h xs) es) ! i =
        (map (instE h (map (psubst s) xs)) es) ! i
      by simp
  qed
  then show ?case by (simp add: map-map)
qed

lemma psubst-elimEnd:
  assumes endpoint: wf-endpt S h ss e A
    and len: length xs = length ss
    and plen: length pms = length (pcons (S h))
  shows psubst s (elimEnd S h C pms lms xs e) =
    elimEnd S h C (map (psubst s) pms) (map (psubst s) lms)
      (map (psubst s) xs) e
  using endpoint
proof cases
  case (wfEArg i)
  then show ?thesis using len by simp
next
  case (wfECon k es)
  have kpm: k < length pms using wfECon plen by simp
  have inst:
      map (psubst s ∘ instE h xs) es =
       map (instE h (map (psubst s) xs)) es
  proof (rule nth-equalityI)
    show length (map (psubst s ∘ instE h xs) es) =
        length (map (instE h (map (psubst s) xs)) es)
      by simp
  next
    fix i
    assume i: i < length (map (psubst s ∘ instE h xs) es)
```

```
    then have ies: i < length es by simp
    have ep: wf-endpt S h ss (es ! i)
        (argTy h (pcons (S h) ! k ! i))
      using wfECon ies by blast
    from psubst-instE[OF ep len] ies
    show (map (psubst s ∘ instE h xs) es) ! i =
        (map (instE h (map (psubst s) xs)) es) ! i
      by simp
  qed
  have mapped:
      map (psubst s)
       (elimArgs h C pms lms (pcons (S h) ! k)
         (map (instE h xs) es)) =
       elimArgs h C (map (psubst s) pms) (map (psubst s) lms)
        (pcons (S h) ! k)
        (map (instE h (map (psubst s) xs)) es)
    by (simp add: psubst-elimArgs map-map inst)
  show ?thesis
    using wfECon kpm mapped by simp
qed

lemma regular-instE:
  assumes endpoint: wf-endpt S h ss e A
    and len: length as = length ss
    and args: list-all (regular S) as
  shows regular S (instE h as e)
  using endpoint
proof (induct rule: wf-endpt.induct)
  case (wfEArg i)
  then show ?case
    using len args by (auto simp: list-all-iff)
next
  case (wfECon k es)
  have inst: list-all (regular S) (map (instE h as) es)
  proof (unfold list-all-iff, intro ballI)
    fix x
    assume x ∈ set (map (instE h as) es)
    then obtain e where eset: e ∈ set es
      and x: x = instE h as e by auto
    then obtain i where i: i < length es and e: e = es ! i
      by (auto simp: in-set-conv-nth)
    have regular S (instE h as (es ! i))
      using wfECon i by auto
    with x e show regular S x by simp
  qed
  show ?case using wfECon inst by simp
qed

lemma regular-elimEnd:
```

```
  assumes endpoint: wf-endpt S h ss e A
    and len: length as = length ss
    and args: list-all (regular S) as
    and plen: length pms = length (pcons (S h))
    and llen: length lms = length (lcons (S h))
    and pms: list-all (regular S) pms
    and lms: list-all (regular S) lms
  shows regular S (elimEnd S h C pms lms as e)
  using endpoint
proof cases
  case (wfEArg i)
  have regular S (as ! i)
    using wfEArg len args by (auto simp: list-all-iff)
  then show ?thesis using plen llen pms lms wfEArg by simp
next
  case (wfECon k es)
  have method: regular S (pms ! k)
    using wfECon plen pms by (auto simp: list-all-iff)
  have inst: list-all (regular S) (map (instE h as) es)
  proof (unfold list-all-iff, intro ballI)
    fix x
    assume x ∈ set (map (instE h as) es)
    then obtain e where eset: e ∈ set es
      and x: x = instE h as e by auto
    then obtain i where i: i < length es and e: e = es ! i
      by (auto simp: in-set-conv-nth)
    have ep: wf-endpt S h ss (es ! i)
        (argTy h (pcons (S h) ! k ! i))
      using wfECon i by blast
    have regular S (instE h as (es ! i))
      by (rule regular-instE[OF ep len args])
    with x e show regular S x by simp
  qed
  have eargs: list-all (regular S)
      (elimArgs h C pms lms (pcons (S h) ! k)
        (map (instE h as) es))
    by (rule regular-elimArgs[OF plen llen pms lms inst])
  show ?thesis
    using wfECon method eargs by (simp add: regular-apps)
qed

lemma typed-list-regular:
  assumes ∀ i < length xs. ∃ A. wt S D G (xs ! i) A
  shows list-all (regular S) xs
  unfolding list-all-iff
proof
  fix x
  assume x ∈ set xs
  then obtain i where i: i < length xs and x: x = xs ! i
```

```
    by (auto simp: in-set-conv-nth)
  from assms[rule-format, OF i] obtain A where
    wt S D G (xs ! i) A by blast
  then have regular S (xs ! i) by (rule regular-wt)
  with x show regular S x by simp
qed

lemma map-psubst-identity-wt:
  assumes typed: ∀ i < length xs. ∃ A. wt S D G (xs ! i) A
    and fixed: ∀ i < length G. s i = Var i
  shows map (psubst s) xs = xs
proof (rule map-idI)
  fix x
  assume x ∈ set xs
  then obtain i where i: i < length xs and x: x = xs ! i
    by (auto simp: in-set-conv-nth)
  from typed[rule-format, OF i] obtain A where
    wt S D G (xs ! i) A by blast
  from psubst-identity-wt[OF this fixed] x
  show psubst s x = x by simp
qed

lemma open-boundary-check-base:
  assumes open-boundary-check S D G h C pms lms
  shows boundary-check-base S D G h C pms lms
  using assms
  unfolding open-boundary-check-def boundary-check-base-def by blast

theorem open-face-typed:
  assumes base: boundary-check-base S D G h C pms lms
    and k: k < length (lcons (S h))
    and face: d = D0 ∨ d = D1
  shows wt S D (open-context G h (lcons (S h) ! k))
    (open-face S G h C pms lms k d) C
proof −
  from base have wf: wf-sigenv S
    and zero: 0 ∉ D
    and plen: length pms = length (pcons (S h))
    and llen: length lms = length (lcons (S h))
    and ptyped: ∀ i < length pms.
      wt S D G (pms ! i) (methodTy h C (pcons (S h) ! i))
    and ltyped: ∀ i < length lms.
      wt S (insert 0 D) G (lms ! i)
        (methodTy h C (lcons (S h) ! i))
    unfolding boundary-check-base-def by blast+
  let ?ss = lcons (S h) ! k
  let ?K = open-context G h ?ss
  have klm: k < length lms using k llen by simp
  have ptyped-open: ∀ i < length pms.
```

```
      wt S D ?K (pms ! i) (methodTy h C (pcons (S h) ! i))
    unfolding open-context-def
    using ptyped by (blast intro: wt-context-append)
  have ltyped-open: ∀ i < length lms.
      wt S (insert 0 D) ?K (lms ! i)
       (methodTy h C (lcons (S h) ! i))
    unfolding open-context-def
    using ltyped by (blast intro: wt-context-append)
  have args: ∀ i < length ?ss.
      wt S D ?K (open-args G ?ss ! i) (argTy h (?ss ! i))
    using open-args-typed by blast
  have recursive: ⋀a. wt S D ?K a (Hit h) ⟹
      wt S D ?K (HElim h C pms lms a) C
    by (intro wt.wtElim[OF - plen llen ptyped-open ltyped-open])
  have lm: wt S (insert 0 D) ?K (lms ! k) (methodTy h C ?ss)
    using ltyped-open klm by simp
  have method: wt S D ?K (dsubst S 0 d (lms ! k))
     (methodTy h C ?ss)
  proof (rule disjE[OF face])
    assume d: d = D0
    have wt S (dctx-subst 0 D0 (insert 0 D)) ?K
       (dsubst S 0 D0 (lms ! k)) (methodTy h C ?ss)
      by (rule wt-dimension-subst[OF lm wf])
    with zero d show ?thesis by (simp add: dctx-subst-def)
  next
    assume d: d = D1
    have wt S (dctx-subst 0 D1 (insert 0 D)) ?K
       (dsubst S 0 D1 (lms ! k)) (methodTy h C ?ss)
      by (rule wt-dimension-subst[OF lm wf])
    with zero d show ?thesis by (simp add: dctx-subst-def)
  qed
  show ?thesis
    unfolding open-face-def
    by (rule wt-apps-elimArgs[
         OF recursive open-args-length args method])
qed

theorem open-endpoint-typed:
  assumes base: boundary-check-base S D G h C pms lms
    and k: k < length (lcons (S h))
    and endpoint: wf-endpt S h (lcons (S h) ! k) e (Hit h)
  shows wt S D (open-context G h (lcons (S h) ! k))
    (open-endpoint S G h C pms lms k e) C
proof −
  from base have plen: length pms = length (pcons (S h))
    and llen: length lms = length (lcons (S h))
    and ptyped: ∀ i < length pms.
      wt S D G (pms ! i) (methodTy h C (pcons (S h) ! i))
    and ltyped: ∀ i < length lms.
```

```
      wt S (insert 0 D) G (lms ! i)
        (methodTy h C (lcons (S h) ! i))
    unfolding boundary-check-base-def by blast+
  let ?ss = lcons (S h) ! k
  let ?K = open-context G h ?ss
  have ptyped-open: ∀ i < length pms.
      wt S D ?K (pms ! i) (methodTy h C (pcons (S h) ! i))
    unfolding open-context-def
    using ptyped by (blast intro: wt-context-append)
  have ltyped-open: ∀ i < length lms.
      wt S (insert 0 D) ?K (lms ! i)
        (methodTy h C (lcons (S h) ! i))
    unfolding open-context-def
    using ltyped by (blast intro: wt-context-append)
  have args: ∀ i < length (open-args G ?ss).
      wt S D ?K (open-args G ?ss ! i) (argTy h (?ss ! i))
    using open-args-typed by simp
  show ?thesis
    unfolding open-endpoint-def
    by (rule wt-elimEnd[
          OF endpoint open-args-length args
            plen llen ptyped-open ltyped-open])
qed

lemma open-joinable-iff-canonical-nf-eq:
  assumes base: boundary-check-base S D G h C pms lms
    and k: k < length (lcons (S h))
    and endpoint: wf-endpt S h (lcons (S h) ! k) e (Hit h)
    and face: d = D0 ∨ d = D1
  shows joinable S
      (open-face S G h C pms lms k d)
      (open-endpoint S G h C pms lms k e) ⟷
    canonical-nf S (open-face S G h C pms lms k d) =
      canonical-nf S (open-endpoint S G h C pms lms k e)
  by (rule typed-joinable-iff-canonical-nf-eq[
        OF open-face-typed[OF base k face]
          open-endpoint-typed[OF base k endpoint]])

theorem normal-boundary-check-iff-open-boundary-check:
  normal-boundary-check S D G h C pms lms ⟷
    open-boundary-check S D G h C pms lms
proof
  assume normal: normal-boundary-check S D G h C pms lms
  then have base: boundary-check-base S D G h C pms lms
    and equal:
      ∀ k < length (lcons (S h)).
        canonical-nf S (open-face S G h C pms lms k D0) =
          canonical-nf S
            (open-endpoint S G h C pms lms k
```

```
        (fst (lends (S h) ! k))) ∧
    canonical-nf S (open-face S G h C pms lms k D1) =
      canonical-nf S
        (open-endpoint S G h C pms lms k
          (snd (lends (S h) ! k)))
  unfolding normal-boundary-check-def by blast+
from base have wf: wf-sigenv S
  unfolding boundary-check-base-def by blast
have sig: wf-hsig S h
  using wf unfolding wf-sigenv-def by blast
have obligations:
  ∀ k < length (lcons (S h)).
    joinable S
      (open-face S G h C pms lms k D0)
      (open-endpoint S G h C pms lms k
        (fst (lends (S h) ! k))) ∧
    joinable S
      (open-face S G h C pms lms k D1)
      (open-endpoint S G h C pms lms k
        (snd (lends (S h) ! k)))
proof (intro allI impI)
  fix k
  assume k: k < length (lcons (S h))
  have zero:
    joinable S
      (open-face S G h C pms lms k D0)
      (open-endpoint S G h C pms lms k
        (fst (lends (S h) ! k))) ⟷
    canonical-nf S (open-face S G h C pms lms k D0) =
      canonical-nf S
        (open-endpoint S G h C pms lms k
          (fst (lends (S h) ! k)))
    by (rule open-joinable-iff-canonical-nf-eq[
        OF base k wf-hsig-lhs[OF sig k]]) simp
  have one:
    joinable S
      (open-face S G h C pms lms k D1)
      (open-endpoint S G h C pms lms k
        (snd (lends (S h) ! k))) ⟷
    canonical-nf S (open-face S G h C pms lms k D1) =
      canonical-nf S
        (open-endpoint S G h C pms lms k
          (snd (lends (S h) ! k)))
    by (rule open-joinable-iff-canonical-nf-eq[
        OF base k wf-hsig-rhs[OF sig k]]) simp
  from equal[rule-format, OF k] zero one
  show joinable S
    (open-face S G h C pms lms k D0)
    (open-endpoint S G h C pms lms k
```

```
        (fst (lends (S h) ! k))) ∧
      joinable S
        (open-face S G h C pms lms k D1)
        (open-endpoint S G h C pms lms k
          (snd (lends (S h) ! k)))
      by blast
  qed
  from base obligations show
      open-boundary-check S D G h C pms lms
    unfolding boundary-check-base-def open-boundary-check-def by blast
next
  assume checked: open-boundary-check S D G h C pms lms
  have base: boundary-check-base S D G h C pms lms
    by (rule open-boundary-check-base[OF checked])
  from checked have joins:
      ∀ k < length (lcons (S h)).
        joinable S
          (open-face S G h C pms lms k D0)
          (open-endpoint S G h C pms lms k
            (fst (lends (S h) ! k))) ∧
        joinable S
          (open-face S G h C pms lms k D1)
          (open-endpoint S G h C pms lms k
            (snd (lends (S h) ! k)))
    unfolding open-boundary-check-def by blast
  from base have wf: wf-sigenv S
    unfolding boundary-check-base-def by blast
  have sig: wf-hsig S h
    using wf unfolding wf-sigenv-def by blast
  have equal:
      ∀ k < length (lcons (S h)).
        canonical-nf S (open-face S G h C pms lms k D0) =
          canonical-nf S
            (open-endpoint S G h C pms lms k
              (fst (lends (S h) ! k))) ∧
        canonical-nf S (open-face S G h C pms lms k D1) =
          canonical-nf S
            (open-endpoint S G h C pms lms k
              (snd (lends (S h) ! k)))
  proof (intro allI impI)
    fix k
    assume k: k < length (lcons (S h))
    have zero:
        joinable S
          (open-face S G h C pms lms k D0)
          (open-endpoint S G h C pms lms k
            (fst (lends (S h) ! k))) ⟷
        canonical-nf S (open-face S G h C pms lms k D0) =
          canonical-nf S
```

```
        (open-endpoint S G h C pms lms k
          (fst (lends (S h) ! k)))
    by (rule open-joinable-iff-canonical-nf-eq[
        OF base k wf-hsig-lhs[OF sig k]]) simp
  have one:
    joinable S
      (open-face S G h C pms lms k D1)
      (open-endpoint S G h C pms lms k
        (snd (lends (S h) ! k))) ⟷
    canonical-nf S (open-face S G h C pms lms k D1) =
      canonical-nf S
        (open-endpoint S G h C pms lms k
          (snd (lends (S h) ! k)))
    by (rule open-joinable-iff-canonical-nf-eq[
        OF base k wf-hsig-rhs[OF sig k]]) simp
  from joins[rule-format, OF k] zero one
  show canonical-nf S (open-face S G h C pms lms k D0) =
    canonical-nf S
      (open-endpoint S G h C pms lms k
        (fst (lends (S h) ! k))) ∧
   canonical-nf S (open-face S G h C pms lms k D1) =
    canonical-nf S
      (open-endpoint S G h C pms lms k
        (snd (lends (S h) ! k)))
    by blast
 qed
 from base equal show normal-boundary-check S D G h C pms lms
  unfolding normal-boundary-check-def by blast
qed

theorem open-boundary-check-sound:
 assumes check: open-boundary-check S D G h C pms lms
 shows typed-coherent-methods S h C pms lms
proof −
 from check have wf: wf-sigenv S
  and zeroD: 0 ∉ D
  and plen: length pms = length (pcons (S h))
  and llen: length lms = length (lcons (S h))
  and ptyped: ∀ k < length pms.
    wt S D G (pms ! k) (methodTy h C (pcons (S h) ! k))
  and ltyped: ∀ k < length lms.
    wt S (insert 0 D) G (lms ! k)
      (methodTy h C (lcons (S h) ! k))
  and open-obligations:
    ∀ k < length (lcons (S h)).
      joinable S
        (open-face S G h C pms lms k D0)
        (open-endpoint S G h C pms lms k
          (fst (lends (S h) ! k))) ∧
```

```
      joinable S
        (open-face S G h C pms lms k D1)
        (open-endpoint S G h C pms lms k
          (snd (lends (S h) ! k)))
  unfolding open-boundary-check-def by blast+
have sig: wf-hsig S h
  using wf unfolding wf-sigenv-def by blast
have pms-reg: list-all (regular S) pms
proof (rule typed-list-regular)
  show ∀ i < length pms. ∃ A. wt S D G (pms ! i) A
    using ptyped by blast
qed
have lms-reg: list-all (regular S) lms
proof (rule typed-list-regular)
  show ∀ i < length lms.
      ∃ A. wt S (insert 0 D) G (lms ! i) A
    using ltyped by blast
qed
show ?thesis
  unfolding typed-coherent-methods-def
proof (intro conjI)
  show length pms = length (pcons (S h)) by (rule plen)
next
  show length lms = length (lcons (S h)) by (rule llen)
next
  show ∀ k E H as.
      k < length (lcons (S h)) ⟶
      length as = length (lcons (S h) ! k) ⟶
      0 ∉ E ⟶
      (∀ i < length as.
        wt S E H (as ! i)
          (argTy h (lcons (S h) ! k ! i))) ⟶
      joinable S
        (apps (dsubst S 0 D0 (lms ! k))
          (elimArgs h C pms lms (lcons (S h) ! k) as))
        (elimEnd S h C pms lms as
          (fst (lends (S h) ! k))) ∧
      joinable S
        (apps (dsubst S 0 D1 (lms ! k))
          (elimArgs h C pms lms (lcons (S h) ! k) as))
        (elimEnd S h C pms lms as
          (snd (lends (S h) ! k)))
  proof (intro allI impI)
    fix k E H as
    assume k: k < length (lcons (S h))
      and len: length as = length (lcons (S h) ! k)
      and zeroE: 0 ∉ E
      and args: ∀ i < length as.
        wt S E H (as ! i) (argTy h (lcons (S h) ! k ! i))
```

```
  let ?ss = lcons (S h) ! k
  let ?oas = open-args G ?ss
  let ?env = open-inst G as
  have klm: k < length lms using k llen by simp
have prefix: ∀ i < length G. ?env i = Var i
  using open-inst-prefix by blast
have arg-exists: ∀ i < length as. ∃ A. wt S E H (as ! i) A
  using args by blast
have env-reg: ⋀i. regular S (?env i)
  by (rule open-inst-regular[OF arg-exists])
have env-closed: ⋀i. dclosed0 S (?env i)
  by (rule open-inst-dclosed0[OF zeroE arg-exists])
have pms-fixed: map (psubst ?env) pms = pms
proof (rule map-psubst-identity-wt[OF - prefix])
  show ∀ i < length pms. ∃ A. wt S D G (pms ! i) A
    using ptyped by blast
qed
have lms-fixed: map (psubst ?env) lms = lms
proof (rule map-psubst-identity-wt[OF - prefix])
  show ∀ i < length lms.
      ∃ A. wt S (insert 0 D) G (lms ! i) A
    using ltyped by blast
qed
have args-fixed: map (psubst ?env) ?oas = as
  by (rule psubst-open-args[OF len])
have oas-reg: list-all (regular S) ?oas
  by (auto simp: open-args-def list-all-iff)
have elim-args-reg: list-all (regular S)
    (elimArgs h C pms lms ?ss ?oas)
  by (rule regular-elimArgs[
       OF plen llen pms-reg lms-reg oas-reg])

have lm-typed: wt S (insert 0 D) G (lms ! k)
    (methodTy h C ?ss)
  using ltyped klm by simp
have face0-typed: wt S D G (dsubst S 0 D0 (lms ! k))
    (methodTy h C ?ss)
proof −
  have wt S (dctx-subst 0 D0 (insert 0 D)) G
      (dsubst S 0 D0 (lms ! k)) (methodTy h C ?ss)
    by (rule wt-dimension-subst[OF lm-typed wf])
  with zeroD show ?thesis
    by (simp add: dctx-subst-def)
qed
have face1-typed: wt S D G (dsubst S 0 D1 (lms ! k))
    (methodTy h C ?ss)
proof −
  have wt S (dctx-subst 0 D1 (insert 0 D)) G
      (dsubst S 0 D1 (lms ! k)) (methodTy h C ?ss)
```

```
    by (rule wt-dimension-subst[OF lm-typed wf])
  with zeroD show ?thesis
    by (simp add: dctx-subst-def)
qed
have face0-head-reg: regular S (dsubst S 0 D0 (lms ! k))
  by (rule regular-wt[OF face0-typed])
have face1-head-reg: regular S (dsubst S 0 D1 (lms ! k))
  by (rule regular-wt[OF face1-typed])
have face0-reg: regular S (open-face S G h C pms lms k D0)
  unfolding open-face-def
  by (rule regular-apps[OF face0-head-reg elim-args-reg])
have face1-reg: regular S (open-face S G h C pms lms k D1)
  unfolding open-face-def
  by (rule regular-apps[OF face1-head-reg elim-args-reg])

have lhs-ep: wf-endpt S h ?ss
    (fst (lends (S h) ! k)) (Hit h)
  by (rule wf-hsig-lhs[OF sig k])
have rhs-ep: wf-endpt S h ?ss
    (snd (lends (S h) ! k)) (Hit h)
  by (rule wf-hsig-rhs[OF sig k])
have lhs-reg: regular S
    (open-endpoint S G h C pms lms k
      (fst (lends (S h) ! k)))
  unfolding open-endpoint-def
  by (rule regular-elimEnd[
        OF lhs-ep open-args-length oas-reg
          plen llen pms-reg lms-reg])
have rhs-reg: regular S
    (open-endpoint S G h C pms lms k
      (snd (lends (S h) ! k)))
  unfolding open-endpoint-def
  by (rule regular-elimEnd[
        OF rhs-ep open-args-length oas-reg
          plen llen pms-reg lms-reg])

have face0-head-fixed:
    psubst ?env (dsubst S 0 D0 (lms ! k)) =
      dsubst S 0 D0 (lms ! k)
  by (rule psubst-identity-wt[OF face0-typed prefix])
have face1-head-fixed:
    psubst ?env (dsubst S 0 D1 (lms ! k)) =
      dsubst S 0 D1 (lms ! k)
  by (rule psubst-identity-wt[OF face1-typed prefix])
have face0-inst:
    psubst ?env (open-face S G h C pms lms k D0) =
      apps (dsubst S 0 D0 (lms ! k))
        (elimArgs h C pms lms ?ss as)
  unfolding open-face-def
```

```
  by (simp add: psubst-elimArgs pms-fixed lms-fixed
      args-fixed face0-head-fixed)
have face1-inst:
    psubst ?env (open-face S G h C pms lms k D1) =
      apps (dsubst S 0 D1 (lms ! k))
        (elimArgs h C pms lms ?ss as)
  unfolding open-face-def
  by (simp add: psubst-elimArgs pms-fixed lms-fixed
      args-fixed face1-head-fixed)
have lhs-inst:
    psubst ?env
      (open-endpoint S G h C pms lms k
        (fst (lends (S h) ! k))) =
      elimEnd S h C pms lms as (fst (lends (S h) ! k))
  unfolding open-endpoint-def
  using psubst-elimEnd[OF lhs-ep open-args-length plen,
      where s = ?env]
  by (simp add: pms-fixed lms-fixed args-fixed)
have rhs-inst:
    psubst ?env
      (open-endpoint S G h C pms lms k
        (snd (lends (S h) ! k))) =
      elimEnd S h C pms lms as (snd (lends (S h) ! k))
  unfolding open-endpoint-def
  using psubst-elimEnd[OF rhs-ep open-args-length plen,
      where s = ?env]
  by (simp add: pms-fixed lms-fixed args-fixed)

have open0: joinable S
    (open-face S G h C pms lms k D0)
    (open-endpoint S G h C pms lms k
      (fst (lends (S h) ! k)))
  using open-obligations k by blast
have open1: joinable S
    (open-face S G h C pms lms k D1)
    (open-endpoint S G h C pms lms k
      (snd (lends (S h) ! k)))
  using open-obligations k by blast
have inst0: joinable S
    (psubst ?env (open-face S G h C pms lms k D0))
    (psubst ?env
      (open-endpoint S G h C pms lms k
        (fst (lends (S h) ! k))))
  by (rule joinable-psubst-dclosed[
       OF open0 face0-reg lhs-reg env-reg env-closed])
have inst1: joinable S
    (psubst ?env (open-face S G h C pms lms k D1))
    (psubst ?env
      (open-endpoint S G h C pms lms k
```

```
        (snd (lends (S h) ! k))))
    by (rule joinable-psubst-dclosed[
        OF open1 face1-reg rhs-reg env-reg env-closed])
    show joinable S
      (apps (dsubst S 0 D0 (lms ! k))
        (elimArgs h C pms lms ?ss as))
      (elimEnd S h C pms lms as
        (fst (lends (S h) ! k))) ∧
    joinable S
      (apps (dsubst S 0 D1 (lms ! k))
        (elimArgs h C pms lms ?ss as))
      (elimEnd S h C pms lms as
        (snd (lends (S h) ! k)))
      using inst0 inst1 face0-inst face1-inst lhs-inst rhs-inst
      by simp
  qed
 qed
qed

corollary normal-boundary-check-sound:
 assumes normal-boundary-check S D G h C pms lms
 shows typed-coherent-methods S h C pms lms
 by (rule open-boundary-check-sound)
  (use assms normal-boundary-check-iff-open-boundary-check in blast)

corollary open-boundary-square:
 assumes sig: wf-hsig S h
  and check: open-boundary-check S D G h C pms lms
  and k: k < length (lcons (S h))
  and len: length as = length (lcons (S h) ! k)
  and zero: 0 ∉ E
  and args: ∀ i < length as.
    wt S E H (as ! i) (argTy h (lcons (S h) ! k ! i))
 shows conv S
    (HElim h C pms lms (mkL S h k as D0))
    (apps (dsubst S 0 D0 (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as))
  and conv S
    (HElim h C pms lms (mkL S h k as D1))
    (apps (dsubst S 0 D1 (lms ! k))
      (elimArgs h C pms lms (lcons (S h) ! k) as))
 by (rule typed-elim-path-boundary-square[
      OF sig open-boundary-check-sound[OF check] k len zero args])+

end
theory HIT-Examples
 imports HIT-Open-Boundary-Check
begin
```

20 The schema at work

The standard examples that fit the schema are represented by individual signatures. They are collected in one signature environment so that they can refer to one another: the suspension of the unit type, and the propositional truncation of the natural numbers, are parametrised by the codes of those types.

```
definition emptySig :: hsig where
  emptySig = (| pcons = [], lcons = [], lends = [] |)

definition unitSig :: hsig where
  unitSig = (| pcons = [[]], lcons = [], lends = [] |)

definition natSig :: hsig where
  natSig = (| pcons = [[], [Rec]], lcons = [], lends = [] |)

definition circleSig :: hsig where
  circleSig =
    (| pcons = [[]], lcons = [[]], lends = [(ECon 0 [], ECon 0 [])] |)

definition intervalSig :: hsig where
  intervalSig =
    (| pcons = [[], []], lcons = [[]],
      lends = [(ECon 0 [], ECon 1 [])] |)

definition suspSig :: nat ⇒ hsig where
  suspSig a =
    (| pcons = [[], []], lcons = [[Par a]],
      lends = [(ECon 0 [], ECon 1 [])] |)

definition truncSig :: nat ⇒ hsig where
  truncSig a =
    (| pcons = [[Par a]], lcons = [[Rec, Rec]],
      lends = [(EArg 0, EArg 1)] |)
```

Codes: *0* the circle, *1* the interval, *2* the natural numbers, *3* the empty type, *4* the unit type, *5* the suspension of the unit type, *6* the propositional truncation of the natural numbers.

```
definition demo :: sigenv where
  demo n =
    (if n = 0 then circleSig
     else if n = 1 then intervalSig
     else if n = 2 then natSig
     else if n = 3 then emptySig
     else if n = 4 then unitSig
     else if n = 5 then suspSig 4
     else if n = 6 then truncSig 2
     else emptySig)
```

lemma *demo-simps* [*simp*]:
demo 0 = *circleSig*
demo 1 = *intervalSig*
demo 2 = *natSig*
demo 3 = *emptySig*
demo 4 = *unitSig*
demo 5 = *suspSig 4*
demo 6 = *truncSig 2*
by (*simp-all add*: *demo-def*)

20.1 Well-formed signatures

lemma *demo-wf-circle*: *wf-hsig demo 0*
by (*auto simp*: *wf-hsig-def circleSig-def demo-def*
intro!: *wf-endpt.wfECon*)

lemma *demo-wf-interval*: *wf-hsig demo 1*
by (*auto simp*: *wf-hsig-def intervalSig-def demo-def*
intro!: *wf-endpt.wfECon*)

lemma *demo-wf-nat*: *wf-hsig demo 2*
by (*simp add*: *wf-hsig-def natSig-def demo-def*)

lemma *demo-wf-empty*: *wf-hsig demo 3*
by (*simp add*: *wf-hsig-def emptySig-def demo-def*)

lemma *demo-wf-unit*: *wf-hsig demo 4*
by (*simp add*: *wf-hsig-def unitSig-def demo-def*)

lemma *demo-wf-suspension*: *wf-hsig demo 5*
by (*auto simp*: *wf-hsig-def suspSig-def demo-def*
intro!: *wf-endpt.wfECon*)

lemma *demo-wf-truncation*: *wf-hsig demo 6*
proof −
have *e0*: *wf-endpt demo 6* [*Rec*, *Rec*] (*EArg 0*) (*Hit 6*)
proof −
have *wf-endpt demo 6* [*Rec*, *Rec*] (*EArg 0*)
(*argTy 6* ([*Rec*, *Rec*] ! *0*))
by (*rule wf-endpt.wfEArg*) *simp*
then show *?thesis* **by** *simp*
qed
have *e1*: *wf-endpt demo 6* [*Rec*, *Rec*] (*EArg 1*) (*Hit 6*)
proof −
have *wf-endpt demo 6* [*Rec*, *Rec*] (*EArg 1*)
(*argTy 6* ([*Rec*, *Rec*] ! *1*))
by (*rule wf-endpt.wfEArg*) *simp*
then show *?thesis* **by** *simp*

```
  qed
  show ?thesis
    using e0 e1 by (simp add: wf-hsig-def truncSig-def demo-def)
qed

lemma demo-ge-seven:
  assumes 7 ≤ h
  shows demo h = emptySig
  using assms by (simp add: demo-def)

theorem demo-wf: wf-sigenv demo
  unfolding wf-sigenv-def
proof
  fix h :: nat
  consider h = 0 | h = 1 | h = 2 | h = 3 |
    h = 4 | h = 5 | h = 6 | 7 ≤ h
    by linarith
  then show wf-hsig demo h
  proof cases
    case 1
    with demo-wf-circle show ?thesis by simp
  next
    case 2
    with demo-wf-interval show ?thesis by simp
  next
    case 3
    with demo-wf-nat show ?thesis by simp
  next
    case 4
    with demo-wf-empty show ?thesis by simp
  next
    case 5
    with demo-wf-unit show ?thesis by simp
  next
    case 6
    with demo-wf-suspension show ?thesis by simp
  next
    case 7
    with demo-wf-truncation show ?thesis by simp
  next
    case 8
    with demo-ge-seven[OF this] show ?thesis
      by (simp add: wf-hsig-def emptySig-def)
  qed
qed
```

20.2 Boundaries of path constructors

The two endpoints of a path constructor are not extra rules: they are what the path constructor *is* at the two constant dimensions.

abbreviation *base* :: *tm* **where** *base* ≡ *PCon 0 0* []
abbreviation *loop* :: *nat* ⇒ *tm* **where** *loop j* ≡ *LCon 0 0* [] (*DVar j*)

theorem *loop-boundary-zero*: *mkL demo 0 0* [] *D0* = *base*
  by (*simp add*: *lhsOf-def circleSig-def*)

theorem *loop-boundary-one*: *mkL demo 0 0* [] *D1* = *base*
  by (*simp add*: *rhsOf-def circleSig-def*)

theorem *loop-dsubst-zero*: *dsubst demo j D0* (*loop j*) = *base*
  by (*simp add*: *lhsOf-def circleSig-def*)

theorem *loop-dsubst-one*: *dsubst demo j D1* (*loop j*) = *base*
  by (*simp add*: *rhsOf-def circleSig-def*)

abbreviation *izero* :: *tm* **where** *izero* ≡ *PCon 1 0* []
abbreviation *ione* :: *tm* **where** *ione* ≡ *PCon 1 1* []
abbreviation *seg* :: *nat* ⇒ *tm* **where** *seg j* ≡ *LCon 1 0* [] (*DVar j*)

theorem *seg-boundary*:
  dsubst demo j D0 (*seg j*) = *izero*
  dsubst demo j D1 (*seg j*) = *ione*
  by (*simp-all add*: *lhsOf-def rhsOf-def intervalSig-def demo-def*)

For the suspension, the meridian at a point of the parameter type joins the two poles; for the propositional truncation, the squashing path joins its own two arguments, which is the case in which an endpoint is an argument rather than a point constructor.

theorem *merid-boundary*:
  dsubst demo j D0 (*LCon 5 0* [*a*] (*DVar j*)) = *PCon 5 0* []
  dsubst demo j D1 (*LCon 5 0* [*a*] (*DVar j*)) = *PCon 5 1* []
  by (*simp-all add*: *lhsOf-def rhsOf-def suspSig-def*)

theorem *squash-boundary*:
  dsubst demo j D0 (*LCon 6 0* [*x*, *y*] (*DVar j*)) =
    dsubst demo j D0 x
  dsubst demo j D1 (*LCon 6 0* [*x*, *y*] (*DVar j*)) =
    dsubst demo j D1 y
  by (*simp-all add*: *lhsOf-def rhsOf-def truncSig-def*)

20.3 Boundary-coherent path methods

The identity eliminator for the interval uses its two point constructors as point methods and its segment as path method. The checked boundary condition verifies both faces.

```
theorem interval-identity-coherent:
  coherent-methods demo 1 (Hit 1) [izero, ione] [seg 0]
  by (auto simp: coherent-methods-def boundary-args-def intervalSig-def
      demo-def lhsOf-def rhsOf-def intro: conv-refl)

theorem interval-identity-open-boundary-check:
  assumes 0 ∉ D
  shows open-boundary-check demo D [] 1 (Hit 1)
    [izero, ione] [seg 0]
proof −
  have ptyped: ∀ k < length [izero, ione].
      wt demo D [] ([izero, ione] ! k)
        (methodTy 1 (Hit 1) (pcons (demo 1) ! k))
  proof (intro allI impI)
    fix k
    assume k < length [izero, ione]
    then have k < 2 by simp
    then have k = 0 ∨ k = 1 by arith
    then show wt demo D [] ([izero, ione] ! k)
        (methodTy 1 (Hit 1) (pcons (demo 1) ! k))
      by (auto simp: intervalSig-def demo-def intro!: wt.intros)
  qed
  have ltyped: ∀ k < length [seg 0].
      wt demo (insert 0 D) [] ([seg 0] ! k)
        (methodTy 1 (Hit 1) (lcons (demo 1) ! k))
    by (auto simp: intervalSig-def demo-def intro!: wt.intros)
  show ?thesis
    using demo-wf assms ptyped ltyped
    by (auto simp: open-boundary-check-def open-face-def open-endpoint-def
        open-args-def intervalSig-def demo-def lhsOf-def rhsOf-def
        joinable-def)
qed

theorem interval-identity-normal-boundary-check:
  assumes 0 ∉ D
  shows normal-boundary-check demo D [] 1 (Hit 1)
    [izero, ione] [seg 0]
  using interval-identity-open-boundary-check[OF assms]
    normal-boundary-check-iff-open-boundary-check by blast

theorem interval-identity-typed-coherent:
  typed-coherent-methods demo 1 (Hit 1) [izero, ione] [seg 0]
  by (rule normal-boundary-check-sound[
        OF interval-identity-normal-boundary-check, where D = {}])
    simp
```

In contrast, a constant path method can be coherent only if its value is convertible to the right point method. Thus the unchecked counterexample with unrelated endpoint methods is rejected by the checked judgment.

theorem *interval-constant-method-requires-right-endpoint*:
  assumes *coh*: *coherent-methods demo 1 C* [*u*, *v*] [*u*]
    and *fixed*: *dsubst demo 0 D1 u* = *u*
  shows *conv demo u v*
proof −
  have *conv demo*
      (*apps* (*dsubst demo 0 D1* ([*u*] ! *0*))
        (*elimArgs 1 C* [*u*, *v*] [*u*] (*lcons* (*demo 1*) ! *0*) []))
      (*elimEnd demo 1 C* [*u*, *v*] [*u*] []
        (*snd* (*lends* (*demo 1*) ! *0*)))
    by (*rule coherent-methods-one*[*OF coh*, *of 0* []])
      (*simp-all add*: *intervalSig-def demo-def boundary-args-def*)
  with *fixed* **show** *?thesis*
    by (*simp add*: *intervalSig-def demo-def*)
qed

theorem *interval-open-check-rejects-constant-method*:
  assumes *check*: *open-boundary-check demo D G 1 C* [*u*, *v*] [*u*]
    and *fixed*: *dsubst demo 0 D1 u* = *u*
  shows *conv demo u v*
proof −
  have *joinable demo u v*
    using *check fixed*
    unfolding *open-boundary-check-def open-face-def open-endpoint-def*
      open-args-def intervalSig-def demo-def
    by *simp*
  then show *?thesis* **by** (*rule joinable-conv*)
qed

abbreviation *interval-id* :: *nat* ⇒ *tm* **where**
  interval-id j ≡
    HElim 1 (*Hit 1*) [*izero*, *ione*] [*seg 0*] (*seg j*)

lemma *interval-identity-locally-checked*:
  checked demo (*interval-id j*)
  using *demo-wf-interval interval-identity-typed-coherent* **by** *simp*

lemma *seg-locally-checked*: *checked demo* (*seg j*)
  using *demo-wf-interval* **by** *simp*

lemma *seg-normal*: *nform demo* (*seg j*)
  unfolding *nform-def*
  by (*auto elim*: *red.cases*)

lemma *interval-point-method-no-step*:
  assumes [*izero*, *ione*] = *xs* @ *m* # *ys* **and** *red demo m m′*
  shows *False*
proof −
  have *m* ∈ *set* [*izero*, *ione*] **using** *assms*(*1*) **by** *auto*

```
  with assms(2) show False by (auto elim: red.cases)
qed

lemma interval-path-method-no-step:
  assumes [seg 0] = xs @ m # ys and red demo m m'
  shows False
proof −
  have m ∈ set (xs @ m # ys) by simp
  then have m ∈ set [seg 0] using assms(1) by simp
  then have m = seg 0 by simp
  with assms(2) show False by (auto elim: red.cases)
qed

lemma interval-identity-red:
  assumes red demo (interval-id j) u
  shows u = seg j
using assms
proof (rule red-HElimE)
  fix k as
  assume source: seg j = PCon 1 k as
    and k < length (pcons (demo 1))
    and u = apps ([izero, ione] ! k)
      (elimArgs 1 (Hit 1) [izero, ione] [seg 0]
        (pcons (demo 1) ! k) as)
  from source show u = seg j by simp
next
  fix k as q
  assume source: seg j = LCon 1 k as (DVar q)
    and k: k < length (lcons (demo 1))
    and target: u =
      apps (dsubst demo 0 (DVar q) ([seg 0] ! k))
        (elimArgs 1 (Hit 1) [izero, ione] [seg 0]
          (lcons (demo 1) ! k) as)
  from source have k = 0 as = [] q = j by simp-all
  with target show u = seg j
    by (simp add: intervalSig-def demo-def)
next
  fix xs m m' ys
  assume methods: [izero, ione] = xs @ m # ys
    and step: red demo m m'
    and u = HElim 1 (Hit 1) (xs @ m' # ys) [seg 0] (seg j)
  from interval-point-method-no-step[OF methods step]
  show u = seg j by blast
next
  fix xs m m' ys
  assume methods: [seg 0] = xs @ m # ys
    and step: red demo m m'
    and u = HElim 1 (Hit 1) [izero, ione] (xs @ m' # ys) (seg j)
  from interval-path-method-no-step[OF methods step]
```

```
  show u = seg j by blast
next
  fix t'
  assume step: red demo (seg j) t'
    and u = HElim 1 (Hit 1) [izero, ione] [seg 0] t'
  with seg-normal show u = seg j
    unfolding nform-def by blast
qed

lemma interval-identity-reducts:
  assumes reds demo (interval-id j) u
  shows u = interval-id j ∨ u = seg j
  using assms
  by (metis interval-identity-red nform-reds-eq reds-first-step seg-normal)

lemma interval-identity-hereditarily-checked:
  hereditarily-checked demo (interval-id j)
  unfolding hereditarily-checked-def
  using interval-identity-locally-checked interval-identity-reducts
    seg-locally-checked by blast

theorem interval-identity-checked:
  assumes j: j ∈ D and zero: 0 ∉ D
  shows cwt demo D []
    (interval-id j) (Hit 1)
proof (rule cwtI[OF zero])
  show wt demo D [] (interval-id j) (Hit 1)
  proof (rule wt.wtElim)
    show wt demo D [] (seg j) (Hit 1)
      by (rule wt.wtLCon[OF j])
        (simp-all add: intervalSig-def demo-def)
  next
    show ∀ k < length [izero, ione].
          wt demo D [] ([izero, ione] ! k) (methodTy 1 (Hit 1) (pcons (demo 1) !
k))
        ∀ k < length [seg 0].
          wt demo (insert 0 D) [] ([seg 0] ! k) (methodTy 1 (Hit 1) (lcons (demo
1) ! k))
      using interval-identity-open-boundary-check open-boundary-check-def zero
      by auto
  qed (auto simp: intervalSig-def demo-def)
next
  show hereditarily-checked demo (interval-id j)
    by (rule interval-identity-hereditarily-checked)
qed

theorem interval-identity-boundary-square:
  conv demo
    (HElim 1 (Hit 1) [izero, ione] [seg 0] izero)
```

```
    izero
  conv demo
    (HElim 1 (Hit 1) [izero, ione] [seg 0] ione)
    ione
  using elim-path-boundary-square[
      OF demo-wf-interval interval-identity-coherent,
      where k = 0 and as = []]
  by (simp-all add: intervalSig-def demo-def lhsOf-def rhsOf-def
      boundary-args-def)
```

20.4 Computation rules

The eliminator computes on the point constructor of the circle, and *judgmentally* on its path constructor: this is the cubical-style computation rule that distinguishes this operational presentation from axiomatic presentations whose path computation is only propositional.

```
theorem circle-rec-base:
  red demo (HElim 0 C [b] [l] base) b
proof −
  have red demo (HElim 0 C [b] [l] (PCon 0 0 []))
      (apps ([b] ! 0) (elimArgs 0 C [b] [l] (pcons (demo 0) ! 0) []))
    by (rule red.elimPoint) (simp add: circleSig-def)
  then show ?thesis by (simp add: circleSig-def)
qed

theorem circle-rec-loop:
  red demo (HElim 0 C [b] [l] (loop j)) (dsubst demo 0 (DVar j) l)
proof −
  have red demo (HElim 0 C [b] [l] (LCon 0 0 [] (DVar j)))
      (apps (dsubst demo 0 (DVar j) ([l] ! 0))
        (elimArgs 0 C [b] [l] (lcons (demo 0) ! 0) []))
    by (rule red.elimPath) (simp add: circleSig-def)
  then show ?thesis by (simp add: circleSig-def)
qed

theorem interval-rec-seg:
  red demo (HElim 1 C [u, v] [p] (seg j)) (dsubst demo 0 (DVar j) p)
proof −
  have red demo (HElim 1 C [u, v] [p] (LCon 1 0 [] (DVar j)))
      (apps (dsubst demo 0 (DVar j) ([p] ! 0))
        (elimArgs 1 C [u, v] [p] (lcons (demo 1) ! 0) []))
    by (rule red.elimPath) (simp add: intervalSig-def demo-def)
  then show ?thesis by (simp add: intervalSig-def demo-def)
qed

theorem susp-rec-merid:
  red demo (HElim 5 C [u, v] [p] (LCon 5 0 [a] (DVar j)))
    (App (dsubst demo 0 (DVar j) p) a)
proof −
```

```
  have red demo (HElim 5 C [u, v] [p] (LCon 5 0 [a] (DVar j)))
      (apps (dsubst demo 0 (DVar j) ([p] ! 0))
        (elimArgs 5 C [u, v] [p] (lcons (demo 5) ! 0) [a]))
    by (rule red.elimPath) (simp add: suspSig-def)
  then show ?thesis by (simp add: suspSig-def)
qed
```

Ordinary inductive types are the signatures with no path constructors, and their computation rules are the same rule at a signature whose list of path constructors is empty.

```
theorem nat-rec-zero:
  red demo (HElim 2 C [z, s] [] (PCon 2 0 [])) z
proof −
  have red demo (HElim 2 C [z, s] [] (PCon 2 0 []))
      (apps ([z, s] ! 0) (elimArgs 2 C [z, s] [] (pcons (demo 2) ! 0) []))
    by (rule red.elimPoint) (simp add: natSig-def)
  then show ?thesis by (simp add: natSig-def)
qed

theorem nat-rec-succ:
  red demo (HElim 2 C [z, s] [] (PCon 2 1 [n]))
    (App (App s n) (HElim 2 C [z, s] [] n))
proof −
  have red demo (HElim 2 C [z, s] [] (PCon 2 1 [n]))
      (apps ([z, s] ! 1)
        (elimArgs 2 C [z, s] [] (pcons (demo 2) ! 1) [n]))
    by (rule red.elimPoint) (simp add: natSig-def)
  then show ?thesis by (simp add: natSig-def)
qed
```

20.5 Typing

```
theorem base-typed: wt demo D [] base (Hit 0)
  by (rule wt.wtPCon) (simp-all add: circleSig-def)

theorem loop-typed:
  assumes j ∈ D
  shows wt demo D [] (loop j) (Hit 0)
  by (rule wt.wtLCon[OF assms]) (simp-all add: circleSig-def)
```

A path constructor needs a dimension variable, so in the empty dimension context the circle has no path-constructor terms at all. This is the syntactic shadow of the canonical-forms theorem: the closed values of a higher inductive type in dimension context zero are its point constructors.

```
theorem no-loop-without-dimension:
  ¬ wt demo {} G (LCon h k as (DVar j)) A
  by (auto elim: wt.cases)
```

The empty type is the signature with no constructors, and it has neither point nor path constructor terms.

```
theorem empty-no-constructors:
  ¬ wt demo D G (PCon 3 k as) A
  ¬ wt demo D G (LCon 3 k as d) A
  by (auto simp add: emptySig-def elim: wt.cases)
```

20.6 The loop is a value that only a dimension makes possible

Both the base point and the loop are normal forms: no computation rule applies to a bare constructor term without arguments.

```
theorem base-nform: nform demo base
  using nform-def red.cases by fastforce

theorem loop-nform: nform demo (loop j)
  unfolding nform-def
proof
  assume ∃ u. red demo (loop j) u
  then obtain u where red demo (LCon 0 0 [] (DVar j)) u by blast
  then show False by (cases rule: red.cases) auto
qed
```

The loop is not a canonical value: canonical values are point constructors, while a path constructor needs a dimension variable. The loop is therefore a term that exists only relative to a dimension, which is exactly the cubical-style reading of a path.

```
theorem loop-not-canonical: ¬ canonical demo 0 (loop j)
  by (auto elim: canonical.cases)

theorem base-canonical: canonical demo 0 base
  by (rule canonicalI) (simp-all add: circleSig-def)

end
```

21 AI assistance

AI assistance was used for proof engineering. The final definitions, statements, and proofs are checked by Isabelle.

References

[1] E. Cavallo and R. Harper. Higher inductive types in cubical computational type theory. *Proceedings of the ACM on Programming Languages*, 3(POPL):1:1–1:27, 2019. DOI: `https://doi.org/10.1145/3290314`.

[2] J. Chen. Homotopy type theory in Isabelle. In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceed-*

ings in Informatics, pages 12:1–12:8, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. DOI: https://doi.org/10.4230/LIPIcs.ITP.2021.12.

[3] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. DOI: https://doi.org/10.4230/LIPIcs.TYPES.2015.5.

[4] S. Huber. Canonicity for cubical type theory. *Journal of Automated Reasoning*, 63(2):173–210, 2019. DOI: https://doi.org/10.1007/s10817-018-9469-1.

[5] A. Kaposi and A. Kovács. Signatures and induction principles for higher inductive-inductive types. *Logical Methods in Computer Science*, 16(1):10:1–10:30, 2020. DOI: https://doi.org/10.23638/LMCS-16(1:10)2020.

[6] P. L. Lumsdaine and M. Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 169(1):159–208, 2020. DOI: https://doi.org/10.1017/S030500411900015X.

[7] K. Sojakova. Higher inductive types as homotopy-initial algebras. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, pages 31–42, New York, NY, USA, 2015. ACM. DOI: https://doi.org/10.1145/2676726.2676983.

[8] J. Sterling and C. Angiuli. Normalization for cubical type theory. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–15. IEEE, 2021. DOI: https://doi.org/10.1109/LICS52264.2021.9470719.

[9] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, Princeton, NJ, 2013. https://homotopytypetheory.org/book/.

[10] N. Veltri and N. van der Weide. Constructing higher inductive types as groupoid quotients. *Logical Methods in Computer Science*, 17(2):8:1–8:42, 2021. DOI: https://doi.org/10.23638/LMCS-17(2:8)2021.