Introduction to Higher Inductive Types

Benedikt Ahrens

Institut de Recherche en Informatique de Toulouse

2013–11–21

Outline

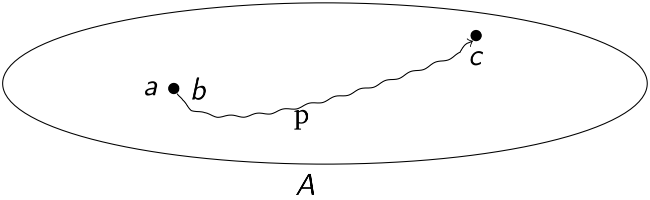
1. Preliminaries: reminders and some more logic
2. Inductive Types
3. Higher Inductive Types
   - First example: Interval
   - The circle and dependent elimination
   - Some more examples of HITs
   - Implementation in Coq

- In the first two parts pure MLTT
- In third part we use Univalence Axiom for some results
- but HITs make sense without UA

Outline

Reminder: two different equalities in MLTT

- $\emptyset \vdash A$
- $\emptyset \vdash a \equiv b : A$ convertibility
- $\emptyset \vdash p : \mathtt{Id}(b, c)$ homotopy



Reminder: Homotopy levels

Homotopy levels: the complete picture

$$\mathsf{isContr}(A) :\equiv \sum_{(a:A)} \prod_{(x:A)} \mathtt{Id}(x, a)$$

$$\mathsf{isProp}(A) :\equiv \prod_{x,y:A} \mathsf{isContr}(\mathtt{Id}(x, y))$$

$$\mathsf{isSet}(A) :\equiv \prod_{x,y:A} \mathsf{isProp}(\mathtt{Id}(x, y))$$

$$\vdots$$

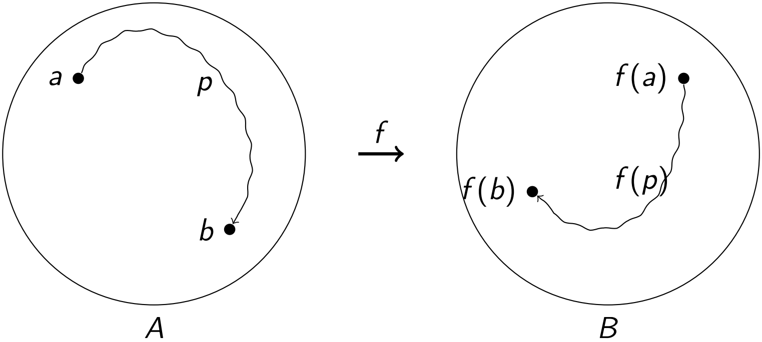
$$\mathsf{isofhlevel}_{n+1}(A) :\equiv \prod_{x,y:A} \mathsf{isofhlevel}_n(\mathtt{Id}(x, y))$$

But we will not need the higher levels.

Reminder: In MLTT, functions are functors

Map on paths: A function $f : A \to B$

- maps points of A to points in B
- maps paths in A to paths in B
- `f_equal: (a ⇝ b) -> (f(a) ⇝ f(b))` in Coq



Reminder: The Univalence Axiom

Definition (From paths to isomorphisms)

$$\begin{aligned} \mathsf{idtoiso}_{A,B} : \mathtt{Id}(A,B) &\to \mathsf{Iso}(A,B) \\ \mathtt{refl}_A &\mapsto (x \mapsto x, p) \end{aligned}$$

Univalence Axiom

$$\mathsf{univalence} : \prod_{A\ B:\mathscr{U}} \mathsf{isIso}(\mathsf{idtoiso}_{A,B})$$

In particular, Univalence gives a map backwards:

$$\mathsf{isotoid}_{A,B} : \mathsf{Iso}(A,B) \to \mathtt{Id}(A,B)$$

Some more logic: defining True

True

1. $\emptyset \vdash \texttt{True}$

2. $\emptyset \vdash \texttt{I} : \texttt{True}$

3. $$\frac{x : \texttt{True} \vdash C(x) \qquad d_I : C(\texttt{I})}{\texttt{True\_rect}(d_I) : \prod_{x:\texttt{True}} C(x)}$$

4. $\texttt{True\_rect}(d_I)(\texttt{I}) \equiv d_I$

Lemma

The type True *is contractible, hence in particular a proposition.*

Some more logic: defining $\texttt{False}$

$\texttt{False}$

1. $\emptyset \vdash \texttt{False}$
2.
3. $$\frac{\Gamma \vdash C \qquad \Gamma \vdash p : \texttt{False}}{\texttt{False\_rect}(p) : C}$$
4.

Lemma

The type $\texttt{False}$ *is a proposition.*

Definition (Negation)

$$\neg A := A \to \texttt{False}$$

Outline

Overview

- Inductive types are part of MLTT
- general form: W-types
- semantics: initial algebras of polynomial functors
- in Coq implemented via `Inductive` primitive
  - ⇝ more convenient than W-types

Inductive types: first examples

```
Inductive Nat : Type := O : Nat
                      | S : Nat -> Nat
```

Intuitively,...

the type `Nat` is **freely generated by constructors and operations on terms**.

However,...

there are no operations—other than the constructors—on terms which add inhabitants to `Nat`.

More formally ...

Natural numbers type, formally

1. Nat is a type
2. $\mathtt{0} : \mathtt{Nat}$ and if $n : \mathtt{Nat}$ then $\mathtt{S}(n) : \mathtt{Nat}$
3. $$\frac{n : \mathtt{Nat} \vdash C(n) \qquad d_0 : C(\mathtt{0}) \qquad n : \mathtt{Nat} \vdash d_S(n) : C(n) \to C(\mathtt{S}n)}{\mathtt{Nat\_rect}(d_0, d_S) : \prod_{n:\mathtt{Nat}} C(n)}$$
4. $\mathtt{Nat\_rect}(d_0, d_S)(\mathtt{0}) \equiv d_0$
   $\mathtt{Nat\_rect}(d_0, d_S)(\mathtt{S}n) \equiv d_S(\mathtt{Nat\_rect}(d_0, d_S)(n))$

Non-dependent eliminator for Nat

If C does not depend on $n : \texttt{Nat}$:

❸ $$\frac{\emptyset \vdash C \qquad \emptyset \vdash d_0 : C \qquad \emptyset \vdash d_S : C \to C}{\texttt{Nat\_rect}(d_0, d_S) : \texttt{Nat} \to C}$$

Initial algebra semantics—Iteration

$(\texttt{Nat}, \texttt{O}, \texttt{S})$ is the initial object in the category with

- objects : $(X, x : X, f : X \to X)$
- morphisms: ...

A variant of natural numbers

Instead of

4. $\mathtt{Nat\_rect}(d_{\mathtt{0}}, d_{\mathtt{S}})(\mathtt{0}) \equiv d_{\mathtt{0}}$

   $\mathtt{Nat\_rect}(d_{\mathtt{0}}, d_{\mathtt{S}})(\mathtt{S}(n)) \equiv d_{\mathtt{S}}(\mathtt{Nat\_rect}(d_{\mathtt{0}}, d_{\mathtt{S}})(n))$

we might only ask for

4. $\mathtt{Nat\_rect}(d_{\mathtt{0}}, d_{\mathtt{S}})(\mathtt{0}) \rightsquigarrow d_{\mathtt{0}}$

   $\mathtt{Nat\_rect}(d_{\mathtt{0}}, d_{\mathtt{S}})(\mathtt{S}(n)) \rightsquigarrow d_{\mathtt{S}}(\mathtt{Nat\_rect}(d_{\mathtt{0}}, d_{\mathtt{S}})(n))$

Propositional equality suffices…

to determine the recursor `Nat_rect` uniquely.

Discrimination: needs a universe

Discriminating constructors

Given a universe containing $\mathtt{True}$ and $\mathtt{False}$, we can prove

$$\neg(\mathtt{0} \rightsquigarrow \mathtt{S}(\mathtt{0}))$$

- define $P(\mathtt{0}) := \mathtt{True}$ and $P(\mathtt{S}n) := \mathtt{False}$:

$$P := \mathtt{Nat\_rect}(\mathtt{True}, \lambda n \lambda x.\mathtt{False})$$

- suppose having $p : \mathtt{0} \rightsquigarrow \mathtt{S}(\mathtt{0})$, by substitution principle we get

$$\mathtt{subst}(p) : \mathtt{True} \to \mathtt{False}$$

- define the image of p to be $\mathtt{subst}(p)(\mathtt{I}) : \mathtt{False}$

Interlude: more on homotopy levels

Theorem (Hedberg '98)

*A type A is a **set** if it has "decidable equality", i.e. if we can define a term of type*

$$\texttt{eq\_dec}(A) : \prod_{x,y:A} (x \rightsquigarrow y) + \neg(x \rightsquigarrow y)$$

Consequence

The type `Nat` is a set.

More inductive types

Lists

```
Inductive list (A : Type) : Type :=
    nil : list A
  | cons : A -> list A -> list A
```

Exercise

Write down the 4 rules implementing the type of lists.

Lemma

The homotopy level of `list A` *is the same as that of* `A`.

Theorem (Hofmann & Streicher '95)

Cannot produce a type that is not a set (without univalence).

Outline

1. Preliminaries: reminders and some more logic

2. Inductive Types

3. Higher Inductive Types
    - First example: Interval
    - The circle and dependent elimination
    - Some more examples of HITs
    - Implementation in Coq

Overview over HITs

- suggested **extension** of MLTT
- inspired by **types-as-spaces** interpretation
- but make sense in **types-as-sets** interpretation, **too**
- some results I present depend on Univalence, but the definitions do not

Table of Contents

Idea of HITs

Idea

Introduction rule ② can also **introduce paths**.

- for regular ITs constructors must land in the type itself

- Might look like this in Coq:

```
Inductive Interval : Type :=
     left : Interval (* point *)
   | right : Interval (* point *)
   | seg : left ⇝ right (* path *)
```

- Which are the 4 rules describing `Interval`?

Rules for Interval type

1. $\emptyset \vdash \texttt{Interval}$

2. $\emptyset \vdash \texttt{left} : \texttt{Interval}$
   $\emptyset \vdash \texttt{right} : \texttt{Interval}$
   $\emptyset \vdash \mathsf{seg} : \texttt{left} \rightsquigarrow \texttt{right}$

3. $$\frac{d_{\texttt{left}} : C \qquad d_{\texttt{right}} : C \qquad d_{\mathsf{seg}} : d_{\texttt{left}} \rightsquigarrow d_{\texttt{right}}}{\texttt{Interval\_rect}(d_{\texttt{left}}, d_{\texttt{right}}, d_{\mathsf{seg}}) : \texttt{Interval} \to C}$$

4. $\texttt{Interval\_rect}(d_{\texttt{left}}, d_{\texttt{right}}, d_{\mathsf{seg}})(\texttt{left}) \equiv d_{\texttt{left}}$
   $\texttt{Interval\_rect}(d_{\texttt{left}}, d_{\texttt{right}}, d_{\mathsf{seg}})(\texttt{right}) \equiv d_{\texttt{right}}$
   $\texttt{Interval\_rect}(d_{\texttt{left}}, d_{\texttt{right}}, d_{\mathsf{seg}})(\mathsf{seg}) \rightsquigarrow d_{\mathsf{seg}}$

Comments about the Interval type

- We have

$$\texttt{Interval} \to X \quad \cong \quad \sum_{x,y:X} x \rightsquigarrow y$$

- The type `Interval` is contractible.

Lemma (Semantics)

The type `Interval` *is the initial object in the category with:*

- *objects :* $(X, x, y : X, p : x \rightsquigarrow y)$
- *morphisms:* $\ldots$

Table of Contents

The circle

```
Inductive Circle : Type :=
    base : Circle (* point *)
  | loop : base ⇝ base (* path *)
```

1. $\emptyset \vdash \texttt{Circle}$
2. $\emptyset \vdash \texttt{base} : \texttt{Circle}$
   $\emptyset \vdash \texttt{loop} : \texttt{base} \rightsquigarrow \texttt{base}$
3. $$\frac{b : C \qquad \ell : b \rightsquigarrow b}{\texttt{Circle\_rect}(b, \ell) : \texttt{Circle} \to C}$$
4. $\texttt{Circle\_rect}(b, \ell)(\texttt{base}) \equiv b$
   $\texttt{Circle\_rect}(b, \ell)(\texttt{loop}) \rightsquigarrow \ell$

Dependent elimination for the circle

Goal

Given a dependent type $B : \texttt{Circle} \to \mathcal{U}$, define a dependent function of type

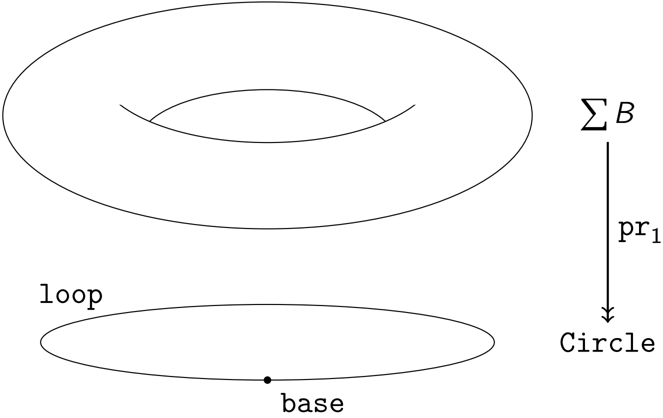
$$\prod_{x:\texttt{Circle}} B(x)$$

Equivalently, define a function $f : \texttt{Circle} \to \sum_{x:\texttt{Circle}} B(x)$ such that

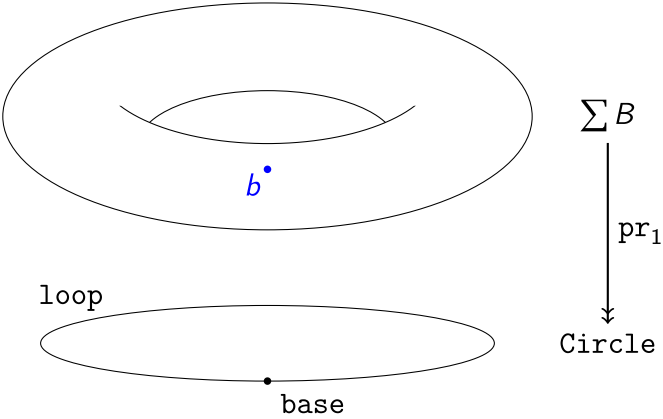
$$\texttt{pr}_1 \circ f = \texttt{id}$$

For this, we need to specify

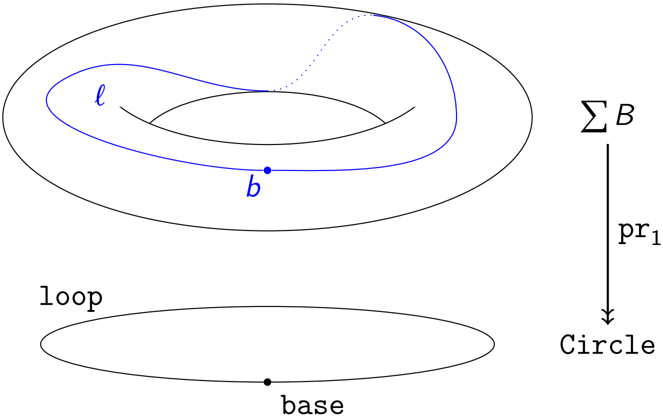
- "point over $\texttt{base}$", i.e. a point $b : B(\texttt{base})$ $(\texttt{pr}_1(b) \equiv \texttt{base})$
- "path over $\texttt{loop}$", i.e. a path $\ell : b \rightsquigarrow b$ s.t. $pr_1(\ell) \rightsquigarrow \texttt{loop}$



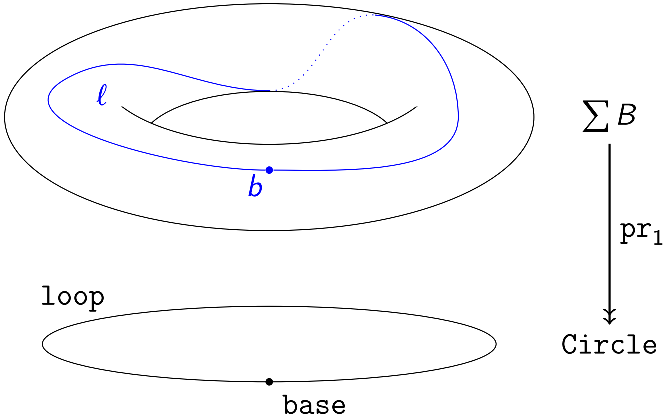
$\sum B$
$\mathtt{pr}_1$
Circle
loop
base



$\sum B$
b
$\mathtt{pr}_1$
loop
Circle
base



ℓ
b
$\sum B$
$\mathtt{pr}_1$
loop
Circle
base



How to ensure that ℓ lies over `loop`?

About paths in total spaces

Recall the substitution principle

For $x : A \vdash B(x)$ and $p : a \rightsquigarrow b$ we have

$$\mathtt{subst}(p) = p_* : B(a) \to B(b)$$

Lemma

For any dependent type $x : A \vdash B(x)$ *and* $x, y : \sum_A B$ *one has*

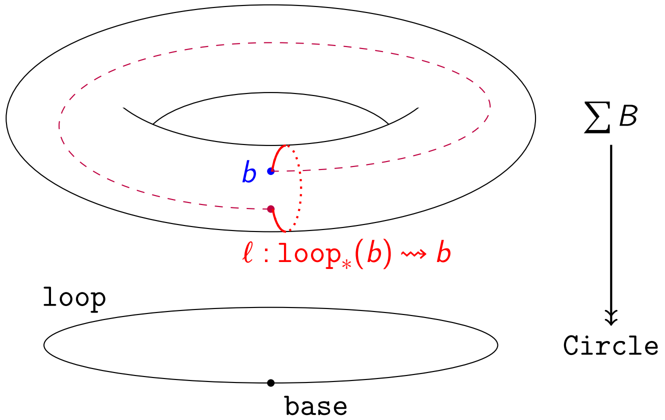
$$x \rightsquigarrow y \quad \cong \sum_{p:\mathtt{pr}_1(x) \rightsquigarrow \mathtt{pr}_1(y)} p_*(\mathtt{pr}_2 x) \rightsquigarrow \mathtt{pr}_2 y$$

- For the circle, we want p to be `loop`.

$\Rightarrow$ We really only need to specify a path

$$\mathtt{loop}_*(b) \rightsquigarrow b$$

Simplified recursion principle



Dependent elimination for the circle

3.
$$\frac{b : C(\text{base}) \qquad \ell : \texttt{loop}_*(b) \rightsquigarrow b}{\texttt{Circle\_rect}(b, \ell) : \prod_{x:\texttt{Circle}} C(x)}$$

4.
$$\texttt{Circle\_rect}(b, \ell)(\text{base}) \equiv b$$
$$\texttt{Circle\_rect}(b, \ell)(\texttt{loop}) \rightsquigarrow \ell$$

The circle is not a point

Lemma

Using univalence, we have $\neg(\texttt{loop} \rightsquigarrow \texttt{refl}_{\texttt{base}})$.

- Suppose $p : \texttt{loop} \rightsquigarrow \texttt{refl}_{\texttt{base}}$.
- For $f : \texttt{Circle} \to B$ specified by $b : B$ and $\ell : b \rightsquigarrow b$ it follows

$$\ell \rightsquigarrow f(\texttt{loop}) \rightsquigarrow f(\texttt{refl}_{\texttt{base}}) \rightsquigarrow \texttt{refl}_b$$

- Since ℓ arbitrary, B is a set.
- But choosing $B := \mathscr{U}$ leads to contradiction, since the universe is not a set:
- type Bool has two different loops corresponding to id and negation, resp.

Fundamental group of the $\texttt{Circle}$

Definition (Loop space)

Define

$$\Omega(X,x) := \texttt{Id}_X(x,x) = x \rightsquigarrow x \ .$$

Theorem

Using Univalence, we have

$$\Omega(\texttt{Circle}, \texttt{base}) \simeq \mathbb{Z} \ .$$

$$\begin{array}{rcl} \Omega(\texttt{Circle}, \texttt{base}) & \simeq & \mathbb{Z} \\ \texttt{loop}^n & \longleftrightarrow & n : \mathbb{Z} \end{array}$$

i.e. freely generated by constructors **and operations**

Table of Contents

Another HIT: Prop truncation

```
Inductive PropTrunc (A : Type) : Type :=
    in : A -> Proptrunc A
 | pi : forall x y : Proptrunc A, x ⇝ y.
```

1. if $\Gamma \vdash A$ then $\Gamma \vdash \|A\|_{\texttt{Prop}}$
2. if $a : A$ then $\texttt{in}(a) : \|A\|_{\texttt{Prop}}$
   $\texttt{pi} : \prod_{x,y} x \rightsquigarrow y$
3. $$\frac{\Gamma \vdash p : \prod_{x,y:B} x \rightsquigarrow y \qquad \Gamma \vdash g : A \to B}{\texttt{PT\_rect}(p, g) : \|A\|_{\texttt{Prop}} \to B}$$
4. $\texttt{PT\_rect}(p, g)(\texttt{in}(a)) \equiv g(a)$

Use of propositional truncation

Some type constructors not closed under proposition:

- $$P \vee Q := \|P + Q\|_{\texttt{Prop}}$$
- $$\exists x. P(x) := \left\| \sum_x P(x) \right\|_{\texttt{Prop}}$$

More examples of HITs

- Quotients
- Free algebraic structures (groups,...)
- Various CW-complexes (topological spaces):
  - suspension
  - torus
  - ...
- Cauchy reals (higher inductive-inductive type)

All of them explained in the HoTT book.

Table of Contents

Implementation in Coq

Idea: Implement the 4 rules via Axioms

```
  (* 1 *)
Axiom Circle : Type.
  (* 2 *)
Axiom base : Circle.
Axiom loop : base ⇝ base.
  (* 3 *)
Axiom Circle_rect : forall (P:Circle->Type)
    (b : P base) (ell : subst P loop b ⇝ b),
    forall c : Circle, P c.
  (* 4 *)
Axiom Circle_rect_base : forall P b ell,
    Circle_rect P b ell base ⇝ b
Axiom Circle_rect_loop : forall P b ell,
    ap (Circle_rect P b ell) loop ⇝ ell.
```

Problems with implementation through axioms

- lack of **conversion** of the eliminator on the points

⇒ Computation rule for paths becomes more complicated:

```
Axiom Circle_rect_eq2 : forall P b ell,
 ap (Circle_rect P b ell) loop ⇝
      ap (transp loop) (Circle_rect_eq P b ell) @
         ell
         @ !(Circle_rect_eq P e ell)
```

instead of

```
ap (Circle_rect P b ell) loop ⇝ ell
```

Conclusion

Not usable in practice.

Hiding in Coq modules

```
Module Circle.
    (* 1-2 *)
 Local Inductive Circle : Type := base : Circle.
    (* 2 bis *)
 Axiom loop : base ⇝ base.
    (* 3-4 *)
 Definition Circle_rect (P:Circle->Type)
    (b : P base) (ell : subst loop b ⇝ b)
     : forall (x:Circle), P x
     := fun x => match x with base => b end.
    (* 4 bis *)
 Axiom Circle_rect_loop: forall (P:Circle -> Type)
    (b : P base) (ell : subst loop b ⇝ b),
    ap (Circle_rect P b ell) loop ⇝ ell.
End Circle.
```

A modification to hiding in modules

Problem: outside the module, one sees that

the argument `ell` is not used in the body of `Circle_rect`

$\Longrightarrow$ **Inconsistency** (e.g. "Universe is a set")

Solution: insert pseudo application of `ell`

```
Definition Circle_rect (P : Circle -> Type)
  (b : P base) (ell : subst loop b ⇝ b)
   : forall (x:Circle), P x
  := fun x => match x with base => fun _ => b end
     ell.
```

Bruno Barras' work on native HITs

- B. Barras works on native implementation of HITs in Coq

\+ automatic generation of the eliminator

\- only a subset of allowed HITs

```
Inductive Circle : Type :=
  | base : Circle
with paths :=
  | loop : base ⇝ base.
```

Induction schemes for native circle

```
Circle_rect : forall P (b : P base)
   (ell : subst loop b ⇝ loop),
   forall (c:Circle), P c
Circle_rect2 :
   forall P b ell (c1 c2:Circle) (e : c1 ⇝ c2),
   subst e (Circle_rect P b ell c1) ⇝
     Circle_rect P b ell c2
```

These two are implemented using a new primitive **fixmatch** construct, allowing to pattern-match on both **point and path constructors**.

- matching reduces on path constructors **and** reflexivity
- allows to **deduce** computation rule for `loop`

General form of HITs

```
Inductive I : A -> Type :=
 c : forall y:C1, (forall i:C2 y -> I(fc y i))
     -> I (gc y)
 with paths :=
 d : forall (z:D1)
    (z':forall i:D2 z -> I(fd z i)),
       b1(z,z',c) ⇝ b2(z,z',c) :> I (gd z).
```

- **recursive**, if C2 non-empty for some input
- **half-recursive**, if D2 non-empty for some input

Barras' HITs cover half-recursive types:

- no recursive construction of points
- only 1-paths
- no paths depending on paths

Examples of HITs covered by Barras' implementation

Covered:

- Interval
- Circle
- Suspension
- Cylinder
- Propositional truncation
- Set truncation (using reduction of 2-path to 1-paths)

Not covered:

- set quotient
- naïve set truncation
- 2-sphere (directly)

Barras' HIT prototypes

Online

- `https://github.com/barras/coq`
- branch "hit"

The end