Native implementation of Higher Inductive Types (HITs) in Coq

Bruno Barras

INRIA Saclay - Île de France

September 24, 2013
From the developer perspective

From Intentional Type Theory, 2 incompatibles extensions:

- Dependent functional programming: UIP or K (set theoretic model)
- HoTT: Univalence

We’d better avoid splitting the community by having HITs independent from the K/Univalence choice. theory.

- We expect HITs + K to be consistent.
HITs + K = quotients

In a proof-irrelevant setting, HITs can be seen as a way to implement quotients in Type Theory.

```
Inductive Z_2Z :=
| O
| S (_,:nat)
| mod2 : O = S (S O).
```
Overview

Introduction

How to model HITs in a proof assistant
  Axiomatization
  Private inductive types
  Native implementation

Introducing a subset of HITs
  Examples
  Typing rules: points
  Typing rules: paths
  What about recursive HITs?

Metatheory
Axiomatization

- Each notion (type/intro/elim) is introduced by a new constant.
- Computation rules are represented by paths!

\[
\text{Axiom } S_1 : \text{Type}. \\
\text{Axiom } \text{base} : S_1. \\
\text{Axiom } \text{loop} : \text{base} = \text{base}. \\
\text{Axiom } S_1\_\text{rect} : \forall (P:S_1 \rightarrow \text{Type}) \\
\quad (f:P \text{ base}) \\
\quad (g:\text{transp} P \text{ loop} f = f) \\
\quad (c:S_1), P c. \\
\text{Axiom } S_1\_\text{rect}\_\text{eq} : \forall P f g, \\
\quad S_1\_\text{rect} P f g \text{ base} = f.
\]
Axiomatization: pros and cons

Pros:
- Simple
- Safe (besides typos)

Cons:
- Definitional equality is not modified
  - Computational interpretation is lost
  - Makes path expressions more complex:

\[
\text{Axiom } S1\_\text{rect}\_eq2 : \text{forall } P \ f \ g, \\
\text{apD} \ (S1\_\text{rect} \ P \ f \ g) \ \text{loop} = \\
\text{ap} \ (\text{transp} \ \text{loop}) \ (S1\_\text{rect}\_eq \ P \ f \ g) @ \\
g @ \\
!(S1\_\text{rect}\_eq \ P \ f \ g)
\]

instead of
\[
\text{apD} \ (S1\_\text{rect} \ P \ f \ g) \ \text{loop} = g
\]
Private inductive types

Proposed by Licata for Agda, adapted to Coq by Bertot.

Idea: restrict the use of the eliminator:

Module Circle.
Local Inductive S1 : Type :=
  | base : S1.
Axiom loop : base = base.

Definition S1_rect (P:S1->Type)
  (b : P base) (l : loop # b = b)
  : forall (x:S1), P x
  := fun x => match x with base => b end.

Axiom S1_rect_beta_loop : forall (P : S1 -> Type)
  (b : P base) (l : loop # b = b),
  apD (S1_rect P b l) loop = l.
End Circle.

From now on, match e with base => f end is not allowed,
we must use S1_rect.
Private inductive types: pros and cons

Pros:

- Definitional equality for points

Cons:

- Consistency relies on the library writer.
- No definitional equality for paths.
- In Coq: eliminator does not depend on path argument (Bordg)

\[ S1\text{\_rect} \, P \, f \, g \, c \, \text{and} \, S1\text{\_rect} \, P \, f \, g' \, c \, \text{both convertible to} \, \text{match} \, c \, \text{with base} \Rightarrow f \, \text{end} \]
Attempt to fix the issue

Bertot suggested:

Definition S1_rect (P : S1 -> Type)
   (b : P base) (l : loop # b = b)
   : forall (x:S1), P x
   := fun x => match x with base => fun _ => b end l.

Seems to work!
Native implementation

- Modify the theory implemented
- With fixed new primitive constants and definitional equalities.
Native implementation: pros and cons

Pros:
- Faithfully encode the desired types (no cheating).
- Consistency is warranted by the meta-theoretical properties of the new formalism.

Cons:
- A lot of implementation work.
Overview

Introduction

How to model HITs in a proof assistant
  Axiomatization
  Private inductive types
  Native implementation

Introducing a subset of HITs
  Examples
  Typing rules: points
  Typing rules: paths
  What about recursive HITs?

Metatheory
A limited subset of Higher-Inductive Types

Design proposed by Lumsdaine and Schulman:
- Only point and path constructors.
- Point constructors cannot refer to path constructors.
- Path constructors are homogeneous equalities.
- The usual strict positivity condition applies.
Examples: the circle

```
Inductive S1 : Type :=
  | base : S1
with paths :=
  | loop : base = base.
```

2 induction schemes are generated:

- **S1_rect** : \(\forall P (f:P \text{ base}) (g:\text{transp} P f \text{ loop} = \text{loop}) (c:S1), P c\)

- **S1_rect2** :
  \(\forall P f g (c1 c2:S1) (e:c1=c2), \text{transp} P (S1\_\text{rect} P f g c1) e = S1\_\text{rect} P f g c2\)
  
  Not convertible to \(\text{apD} (S1\_\text{rect} P f g) e\).
Suspension

The following definition of the sphere is not accepted:

```
Inductive S2 : Type :=
   | base2 : S2
with paths :=
   | surf2 : (@idpath _ base2) = (@idpath _ base2).
```

But we can define the suspension of $X$:

```
Inductive Susp (X : Type) : Type :=
   | north : Susp X
   | south : Susp X
with paths :=
   | merid (x:X) : north = south.
```

and define the sphere as the suspension of the circle.
Truncation

prop-truncation:

\[
\text{Inductive } \text{prop\_tr} (X:\text{Type}) : \text{Type} := \\
\quad | \text{proj} : X \to \text{prop\_tr} X \\
\text{with paths := } \\
\quad | \text{contr} (y \ y' : \text{prop\_tr} X) : y = y'.
\]

But set-truncation requires more work (hub/spoke trick):

\[
\text{Inductive } \text{set\_tr} X : \text{Type} := \\
\quad | \text{truncn} : X \to \text{set\_tr} X \\
\quad | \text{hub} : (\text{Circle} \to \text{set\_tr} X) \to \text{set\_tr} X \\
\text{with paths := } \\
\quad | \text{spoke} (l : \text{Circle} \to \text{set\_tr} X) \ (s : \text{Circle}) : \\
\quad \ (\text{hub} \ l) = (l \ s).
\]
General case

The constraints lead to a most general HIT (we forget parameters):

\[
\text{Inductive } I : A \rightarrow \text{Type} :=
\]
\[
c : \forall (y : C_1), (\forall i : C_2 \ y \rightarrow I(fc y i)) \rightarrow I(gc y)
\]
\[
\text{with paths :=}
\]
\[
d : \forall (z : D_1) (z' : \forall i : D_2 \ z \rightarrow I(fd z i)),
\]
\[
b_1(z, z', c) = b_2(z, z', c) :> I(gd z).
\]

where \(b_1\) and \(b_2\) are applicative terms using \(c\) and \(z'\).

This is the analogous of what \(W\)-types are for inductive types.

Terminology:

- \(I\) is recursive if \(C_2\) is not empty for some \(y : C_1\).
- \(I\) is half-recursive if \(D_2\) is not empty for some \(z : D_1\).
Formation rule

Positivity condition applies.

Restriction for path constructors:

- Can have point arguments, but not paths
- Conclusion is an equation which handsides have a limited syntax
- The equation must relate two points with same indices
Introduction rules

No surprises: introduces point and path constructors with the type declared.
Elimination rules

In Coq, the primitive notion is not an elimination constant, but a pattern-matching operator, and a (guarded) fixpoint operator for recursive types. For non-recursive types, the pattern-matching operator and the usual eliminator coincide.
Pattern-matching and half-recursive types

For half-recursive HITs, we need to refer to the image of the elimination rule for the recursive arguments (e.g. prop-truncation):

\[
\text{Inductive } \text{prop\_tr} \ (X : \text{Type}) \ : \ \text{Type} \ := \\
\quad \mid \text{proj} : X \to \text{prop\_tr} \ X \\
\text{with } \text{paths} \ := \\
\quad \mid \text{contr} \ (y \ y' : \text{prop\_tr} \ X) : y = y'.
\]

has the following eliminator:

\[
\text{prop\_tr\_rect} : \\
\quad \text{forall} \ (X : \text{Type}) \ (P : \text{prop\_tr} \ X \to \text{Type}) , \\
\quad (\text{forall} \ x : X, P \ (\text{proj} \ x)) \to \\
\quad (\text{forall} \ (y : \text{prop\_tr} \ X) \ (h : P \ y) \\
\quad \quad (y' : \text{prop\_tr} \ X) \ (h0 : P \ y'), \\
\quad \quad \text{transp} \ P \ (\text{contr} \ y \ y') \ h = h0) \to \\
\quad \text{forall} \ i : \text{prop\_tr} \ X, P \ i
\]
The fixmatch operator

`prop_tr_rect` is defined as

```
prop_tr_rect =
  fun (X : Type) (P : prop_tr X -> Type)
    (f : forall x : X, P (proj x))
    (g : forall (y : prop_tr X) (h : P y)
      (y' : prop_tr X) (h0 : P y'),
      transp P (contr y y') h = h0) (p : prop_tr X) =>
    fixmatch {h} p return (P p) with
    | proj x => f x
    | contr y y' => g y (h y) y' (h y')
  end
```

Note: `fixmatch` is just the concrete syntax for introducing the name `h` in path branches.
Typing rules: fixmatch

\[ \vdash P : \Pi a : A. I a \rightarrow \text{Type} \]
\[ \vdash t : I a \]
\[ y y' \vdash f : P () (c y y') \]
\[ (h : \Pi a : A. \Pi t : I a. P a t) z z' \vdash g : \text{transp } P u' (d z z') = v' \]

\[
\text{fixmatch } \{h\} t \text{ with } c y y' \Rightarrow f \mid d z z' \Rightarrow g \text{ end : } P a t
\]

where \( u' \) and \( v' \) are \( u \) and \( v \) with \( c \) replaced by \( f \) and \( z' \) replaced by \( \lambda i.h(z' i) \).

The \( \nu \)-reduction is defined as usual:

\[
\text{fixmatch}\{h\}c b b' \text{ with } c z z' \Rightarrow f(z, z') \mid \ldots \text{ end}
\]
reduces to \( f(b, b') \).
Path eliminator

fixmatch is extended to paths (and used in the S1_rect2 generated principle).

\[
\Gamma \vdash P : \Pi a : A. I a \rightarrow \text{Type} \\
\Gamma \vdash e : t_1 = t_2 :> I a \\
\Gamma \vdash f : P () (c y y') \\
(h : \Pi a : A. \Pi t : I a. P a t) z z' \vdash g : \text{transp } P u' (d z z') = v'
\]

\[
\Gamma \vdash \text{fixmatch } \{h\} e \text{ with } c y y' \Rightarrow f \mid d z z' \Rightarrow g \ \text{end} \\
: \text{transp } P \ \text{fixmatch } \{h\} t_1 \text{ with...end } e \\
= \text{fixmatch } \{h\} t_2 \text{ with...end }
\]
Reduction rules of the path eliminator

Reduction rules:
fixmatch\{h\}d(a) \text{ with } ... \mid d(z) \Rightarrow g(z,h) \text{ end} \text{ reduces to}
g(a,\text{fun } y \ x \Rightarrow \text{fixmatch}\{h\}x \text{ with } ... \mid d(z) \Rightarrow g(z,h) \text{ end}

We also have a rule for reflexivity:
fixmatch\{h\}r(x) \text{ with } ... \mid d(z) \Rightarrow g(z,h) \text{ end} \text{ reduces to}
r(\text{fixmatch}\{h\}x \text{ with } ... \mid d(z) \Rightarrow g(z,h) \text{ end})
(a bit more tricky than this!)

However, we have not managed to express a rule when the path is a composition.
Path constructor properties

Using \textit{both} reduction rules, we have a closed proof of

\[ \text{apD (S1_rect P f g) loop = g} \]

- Generalizes to all HITs
- Equality does not hold definitionally, the lhs is stuck

This fulfills the requirements for proving e.g. \( \pi_1(S^1) = \mathbb{Z} \) (assuming univalence).
Recursive HITs

In Coq, the usual primitive recursor is not a primitive notion. Rather it results from pattern-matching (case-analysis) and a fixpoint operator (recursion).

More convenient for deep recursion:

```coq
Fixpoint mod2 (n:nat) : nat :=
    match n with
    | O | S O => n
    | S (S n') => mod2 n'
    end.
```

Not acceptable to give that up!
Without deep pattern-matching, one uses a “pipe-line” (this idea generalizes to arbitrary inductive types, cf Gimenez).
Recursive HITs

Does it transport to HITs?

```coq
Inductive Z_2Z :=
| O | S (_:nat)
| mod2 : O = S (S O).
```

```coq
Definition mod2_body (f:Z_2Z->Z_2Z) (n:Z_2Z) : Z_2Z :=
match n with
| O => O
| S k => match k with
  | O => (S O)
  | S n' => f n'
  | mod2 => _ : f (S O) = (S O)
end
| mod2 => _ : f O = O
end.
```

Unfortunately not, currently.
Overview

Introduction

How to model HITs in a proof assistant
  Axiomatization
  Private inductive types
  Native implementation

Introducing a subset of HITs
  Examples
  Typing rules: points
  Typing rules: paths
  What about recursive HITs ?

Metatheory
Syntactic metatheory

- Confluence
  Definitional equality decided by common reduct.

- Subject-Reduction
  “Well-typed programs can’t go wrong”

- Strong normalization
  decidability + “Proof terms don’t hide anything”

- Canonicty
  Proof in normal form begin with an introduction.

Canonicity does not hold.
Canonicity

Canonicity is a global result: lack of it in one type (except types with only weak eliminations) pervades all types.

Sources of non-canonicity:

- $\equiv_{\text{Type}}$: univalence
- $\equiv_{\Pi x:A. B}$: functional extensionality
- $\equiv_I$: path constructors
- in all cases: groupoid ops

But J only deals with reflexivity, not even composition.

To make it worse path composition is derived from J.
Conclusions

J under fire
  - J should be decomposed (as suggested by Coquand’s models)

Implementation:
  - Recursive types not well-supported (set-truncation, quotients)