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 11
- 5.2 Reduction inside constructor arguments 12

6 Typing **13**
- 6.1 Applying a method to the eliminator arguments 14

7 Well-formed signatures and endpoints **14**

8 Dimension support **16**

9 Boundary-coherent elimination **17**
- 9.1 The boundary-computation square 20

10 Structural properties of typing **23**

11 Term substitution **24**

12 Dimension substitution **24**

13 Subject reduction **25**

14 Progress and canonical forms **25**
14.1 Canonical values 26
14.2 Normal forms are closed under the relevant subterms 26
14.3 Progress 27
14.4 The empty signature 28

15 Strong normalization **29**
15.1 Dimension erasure and regular terms 29
15.2 Renaming and simultaneous substitution 32
15.3 Hereditary computability at HIT types 36
15.4 A Kripke logical relation 40
15.5 The hereditary Kripke candidate 42
15.6 Substitution and reducibility 43
15.7 Constructor and method arguments 45
15.8 The fundamental theorem 49

16 Unconditional canonical results and consistency **51**

17 Confluence **51**
17.1 A dimension-capture critical pair 51
17.2 Multi-step compatibility 52
17.3 Dimension-safe reduction 55
17.4 Local peaks 57

18 Canonical normal-form comparison **61**

19 Finite open-term boundary checking **62**

20 The schema at work **68**
20.1 Well-formed signatures 69
20.2 Boundaries of path constructors 70
20.3 Boundary-coherent path methods 71
20.4 Computation rules 73
20.5 Typing 74
20.6 The loop is a value that only a dimension makes possible . . 74

21 AI assistance **75**

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)
  ⟨proof⟩
```

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
  ⟨proof⟩

lemma apps-Cons [simp]: apps t (x # xs) = apps (App t x) xs
  ⟨proof⟩

lemma apps-append: apps t (xs @ ys) = apps (apps t xs) ys
  ⟨proof⟩
```

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 [] = []
  ⟨proof⟩

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
  ⟨proof⟩

lemma dclosed0-Var [simp]: dclosed0 S (Var i)
  ⟨proof⟩

lemma dclosed0-Lam [simp]:
  dclosed0 S (Lam A t) ⟷ dclosed0 S t
  ⟨proof⟩

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

```
    dclosed0 S t ∧ dclosed0 S u
  ⟨proof⟩

lemma dclosed0-PCon [simp]:
  dclosed0 S (PCon h k as) ⟷
    list-all (dclosed0 S) as
  ⟨proof⟩

lemma dclosed0-LCon [simp]:
  dclosed0 S (LCon h k as (DVar i)) ⟷
    i ≠ 0 ∧ list-all (dclosed0 S) as
⟨proof⟩

lemma dclosed0-HElim [simp]:
  dclosed0 S (HElim h C pms lms t) ⟷
    list-all (dclosed0 S) pms ∧ dclosed0 S t
  ⟨proof⟩

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*
⟨*proof*⟩

lemma *conv-sym*:
assumes *conv S t u*
shows *conv S u t*
⟨*proof*⟩

lemma *conv-trans*:
conv S t u ⟹ conv S u v ⟹ conv S t v
⟨*proof*⟩

lemma *red-conv*: *red S t u ⟹ conv S t u*
⟨*proof*⟩

lemma *reds-conv*: *reds S t u ⟹ conv S t u*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *joinable-sym*:
joinable S t u ⟹ joinable S u t
⟨*proof*⟩

lemma *joinableI*:
reds S t v ⟹ reds S u v ⟹ joinable S t u
⟨*proof*⟩

lemma *joinable-conv*:
assumes *joinable S t u*
shows *conv S t u*
⟨*proof*⟩

5.1 Normal forms and the strong-normalization predicate

definition *nform* :: *sigenv ⇒ tm ⇒ bool* **where**

nform S t $\longleftrightarrow$ ($\nexists$ *u*. *red S t u*)

inductive *SN* :: *sigenv* $\Rightarrow$ *tm* $\Rightarrow$ *bool* **for** *S* :: *sigenv* **where**
SNI: ($\bigwedge$*u*. *red S t u* $\Longrightarrow$ *SN S u*) $\Longrightarrow$ *SN S t*

lemma *SN-step*: *SN S t* $\Longrightarrow$ *red S t u* $\Longrightarrow$ *SN S u*
⟨*proof*⟩

lemma *SN-reds*: *reds S t u* $\Longrightarrow$ *SN S t* $\Longrightarrow$ *SN S u*
⟨*proof*⟩

lemma *SN-nform*: *nform S t* $\Longrightarrow$ *SN S t*
⟨*proof*⟩

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*: $\bigwedge$*a b*. *red S a b* $\Longrightarrow$ *red S* (*f a*) (*f b*)
and *sn*: *SN S s*
and *eq*: *s* = *f t*
shows *SN S t*
⟨*proof*⟩

lemma *SN-App1*: *SN S* (*App t u*) $\Longrightarrow$ *SN S t*
⟨*proof*⟩

lemma *SN-App2*: *SN S* (*App t u*) $\Longrightarrow$ *SN S u*
⟨*proof*⟩

lemma *SN-apps-head*: *SN S* (*apps t xs*) $\Longrightarrow$ *SN S t*
⟨*proof*⟩

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*⟩

lemma *SN-PCon-arg*:
assumes *sn*: *SN S* (*PCon h k as*) **and** *i*: *i* < *length as*
shows *SN S* (*as* ! *i*)
⟨*proof*⟩

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*⟩

lemma *SN-LCon-arg*:
  assumes *sn*: *SN S* (*LCon h k as d*) **and** *i*: *i* < *length as*
  shows *SN S* (*as* ! *i*)
⟨*proof*⟩

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: ⋀a. wt S D G a (Hit h) ⟹
      wt S D G (HElim h C pms lms a) C
  shows ⋀as m. length as = length ss ⟹
    (∀ i < length ss. wt S D G (as ! i) (argTy h (ss ! i))) ⟹
    wt S D G m (methodTy h C ss) ⟹
    wt S D G (apps m (elimArgs h C pms lms ss as)) C
⟨proof⟩

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))
  ⟨proof⟩

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)
  ⟨proof⟩

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)
  ⟨proof⟩

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
  ⟨proof⟩

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)
```

⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

theorem *wt-fdims*:
assumes *wt S D G t A*
shows *fdims t* ⊆ *D*
⟨*proof*⟩

lemma *dsubstD-fresh*:
assumes *n* ∉ *dim-support e*
shows *dsubstD n d e* = *e*
⟨*proof*⟩

theorem *wt-dsubst-fresh*:
assumes *typed*: *wt S D G t A* **and** *fresh*: $n \notin D$
shows *dsubst S n d t* = *t*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

corollary *typed-constructor-args-boundary*:
assumes *0* ∉ *D*
and ∀ *i* < *length as*. *wt S D G* (*as* ! *i*) (*argTy h* (*ss* ! *i*))
shows *boundary-args S as*
⟨*proof*⟩

definition *coherent-methods* ::
sigenv ⇒ *nat* ⇒ *ty* ⇒ *tm list* ⇒ *tm list* ⇒ *bool*
where
coherent-methods S h C pms lms ⟷
length pms = *length* (*pcons* (*S h*)) ∧
length lms = *length* (*lcons* (*S h*)) ∧
(∀ *k* < *length* (*lcons* (*S h*)). ∀ *as*.
length as = *length* (*lcons* (*S h*) ! *k*) ⟶
boundary-args S as ⟶
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*))) ∧
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*))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*⟩

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*⟩

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⟩

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⟩
```

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 ⟹ wt S D G t A*
⟨*proof*⟩

lemma *cwt-checked*: *cwt S D G t A ⟹ checked S t*
⟨*proof*⟩

lemma *cwt-dimension-context*: *cwt S D G t A ⟹ 0 ∉ D*
⟨*proof*⟩

lemma *cwt-hereditarily-checked*:
cwt S D G t A ⟹ hereditarily-checked S t
⟨*proof*⟩

lemma *cwtI*:
assumes *0 ∉ D* **and** *wt S D G t A*
and *hereditarily-checked S t*
shows *cwt S D G t A*
⟨*proof*⟩

lemma *hereditarily-checked-step*:
assumes *source*: *hereditarily-checked S t* **and** *step*: *red S t u*
shows *hereditarily-checked S u*
⟨*proof*⟩

lemma *hereditarily-checked-steps*:
assumes *source*: *hereditarily-checked S t* **and** *steps*: *reds S t u*
shows *hereditarily-checked S u*
⟨*proof*⟩

lemma *checked-PCon-wf*:
checked S (PCon h k as) ⟹ wf-hsig S h
⟨*proof*⟩

lemma *checked-LCon-wf*:

checked S (LCon h k as d) ⟹ wf-hsig S h
⟨*proof*⟩

lemma *checked-HElim-coherent*:
checked S (HElim h C pms lms t) ⟹
typed-coherent-methods S h C pms lms
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *nth-ins-ty-lt*:
assumes *i < n* **and** *n ≤ length G*
shows *ins-ty n A G ! i = G ! i*
⟨*proof*⟩

lemma *nth-ins-ty-eq*:
assumes *n ≤ length G*
shows *ins-ty n A G ! n = A*
⟨*proof*⟩

lemma *nth-ins-ty-ge*:
assumes *n ≤ i* **and** *i < length G*
shows *ins-ty n A G ! Suc i = G ! i*
⟨*proof*⟩

lemma *wt-dimension-mono*:
assumes *deriv*: *wt S D G t A* **and** *sub*: *D ⊆ D′*
shows *wt S D′ G t A*
⟨*proof*⟩

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*
⟨*proof*⟩

11 Term substitution

lemma *nth-ins-ty-gt*:
  assumes $n < i$ **and** $i < Suc\ (length\ G)$ **and** $n \leq length\ G$
  shows $ins\text{-}ty\ n\ A\ G\ !\ i = G\ !\ (i - 1)$
⟨*proof*⟩

theorem *wt-subst-aux*:
  assumes *deriv*: $wt\ S\ D\ H\ t\ B$
    and *ctx*: $H = ins\text{-}ty\ n\ A\ G$
    and *replacement*: $wt\ S\ D\ G\ s\ A$
    and *n*: $n \leq length\ G$
  shows $wt\ S\ D\ G\ (subst\ n\ s\ t)\ B$
  ⟨*proof*⟩

corollary *wt-subst*:
  assumes $wt\ S\ D\ (ins\text{-}ty\ n\ A\ G)\ t\ B$
    and $wt\ S\ D\ G\ s\ A$
    and $n \leq length\ G$
  shows $wt\ S\ D\ G\ (subst\ n\ s\ t)\ B$
  ⟨*proof*⟩

12 Dimension substitution

fun *dim-set* :: $dim \Rightarrow nat\ set$ **where**
  $dim\text{-}set\ D0 = \{\}$
| $dim\text{-}set\ D1 = \{\}$
| $dim\text{-}set\ (DVar\ j) = \{j\}$

definition *dctx-subst* :: $nat \Rightarrow dim \Rightarrow nat\ set \Rightarrow nat\ set$ **where**
  $dctx\text{-}subst\ n\ d\ D = (D - \{n\}) \cup dim\text{-}set\ d$

lemma *dctx-subst-insert-nonzero*:
  assumes $n \neq 0$
  shows $dctx\text{-}subst\ n\ d\ (insert\ 0\ D) = insert\ 0\ (dctx\text{-}subst\ n\ d\ D)$
  ⟨*proof*⟩

lemma *dctx-subst-bound-subset*:
  $insert\ 0\ D \subseteq insert\ 0\ (dctx\text{-}subst\ 0\ d\ D)$
  ⟨*proof*⟩

theorem *wt-dimension-subst*:
  assumes *deriv*: $wt\ S\ D\ G\ t\ A$ **and** *wf*: $wf\text{-}sigenv\ S$
  shows $wt\ S\ (dctx\text{-}subst\ n\ d\ D)\ G\ (dsubst\ S\ n\ d\ t)\ A$
  ⟨*proof*⟩

end
theory *HIT-Preservation*
  imports *HIT-Substitution*

begin

13 Subject reduction

lemma *all-nth-list-update*:
  assumes *all*: $\forall j < \mathit{length}\ \mathit{xs}.\ P\ j\ (\mathit{xs}\ !\ j)$
    and *i*: $i < \mathit{length}\ \mathit{xs}$
    and *replacement*: $P\ i\ x$
  shows $\forall j < \mathit{length}\ (\mathit{xs}[i := x]).\ P\ j\ ((\mathit{xs}[i := x])\ !\ j)$
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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*⟩

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* ⇒ *argspec* ⇒ *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*
  ⟨*proof*⟩

lemma *canonAt-Fun* [*simp*]:
  canonAt S (*Fun A1 A2*) *t* = (∃ *B u*. *t* = *Lam B u*)
  ⟨*proof*⟩

14.2 Normal forms are closed under the relevant subterms

lemma *nform-App1*: *nform S* (*App t u*) ⟹ *nform S t*
  ⟨*proof*⟩

lemma *nform-elim-scrutinee*:
nform S (*HElim h C pms lms t*) ⟹ *nform S t*
⟨*proof*⟩

lemma *nform-PCon-arg*:
assumes *nform S* (*PCon h k as*) **and** *i* < *length as*
shows *nform S* (*as* ! *i*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *progress*:
assumes *cwt S* {} [] *t A*
shows *canonAt S A t* ∨ (∃ *u*. *red S t u*)
⟨*proof*⟩

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*⟩

theorem *canonical-forms-raw*:
assumes *wt S* {} [] *t A* **and** *nform S t*
shows *canonAt S A t*
⟨*proof*⟩

theorem *canonical-forms*:
assumes *cwt S* {} [] *t A* **and** *nform S t*
shows *canonAt S A t*
⟨*proof*⟩

corollary *canonical-forms-hit*:
assumes *cwt S* {} [] *t* (*Hit h*) **and** *nform S t*
shows *canonical S h t*
⟨*proof*⟩

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*⟩

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*⟩

corollary *no-closed-normal-inhabitant*:
assumes *pcons* (*S h*) = []
and *cwt S* {} [] *t* (*Hit h*)
and *nform S t*
shows *False*
⟨*proof*⟩

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*⟩

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*⟩

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*
  ⟨*proof*⟩

lemma *variable-dims-erase* [*simp*]: *variable-dims* (*erase-dims S t*)
  ⟨*proof*⟩

lemma *dclosed0-erase* [*simp*]: *dclosed0 S* (*erase-dims S t*)
⟨*proof*⟩

lemma *erase-dims-idem* [*simp*]:
erase-dims S (*erase-dims S t*) = *erase-dims S t*
⟨*proof*⟩

lemma *variable-dims-dsubst-erase*:
erase-dims S (*dsubst S n* (*DVar j*) (*erase-dims S t*)) =
erase-dims S t
⟨*proof*⟩

lemma *dsubst-zero-erase* [*simp*]:
dsubst S 0 (*DVar j*) (*erase-dims S t*) = *erase-dims S t*
⟨*proof*⟩

lemma *erase-dims-dsubst-variable*:
assumes *variable-dims t*
shows *erase-dims S* (*dsubst S n* (*DVar j*) *t*) = *erase-dims S t*
⟨*proof*⟩

lemma *erase-dims-lift* [*simp*]:
erase-dims S (*lift n t*) = *lift n* (*erase-dims S t*)
⟨*proof*⟩

lemma *erase-dims-subst* [*simp*]:
erase-dims S (*subst n s t*) =
subst n (*erase-dims S s*) (*erase-dims S t*)
⟨*proof*⟩

lemma *erase-dims-apps* [*simp*]:
erase-dims S (*apps t xs*) =
apps (*erase-dims S t*) (*map* (*erase-dims S*) *xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *regular-wt*:
assumes *wt S D G t A*
shows *regular S t*
⟨*proof*⟩

lemma *erase-dims-wt*:

assumes *wt S D G t A*
shows *wt S {1} G (erase-dims S t) A*
⟨*proof*⟩

lemma *regular-lift*:
regular S t ⟹ *regular S (lift n t)*
⟨*proof*⟩

lemma *variable-dims-lift*:
variable-dims t ⟹ *variable-dims (lift n t)*
⟨*proof*⟩

lemma *variable-dims-subst*:
assumes *variable-dims s* **and** *variable-dims t*
shows *variable-dims (subst n s t)*
⟨*proof*⟩

lemma *regular-subst*:
assumes *regular S s* **and** *regular S t*
shows *regular S (subst n s t)*
⟨*proof*⟩

lemma *regular-dsubst-variable*:
assumes *regular S t*
shows *regular S (dsubst S n (DVar j) t)*
⟨*proof*⟩

lemma *regular-apps*:
assumes *regular S t* **and** *list-all (regular S) xs*
shows *regular S (apps t xs)*
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *regular-red*:
assumes *step*: *red S t u* **and** *source*: *regular S t*
shows *regular S u*
⟨*proof*⟩

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)*

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *up-ren-id-apply* [*simp*]: *up-ren* (λ*x*. *x*) *i* = *i*
⟨*proof*⟩

lemma *up-ren-comp*:
up-ren (*f* ∘ *g*) = *up-ren f* ∘ *up-ren g*
⟨*proof*⟩

lemma *vrename-id* [*simp*]: *vrename id t* = *t*
⟨*proof*⟩

lemma *vrename-comp*:
vrename f (*vrename g t*) = *vrename* (*f* ∘ *g*) *t*
⟨*proof*⟩

lemma *erase-dims-vrename* [*simp*]:
erase-dims S (*vrename f t*) = *vrename f* (*erase-dims S t*)
⟨*proof*⟩

lemma *regular-vrename* [*simp*]:
regular S (*vrename f t*) = *regular S t*
⟨*proof*⟩

lemma *variable-dims-vrename* [*simp*]:
variable-dims (*vrename f t*) = *variable-dims t*
⟨*proof*⟩

lemma *up-sub-Var* [*simp*]: *up-sub Var* = *Var*
⟨*proof*⟩

lemma *psubst-Var* [*simp*]: *psubst Var t* = *t*
⟨*proof*⟩

lemma *erase-dims-up-sub*:
(λ*i*. *erase-dims S* (*up-sub s i*)) =
up-sub (λ*i*. *erase-dims S* (*s i*))
⟨*proof*⟩

lemma *erase-dims-psubst*:
erase-dims S (*psubst s t*) =
psubst (λ*i*. *erase-dims S* (*s i*)) (*erase-dims S t*)
⟨*proof*⟩

lemma *regular-up-sub*:
assumes ⋀*i*. *regular S* (*s i*)
shows *regular S* (*up-sub s i*)

⟨*proof*⟩

lemma *variable-dims-up-sub*:
assumes $\bigwedge i$. *variable-dims* (*s i*)
shows *variable-dims* (*up-sub s i*)
⟨*proof*⟩

lemma *regular-psubst*:
assumes *reg*: *regular S t*
and *env*: $\bigwedge i$. *regular S* (*s i*)
shows *regular S* (*psubst s t*)
⟨*proof*⟩

lemma *variable-dims-psubst*:
assumes *dims*: *variable-dims t*
and *env*: $\bigwedge i$. *variable-dims* (*s i*)
shows *variable-dims* (*psubst s t*)
⟨*proof*⟩

lemma *Ren-id* [*simp*]: *Ren n n id*
⟨*proof*⟩

lemma *Ren-comp*:
assumes *Ren n m f* **and** *Ren m k g*
shows *Ren n k* (*g* ∘ *f*)
⟨*proof*⟩

lemma *Ren-up*:
assumes *Ren n m f*
shows *Ren* (*Suc n*) (*Suc m*) (*up-ren f*)
⟨*proof*⟩

lemma *Ren-Suc*: *Ren n* (*Suc n*) *Suc*
⟨*proof*⟩

definition *shift-ren* :: *nat* ⇒ *nat* ⇒ *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*)
⟨*proof*⟩

lemma *lift-as-vrename*:
lift n t = *vrename* (*shift-ren n*) *t*
⟨*proof*⟩

lemma *vrename-lift0*:
vrename (*up-ren f*) (*lift 0 t*) = *lift 0* (*vrename f t*)
⟨*proof*⟩

lemma *up-sub-vrename*:
(λ*i*. *vrename* (*up-ren f*) (*up-sub s i*)) =
up-sub (λ*i*. *vrename f* (*s i*))
⟨*proof*⟩

lemma *vrename-psubst*:
vrename f (*psubst s t*) =
psubst (λ*i*. *vrename f* (*s i*)) *t*
⟨*proof*⟩

lemma *up-sub-comp*:
up-sub s ∘ *up-ren f* = *up-sub* (*s* ∘ *f*)
⟨*proof*⟩

lemma *psubst-vrename*:
psubst s (*vrename f t*) = *psubst* (*s* ∘ *f*) *t*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *subst-as-psubst*:
subst n s t = *psubst* (*single-sub n s*) *t*
⟨*proof*⟩

lemma *vrename-subst0*:
vrename f (*subst 0 s t*) =
subst 0 (*vrename f s*) (*vrename* (*up-ren f*) *t*)
⟨*proof*⟩

lemma *vrename-apps* [*simp*]:
vrename f (*apps t xs*) = *apps* (*vrename f t*) (*map* (*vrename f*) *xs*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *dclosed0-vrename*:
assumes *variable-dims t*
shows *dclosed0 S (vrename f t)* ⟷ *dclosed0 S t*
⟨*proof*⟩

lemma *red-vrename*:
assumes *step*: *red S t u* **and** *source*: *regular S t*
shows *red S (vrename f t) (vrename f u)*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *erased-red*:
assumes *step*: *red S t u* **and** *reg*: *regular S t* **and** *fixed*: *erased S t*
shows *erased S u*
⟨*proof*⟩

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 $\Longrightarrow$ *neutral* (*vrename f t*)
⟨*proof*⟩

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

lemma *hcomp-step*:
hcomp S h t $\Longrightarrow$ *red S t u* $\Longrightarrow$ *hcomp S h u*
⟨*proof*⟩

lemma *hcomp-SN*:
hcomp S h t $\Longrightarrow$ *SN S t*
⟨*proof*⟩

lemma *hcomp-Var*: *hcomp S h* (*Var i*)
⟨*proof*⟩

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

inductive *SN-list* :: *sigenv* ⇒ *tm list* ⇒ *bool*
for *S* :: *sigenv*
where
SNL: ($\bigwedge$*ys*. *list-red S xs ys* $\Longrightarrow$ *SN-list S ys*) $\Longrightarrow$
SN-list S xs

lemma *SN-list-step*:
SN-list S xs $\Longrightarrow$ *list-red S xs ys* $\Longrightarrow$ *SN-list S ys*
⟨*proof*⟩

lemma *SN-list-Cons*:
assumes *head*: *SN S a* **and** *tail*: *SN-list S xs*
shows *SN-list S* (*a* # *xs*)
⟨*proof*⟩

lemma *SN-list-Nil*: *SN-list S* []
⟨*proof*⟩

lemma *SN-list-all*:
assumes *list-all* (*SN S*) *xs*
shows *SN-list S xs*
⟨*proof*⟩

lemma *SN-lists-induct* [*consumes 2*, *case-names step*]:
assumes *xs*: *SN-list S xs* **and** *ys*: *SN-list S ys*
and *step*: $\bigwedge$*xs ys*.
($\bigwedge$*xs'*. *list-red S xs xs'* $\Longrightarrow$ *P xs' ys*) $\Longrightarrow$
($\bigwedge$*ys'*. *list-red S ys ys'* $\Longrightarrow$ *P xs ys'*) $\Longrightarrow$
P xs ys
shows *P xs ys*
⟨*proof*⟩

lemma *list-red-append*:
red S a b $\Longrightarrow$
list-red S (*xs* @ *a* # *ys*) (*xs* @ *b* # *ys*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*) =

$(hcomp\ S\ (argHit\ h\ s)\ a \wedge hargs\ S\ h\ ss\ as)$

lemma *hargs-length*:
$hargs\ S\ h\ ss\ as \Longrightarrow length\ as = length\ ss$
⟨*proof*⟩

lemma *hargs-nth*:
assumes $hargs\ S\ h\ ss\ as$ **and** $i < length\ as$
shows $hcomp\ S\ (argHit\ h\ (ss\ !\ i))\ (as\ !\ i)$
⟨*proof*⟩

lemma *hargsI*:
assumes $length\ as = length\ ss$
and $\bigwedge i.\ i < length\ as \Longrightarrow$
$hcomp\ S\ (argHit\ h\ (ss\ !\ i))\ (as\ !\ i)$
shows $hargs\ S\ h\ ss\ as$
⟨*proof*⟩

lemma *hargs-SN*:
assumes $hargs\ S\ h\ ss\ as$
shows $list\text{-}all\ (SN\ S)\ as$
⟨*proof*⟩

lemma *hargs-list-red*:
assumes *step*: $list\text{-}red\ S\ as\ bs$ **and** *args*: $hargs\ S\ h\ ss\ as$
shows $hargs\ S\ h\ ss\ bs$
⟨*proof*⟩

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*⟩

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*⟩

lemma *hcomp-vrename*:
assumes *comp*: $hcomp\ S\ h\ t$ **and** *reg*: $regular\ S\ t$
shows $hcomp\ S\ h\ (vrename\ f\ t)$
⟨*proof*⟩

lemma *SN-vrename*:
assumes *sn*: $SN\ S\ t$ **and** *reg*: $regular\ S\ t$
shows $SN\ S\ (vrename\ f\ t)$
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *RC-erased*:
  RC S A t ⟹ *erased S t*
  ⟨*proof*⟩

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*⟩

lemma *RC-SN*:
  RC S A t ⟹ *SN S t*
  ⟨*proof*⟩

lemma *RC-red*:
  RC S A t ⟹ *red S t u* ⟹ *RC S A u*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*:
$KR\ S\ n\ A\ t \Longrightarrow RC\ S\ A\ t$
⟨*proof*⟩

lemma *KR-regular*:
$KR\ S\ n\ A\ t \Longrightarrow regular\ S\ t$
⟨*proof*⟩

lemma *KR-erased*:
$KR\ S\ n\ A\ t \Longrightarrow erased\ S\ t$
⟨*proof*⟩

lemma *KR-SN*:
$KR\ S\ n\ A\ t \Longrightarrow SN\ S\ t$
⟨*proof*⟩

lemma *KR-Hit-hcomp*:
$KR\ S\ n\ (Hit\ h)\ t \Longrightarrow hcomp\ S\ h\ t$
⟨*proof*⟩

lemma *KR-App*:
assumes $KR\ S\ n\ (Fun\ A\ B)\ t$ **and** $KR\ S\ n\ A\ u$
shows $KR\ S\ n\ B\ (App\ t\ u)$
⟨*proof*⟩

lemma *KR-vrename*:
assumes $KR\ S\ n\ A\ t$ **and** $Ren\ n\ m\ f$
shows $KR\ S\ m\ A\ (vrename\ f\ t)$
⟨*proof*⟩

lemma *KR-red*:
assumes *rel*: $KR\ S\ n\ A\ t$ **and** *step*: $red\ S\ t\ u$
shows $KR\ S\ n\ A\ u$
⟨*proof*⟩

lemma *KR-reds*:
assumes $KR\ S\ n\ A\ t$ **and** $reds\ S\ t\ u$
shows $KR\ S\ n\ A\ u$
⟨*proof*⟩

lemma *KR-neutral*:
assumes *reg*: $regular\ S\ t$ **and** *fixed*: $erased\ S\ t$ **and** *neu*: $neutral\ t$
and *reducts*: $\bigwedge m\ f\ u.\ Ren\ n\ m\ f \Longrightarrow$
$red\ S\ (vrename\ f\ t)\ u \Longrightarrow RC\ S\ A\ u$
shows $KR\ S\ n\ A\ t$
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *KC-erased*:
  KC S n A t ⟹ *erased S t*
  ⟨*proof*⟩

lemma *KC-SN*:
  KC S n A t ⟹ *SN S t*
  ⟨*proof*⟩

lemma *KC-Hit-hcomp*:
  KC S n (*Hit h*) *t* ⟹ *hcomp S h t*
  ⟨*proof*⟩

lemma *erased-vrename*:
  erased S t ⟹ *erased S* (*vrename f t*)
  ⟨*proof*⟩

lemma *KC-vrename*:
  assumes *rel*: *KC S n A t* **and** *ren*: *Ren n m f*
  shows *KC S m A* (*vrename f t*)
  ⟨*proof*⟩

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*⟩

lemma *KC-red*:
  assumes *rel*: *KC S n A t* **and** *step*: *red S t u*
  shows *KC S n A u*
  ⟨*proof*⟩

lemma *KC-reds*:
  assumes *KC S n A t* **and** *reds S t u*
  shows *KC S n A u*
  ⟨*proof*⟩

lemma *KC-neutral*:
assumes *neu*: *neutral t* **and** *reg*: *regular S t* **and** *fixed*: *erased S t*
and *reducts*: $\bigwedge u.$ *red S t u* $\Longrightarrow$ *KC S n A u*
shows *KC S n A t*
⟨*proof*⟩

lemma *KC-Var*: *KC S n A* (*Var i*)
⟨*proof*⟩

15.6 Substitution and reducibility

lemma *subst-lift-same* [*simp*]:
subst n s (*lift n t*) = *t*
⟨*proof*⟩

lemma *shift-ren-zero* [*simp*]: *shift-ren 0* = *Suc*
⟨*proof*⟩

lemma *lift-zero-as-vrename*: *lift 0 t* = *vrename Suc t*
⟨*proof*⟩

lemma *psubst-lift0*:
psubst (*up-sub s*) (*lift 0 t*) = *lift 0* (*psubst s t*)
⟨*proof*⟩

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

lemma *psubst-comp*:
psubst r (*psubst s t*) =
psubst (λ*i*. *psubst r* (*s i*)) *t*
⟨*proof*⟩

lemma *subst0-psubst*:
subst 0 (*psubst s u*) (*psubst* (*up-sub s*) *t*) =
psubst s (*subst 0 u t*)
⟨*proof*⟩

lemma *subst0-psubst-up*:
subst 0 x (*psubst* (*up-sub s*) *t*) =
psubst (*ext-sub x s*) *t*
⟨*proof*⟩

lemma *psubst-apps* [*simp*]:
psubst s (*apps t xs*) = *apps* (*psubst s t*) (*map* (*psubst s*) *xs*)
⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *erased-dsubst-zero*:
  assumes *erased S t*
  shows *dsubst S 0* (*DVar j*) *t* = *t*
⟨*proof*⟩

lemma *erased-up-sub*:
  assumes ⋀*i*. *erased S* (*s i*)
  shows *erased S* (*up-sub s i*)
  ⟨*proof*⟩

lemma *psubst-dsubst-zero*:
  assumes *dims*: *variable-dims t*
    and *env*: ⋀*i*. *erased S* (*s i*)
  shows *psubst s* (*dsubst S 0* (*DVar j*) *t*) =
    dsubst S 0 (*DVar j*) (*psubst s t*)
  ⟨*proof*⟩

lemma *dclosed0-psubst*:
  assumes *dims*: *variable-dims t*
    and *closed*: *dclosed0 S t*
    and *env*: ⋀*i*. *erased S* (*s i*)
  shows *dclosed0 S* (*psubst s t*)
⟨*proof*⟩

lemma *red-psubst*:
  assumes *step*: *red S t u* **and** *reg*: *regular S t*
    and *env*: ⋀*i*. *regular S* (*s i*)
    and *fixed*: ⋀*i*. *erased S* (*s i*)
  shows *red S* (*psubst s t*) (*psubst s u*)
  ⟨*proof*⟩

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*⟩

lemma *red-LamE*:
  assumes *red S* (*Lam A t*) *u*
  obtains *t'* **where** *u* = *Lam A t'* **and** *red S t t'*
  ⟨*proof*⟩

lemma *SN-LamI*:

assumes *SN S t*
shows *SN S* (*Lam A t*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *kc-args-regular*:
assumes *kc-args S n h ss as*
shows *list-all* (*regular S*) *as*

⟨*proof*⟩

lemma *kc-args-erased*:
assumes *kc-args S n h ss as*
shows *list-all* (*erased S*) *as*
⟨*proof*⟩

lemma *kc-args-hargs*:
assumes *kc-args S n h ss as*
shows *hargs S h ss as*
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *k-methods-regular*:
k-methods S n h C css ms $\Longrightarrow$ *list-all* (*regular S*) *ms*
⟨*proof*⟩

lemma *k-methods-erased*:
k-methods S n h C css ms $\Longrightarrow$ *list-all* (*erased S*) *ms*
⟨*proof*⟩

lemma *k-methods-SN*:
k-methods S n h C css ms $\Longrightarrow$ *list-all* (*SN S*) *ms*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*: $\bigwedge a$. *KC S n* (*Hit h*) *a* $\Longrightarrow$
KC S n C (*HElim h C pms lms a*)
shows *KC S n C* (*apps m* (*elimArgs h C pms lms ss as*))
⟨*proof*⟩

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*: $\bigwedge i$. $i <$ *length as* $\Longrightarrow$ *ss* ! i = *Rec* $\Longrightarrow$
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*))
⟨*proof*⟩

lemma *regular-PCon-arg*:
regular S (*PCon h k as*) $\Longrightarrow$ $i <$ *length as* $\Longrightarrow$
regular S (*as* ! i)
⟨*proof*⟩

lemma *regular-LCon-arg*:
regular S (*LCon h k as d*) $\Longrightarrow$ $i <$ *length as* $\Longrightarrow$
regular S (*as* ! i)
⟨*proof*⟩

lemma *erased-PCon-arg*:
assumes *erased S* (*PCon h k as*) **and** $i <$ *length as*
shows *erased S* (*as* ! i)
⟨*proof*⟩

lemma *erased-LCon-arg*:
assumes *erased S* (*LCon h k as d*) **and** $i <$ *length as*
shows *erased S* (*as* ! *i*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *k-env-vrename*:
assumes *env*: *k-env S n G s* **and** *ren*: *Ren n m f*
shows *k-env S m G* (λ*i*. *vrename f* (*s i*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *k-env-Var*: *k-env S* (*length G*) *G Var*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *SN-erase-dims-reflect-regular*:
assumes *regular S t* **and** *SN S* (*erase-dims S t*)
shows *SN S t*
⟨*proof*⟩

theorem *wt-strong-normalization*:
assumes *typed*: *wt S D G t A*
shows *SN S t*
⟨*proof*⟩

theorem *cwt-strong-normalization*:
assumes *wf-sigenv S* **and** *cwt S D G t A*
shows *SN S t*
⟨*proof*⟩

corollary *strong-normalization*:
assumes *wf-sigenv S* **and** *cwt S D G t A*
shows *SN S t*
⟨*proof*⟩

theorem *SN-reaches-normal-form*:
assumes *SN S t*
shows ∃ *u*. *reds S t u* ∧ *nform S u*
⟨*proof*⟩

corollary *SN-normalizes*:
assumes *SN S t*
shows *normalizes S t*
⟨*proof*⟩

corollary *wt-reaches-normal-form*:
assumes *wt S D G t A*

shows $\exists\, u.\ reds\ S\ t\ u \wedge nform\ S\ u$
⟨*proof*⟩

corollary *cwt-reaches-normal-form*:
assumes *wf-sigenv S* **and** *cwt S D G t A*
shows $\exists\, u.\ reds\ S\ t\ u \wedge nform\ S\ u$
⟨*proof*⟩

corollary *cwt-normalizes*:
assumes *wf-sigenv S* **and** *cwt S D G t A*
shows *normalizes S t*
⟨*proof*⟩

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 $\exists\, u.\ reds\ S\ t\ u \wedge nform\ S\ u \wedge canonAt\ S\ A\ u$
⟨*proof*⟩

corollary *checked-hit-has-canonical-normal-form*:
assumes *wf*: *wf-sigenv S* **and** *typed*: *cwt S* {} [] *t* (*Hit h*)
shows $\exists\, u.\ reds\ S\ t\ u \wedge nform\ S\ u \wedge canonical\ S\ h\ u$
⟨*proof*⟩

corollary *empty-signature-consistency*:
assumes *wf*: *wf-sigenv S* **and** *empty*: *pcons* (*S h*) = []
shows $\neg\ (\exists\, t.\ cwt\ S\ \{\}\ []\ t\ (Hit\ h))$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *capture-substitute-then-instantiate*:
dsubst S 0 (*DVar 1*)
(*subst 0 capture-arg capture-body*) = *capture-right*
⟨*proof*⟩

lemma *capture-instantiate-then-substitute*:
subst 0 (*dsubst S 0* (*DVar 1*) *capture-arg*)
(*dsubst S 0* (*DVar 1*) *capture-body*) = *capture-left*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *reds-App*:
assumes *reds S t t'* **and** *reds S u u'*
shows *reds S* (*App t u*) (*App t' u'*)
⟨*proof*⟩

lemma *reds-Lam*:
reds S t t' ⟹ *reds S* (*Lam A t*) (*Lam A t'*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list-reds-length*:
list-reds S xs ys ⟹ *length xs* = *length ys*
⟨*proof*⟩

lemma *list-reds-nth*:
assumes *list-reds S xs ys* **and** *i* < *length xs*
shows *reds S* (*xs* ! *i*) (*ys* ! *i*)
⟨*proof*⟩

lemma *reds-PCon-lists*:
assumes *list-reds S xs ys*
shows *reds S* (*PCon h k* (*pre* @ *xs*)) (*PCon h k* (*pre* @ *ys*))
⟨*proof*⟩

lemma *reds-PCon*:
list-reds S xs ys ⟹
reds S (*PCon h k xs*) (*PCon h k ys*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *reds-LCon*:
list-reds S xs ys ⟹
reds S (*LCon h k xs d*) (*LCon h k ys d*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *reds-HElim-pms*:
list-reds S pms pms' ⟹
reds S (*HElim h C pms lms t*) (*HElim h C pms' lms t*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *reds-HElim-lms*:
list-reds S lms lms' ⟹
reds S (*HElim h C pms lms t*) (*HElim h C pms lms' t*)
⟨*proof*⟩

lemma *reds-HElim-scrutinee*:
reds S t t' ⟹
reds S (*HElim h C pms lms t*) (*HElim h C pms lms t'*)
⟨*proof*⟩

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*⟩

lemma *reds-apps*:
assumes *reds S t t'* **and** *list-reds S xs ys*
shows *reds S* (*apps t xs*) (*apps t' ys*)
⟨*proof*⟩

lemma *list-reds-trans*:
assumes *list-reds S xs ys* **and** *list-reds S ys zs*
shows *list-reds S xs zs*
⟨*proof*⟩

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′*)
⟨*proof*⟩

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*)
⟨*proof*⟩

17.3 Dimension-safe reduction

lemma *dclosed0-lift*:
assumes *variable-dims t* **and** *dclosed0 S t*
shows *dclosed0 S* (*lift n t*)
⟨*proof*⟩

lemma *dclosed0-up-sub*:
assumes ⋀*i*. *dclosed0 S* (*s i*)
and ⋀*i*. *variable-dims* (*s i*)
shows *dclosed0 S* (*up-sub s i*)
⟨*proof*⟩

lemma *psubst-dsubst-zero-dclosed*:
assumes *dims*: *variable-dims t*
and *env-closed*: ⋀*i*. *dclosed0 S* (*s i*)
and *env-dims*: ⋀*i*. *variable-dims* (*s i*)
shows *psubst s* (*dsubst S 0* (*DVar j*) *t*) =
dsubst S 0 (*DVar j*) (*psubst s t*)
⟨*proof*⟩

lemma *dclosed0-psubst-closed*:
assumes *dims*: *variable-dims t* **and** *closed*: *dclosed0 S t*
and *env-closed*: ⋀*i*. *dclosed0 S* (*s i*)
and *env-dims*: ⋀*i*. *variable-dims* (*s i*)
shows *dclosed0 S* (*psubst s t*)
⟨*proof*⟩

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*⟩

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*)

⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *dclosed0-apps*:
assumes *dclosed0 S t* **and** *list-all* (*dclosed0 S*) *xs*
shows *dclosed0 S* (*apps t xs*)
⟨*proof*⟩

lemma *dsubst-variable-closed*:
assumes *reg*: *regular S t* **and** $j \neq 0$
shows *dclosed0 S* (*dsubst S 0* (*DVar j*) *t*)
⟨*proof*⟩

lemma *dclosed0-red*:
assumes *step*: *red S t u* **and** *reg*: *regular S t*
and *closed*: *dclosed0 S t*
shows *dclosed0 S u*
⟨*proof*⟩

lemma *dsubst-same-regular*:
assumes *regular S t*
shows *dsubst S 0* (*DVar 0*) *t* = *t*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

17.4 Local peaks

lemma *red-as-reds*:
red S t u ⟹ *reds S t u*
⟨*proof*⟩

lemma *list-red-imp-list-reds*:
assumes *list-red S xs ys*
shows *list-reds S xs ys*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*⟩

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'*
⟨*proof*⟩

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*⟩

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*⟩

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*

⟨*proof*⟩

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*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *joinable-App-left*:
assumes *joinable S t u*
shows *joinable S* (*App t a*) (*App u a*)
⟨*proof*⟩

lemma *joinable-App-right*:
assumes *joinable S t u*
shows *joinable S* (*App a t*) (*App a u*)
⟨*proof*⟩

lemma *joinable-Lam*:
assumes *joinable S t u*
shows *joinable S* (*Lam A t*) (*Lam A u*)
⟨*proof*⟩

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*⟩

lemma *red-HElim-local-confluence*:
assumes *reg*: *regular S* (*HElim h C pms lms t*)
and *local-pms*: $\bigwedge m\ u\ v.\ m \in set\ pms \Longrightarrow$
red S m u $\Longrightarrow$ *red S m v* $\Longrightarrow$ *joinable S u v*
and *local-lms*: $\bigwedge m\ u\ v.\ m \in set\ lms \Longrightarrow$
red S m u $\Longrightarrow$ *red S m v* $\Longrightarrow$ *joinable S u v*
and *local-t*: $\bigwedge u\ v.\ red\ S\ t\ u \Longrightarrow red\ S\ t\ v \Longrightarrow$
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*⟩

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*
⟨*proof*⟩

lemma *reds-first-step*:
assumes *reds S t u*
shows $t = u \vee (\exists\, v.\ red\ S\ t\ v \wedge reds\ S\ v\ u)$
⟨*proof*⟩

lemma *nform-reds-eq*:
assumes *normal*: *nform S t* **and** *steps*: *reds S t u*
shows $u = t$
⟨*proof*⟩

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 $u = v$
⟨*proof*⟩

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 $\exists\, w.\ reds\ S\ u\ w \wedge reds\ S\ v\ w$

⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

corollary *canonical-nf-reachable*:
wt S D G t A ⟹ *reds S t* (*canonical-nf S t*)
⟨*proof*⟩

corollary *canonical-nf-normal*:
wt S D G t A ⟹ *nform S* (*canonical-nf S t*)
⟨*proof*⟩

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*
⟨*proof*⟩

corollary *canonical-nf-of-normal*:
assumes *typed*: *wt S D G t A* **and** *normal*: *nform S t*
shows *canonical-nf S t* = *t*
⟨*proof*⟩

lemma *regular-reds*:
assumes *reds S t u* **and** *regular S t*
shows *regular S u*
⟨*proof*⟩

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*⟩

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 obli-

gations. 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 (λi. 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
  ⟨proof⟩

lemma open-args-nth:
```

assumes $i < \mathit{length}\ \mathit{ss}$
shows $\mathit{open\text{-}args}\ G\ \mathit{ss}\ !\ i = \mathit{Var}\ (\mathit{length}\ G + i)$
⟨*proof*⟩

lemma *open-inst-prefix*:
assumes $i < \mathit{length}\ G$
shows $\mathit{open\text{-}inst}\ G\ \mathit{as}\ i = \mathit{Var}\ i$
⟨*proof*⟩

lemma *psubst-open-args*:
assumes $\mathit{length}\ \mathit{as} = \mathit{length}\ \mathit{ss}$
shows $\mathit{map}\ (\mathit{psubst}\ (\mathit{open\text{-}inst}\ G\ \mathit{as}))\ (\mathit{open\text{-}args}\ G\ \mathit{ss}) = \mathit{as}$
⟨*proof*⟩

lemma *up-sub-prefix*:
assumes $\forall\, i < \mathit{length}\ G.\ s\ i = \mathit{Var}\ i$
shows $\forall\, i < \mathit{length}\ (A\ \#\ G).\ \mathit{up\text{-}sub}\ s\ i = \mathit{Var}\ i$
⟨*proof*⟩

theorem *wt-context-append*:
assumes $\mathit{wt}\ S\ D\ G\ t\ A$
shows $\mathit{wt}\ S\ D\ (G\ @\ H)\ t\ A$
⟨*proof*⟩

lemma *open-args-typed*:
assumes $i < \mathit{length}\ \mathit{ss}$
shows $\mathit{wt}\ S\ D\ (\mathit{open\text{-}context}\ G\ h\ \mathit{ss})$
$(\mathit{open\text{-}args}\ G\ \mathit{ss}\ !\ i)\ (\mathit{argTy}\ h\ (\mathit{ss}\ !\ i))$
⟨*proof*⟩

lemma *psubst-identity-wt*:
assumes *typed*: $\mathit{wt}\ S\ D\ G\ t\ A$
and *fixed*: $\forall\, i < \mathit{length}\ G.\ s\ i = \mathit{Var}\ i$
shows $\mathit{psubst}\ s\ t = t$
⟨*proof*⟩

lemma *open-inst-regular*:
assumes *args*: $\forall\, i < \mathit{length}\ \mathit{as}.\ \exists\, A.\ \mathit{wt}\ S\ D\ H\ (\mathit{as}\ !\ i)\ A$
shows $\mathit{regular}\ S\ (\mathit{open\text{-}inst}\ G\ \mathit{as}\ i)$
⟨*proof*⟩

lemma *open-inst-dclosed0*:
assumes *zero*: $0 \notin D$
and *args*: $\forall\, i < \mathit{length}\ \mathit{as}.\ \exists\, A.\ \mathit{wt}\ S\ D\ H\ (\mathit{as}\ !\ i)\ A$
shows $\mathit{dclosed0}\ S\ (\mathit{open\text{-}inst}\ G\ \mathit{as}\ i)$
⟨*proof*⟩

lemma *reds-psubst-dclosed*:
assumes *steps*: $\mathit{reds}\ S\ t\ u$

and *reg*: *regular S t*
and *env-reg*: $\bigwedge i$. *regular S* (*s i*)
and *env-closed*: $\bigwedge i$. *dclosed0 S* (*s i*)
shows *reds S* (*psubst s t*) (*psubst s u*)
⟨*proof*⟩

lemma *joinable-psubst-dclosed*:
assumes *join*: *joinable S t u*
and *treg*: *regular S t*
and *ureg*: *regular S u*
and *env-reg*: $\bigwedge i$. *regular S* (*s i*)
and *env-closed*: $\bigwedge i$. *dclosed0 S* (*s i*)
shows *joinable S* (*psubst s t*) (*psubst s u*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *typed-list-regular*:
assumes $\forall\, i < \mathit{length}\ \mathit{xs}.\ \exists\, A.\ \mathit{wt}\ S\ D\ G\ (\mathit{xs}\ !\ i)\ A$
shows *list-all* (*regular* S) *xs*
⟨*proof*⟩

lemma *map-psubst-identity-wt*:
assumes *typed*: $\forall\, i < \mathit{length}\ \mathit{xs}.\ \exists\, A.\ \mathit{wt}\ S\ D\ G\ (\mathit{xs}\ !\ i)\ A$
and *fixed*: $\forall\, i < \mathit{length}\ G.\ s\ i = \mathit{Var}\ i$
shows *map* (*psubst* s) $\mathit{xs} = \mathit{xs}$
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *open-face-typed*:
assumes *base*: *boundary-check-base S D G h C pms lms*
and k: $k < \mathit{length}\ (\mathit{lcons}\ (S\ h))$
and *face*: $d = D0 \vee d = D1$
shows *wt S D* (*open-context G h* (*lcons* (*S h*) ! *k*))
(*open-face S G h C pms lms k d*) *C*
⟨*proof*⟩

theorem *open-endpoint-typed*:
assumes *base*: *boundary-check-base S D G h C pms lms*
and k: $k < \mathit{length}\ (\mathit{lcons}\ (S\ h))$
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*⟩

lemma *open-joinable-iff-canonical-nf-eq*:
assumes *base*: *boundary-check-base S D G h C pms lms*
and k: $k < \mathit{length}\ (\mathit{lcons}\ (S\ h))$
and *endpoint*: *wf-endpt S h* (*lcons* (*S h*) ! *k*) *e* (*Hit h*)
and *face*: $d = D0 \vee d = D1$
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*)
⟨*proof*⟩

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*⟩

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*⟩

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*
  ⟨*proof*⟩

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*))
  ⟨*proof*⟩

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*
⟨*proof*⟩

20.1 Well-formed signatures

lemma *demo-wf-circle*: *wf-hsig demo 0*
⟨*proof*⟩

lemma *demo-wf-interval*: *wf-hsig demo 1*
⟨*proof*⟩

lemma *demo-wf-nat*: *wf-hsig demo 2*
⟨*proof*⟩

lemma *demo-wf-empty*: *wf-hsig demo 3*
⟨*proof*⟩

lemma *demo-wf-unit*: *wf-hsig demo 4*
⟨*proof*⟩

lemma *demo-wf-suspension*: *wf-hsig demo 5*
⟨*proof*⟩

lemma *demo-wf-truncation*: *wf-hsig demo 6*
⟨*proof*⟩

lemma *demo-ge-seven*:
assumes $7 \leq h$
shows *demo h* = *emptySig*
⟨*proof*⟩

theorem *demo-wf*: *wf-sigenv demo*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *loop-boundary-one*: *mkL demo 0 0* [] *D1* = *base*
⟨*proof*⟩

theorem *loop-dsubst-zero*: *dsubst demo j D0* (*loop j*) = *base*
⟨*proof*⟩

theorem *loop-dsubst-one*: *dsubst demo j D1* (*loop j*) = *base*
⟨*proof*⟩

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
  ⟨*proof*⟩

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 []
  ⟨*proof*⟩

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
  ⟨*proof*⟩

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]
  ⟨*proof*⟩

theorem *interval-identity-open-boundary-check*:
  assumes *0 ∉ D*
  shows *open-boundary-check demo D [] 1 (Hit 1)*
    [izero, ione] [seg 0]
⟨*proof*⟩

theorem *interval-identity-normal-boundary-check*:
  assumes *0 ∉ D*
  shows *normal-boundary-check demo D [] 1 (Hit 1)*
    [izero, ione] [seg 0]
  ⟨*proof*⟩

theorem *interval-identity-typed-coherent*:
  typed-coherent-methods demo 1 (Hit 1) [izero, ione] [seg 0]
  ⟨*proof*⟩

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*⟩

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*⟩

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*)
  ⟨*proof*⟩

lemma *seg-locally-checked*: *checked demo* (*seg j*)
  ⟨*proof*⟩

lemma *seg-normal*: *nform demo* (*seg j*)
  ⟨*proof*⟩

lemma *interval-point-method-no-step*:
  assumes [*izero*, *ione*] = *xs* @ *m* # *ys* **and** *red demo m m′*
  shows *False*
⟨*proof*⟩

lemma *interval-path-method-no-step*:
  assumes [*seg 0*] = *xs* @ *m* # *ys* **and** *red demo m m′*
  shows *False*
⟨*proof*⟩

lemma *interval-identity-red*:
  assumes *red demo* (*interval-id j*) *u*
  shows *u* = *seg j*
⟨*proof*⟩

lemma *interval-identity-reducts*:
  assumes *reds demo* (*interval-id j*) *u*
  shows *u* = *interval-id j* ∨ *u* = *seg j*
  ⟨*proof*⟩

lemma *interval-identity-hereditarily-checked*:
  hereditarily-checked demo (*interval-id j*)
  ⟨*proof*⟩

theorem *interval-identity-checked*:
  assumes *j*: $j \in D$ **and** *zero*: $0 \notin D$
  shows *cwt demo D* []
    (*interval-id j*) (*Hit 1*)
⟨*proof*⟩

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
  ⟨*proof*⟩

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*⟩

theorem *circle-rec-loop*:
  red demo (*HElim 0 C* [*b*] [*l*] (*loop j*)) (*dsubst demo 0* (*DVar j*) *l*)
⟨*proof*⟩

theorem *interval-rec-seg*:
  red demo (*HElim 1 C* [*u*, *v*] [*p*] (*seg j*)) (*dsubst demo 0* (*DVar j*) *p*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

20.5 Typing

theorem *base-typed*: *wt demo D* [] *base* (*Hit 0*)
⟨*proof*⟩

theorem *loop-typed*:
assumes $j \in D$
shows *wt demo D* [] (*loop j*) (*Hit 0*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

theorem *loop-nform*: *nform demo* (*loop j*)
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *base-canonical*: *canonical demo 0 base*
⟨*proof*⟩

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 Proceedings 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.