# Higher Inductive Types: The circle and friends, axiomatically


Peter LeFanu Lumsdaine

Dalhousie University
Halifax, Nova Scotia



Foundational Methods in Computer Science
Kananaskis, June 2011

# DTT

Dependent Type Theory (Martin-Löf, Calculus of Constructions, etc.): highly expressive constructive theory, potential foundation for maths.

Central concept: *terms* of *types*.

| | |
|---|---|
| $\vdash \mathbb{N}$ **type** | `Nat : Type` |
| $\vdash 0 : \mathbb{N}$ | `O : Nat` |
| (M-L notation) | (pseudo-Coq syntax) |

Both can be *dependent* on (typed) variables:

$$n : \mathbb{N} \vdash \mathbb{R}^n \ \textsf{type}$$

```
Real_Vec (n:Nat) : Type
```

"$\mathbb{R}^n$ is a dependent type over $\mathbb{N}$."

# DTT

Terms of dependent types, e.g. the "polymorphic zero vector":

$$n : \mathbb{N} \vdash \mathbf{0}_n : \mathbb{R}^n$$
$$\vdash \mathbf{0} : \prod_n \mathbb{R}^n$$

```
poly_zero (n:Nat) : Real_Vec n
poly_zero : forall (n:Nat), Real_Vec n
```

Original intended interpretation: **Sets**. Types are sets; terms are elements of sets.

Dependent type over $X$:

$$X \xrightarrow{Y} \mathbf{Sets} \qquad \text{or} \qquad \begin{array}{c} Y = \sum_{i \in X} Y_i \\ \downarrow \\ X \end{array}$$

# DTT

Logic within dependent type theory: Curry-Howard.

```
Euclid : forall (n:Nat), exists (p:Nat),
                          (p > n) /\ (isPrime p).
```

A *predicate* on `X:Type` is represented as a dependent type `P : X -> Type`.

(In classical set model, $P(x)$ will be 1 or 0, depending on whether $P$ holds at $x$.)

# Homotopy Type Theory

Predicate representing equality/identity:

$$x, y : A \vdash \mathrm{Id}_A(x, y)\ \mathsf{type} \qquad \texttt{Id (x y:A) : Type}$$

```
isPrime (n:Nat) : Type
  := ~(Id n 1) /\
     forall d:Nat, (d divides n) ->
                   (Id d 1) \/ (Id d n).
```

Has clear, elegant axioms, and excellent computational behaviour. Can one prove it represents a proposition, i.e. any two terms `p q : Id x y` are equal?

# Homotopy Type Theory

Predicate representing equality/identity:

$$x, y : A \vdash \mathrm{Id}_A(x, y) \ \textsf{type}$$ `Id (x y:A) : Type`

```
isPrime (n:Nat) : Type
  := ~(Id n 1) /\
     forall d:Nat, (d divides n) ->
                   (Id d 1) \/ (Id d n).
```

Has clear, elegant axioms, and excellent computational behaviour. Can one prove it represents a proposition, i.e. any two terms `p q : Id x y` are equal?

**“Problem”.** No! (Hofmann-Streicher groupoid model, 1995.)

Why should this be a problem?

# Homotopy Type Theory

Problem: a mismatch! Original conception: a theory of something like sets. Formulation largely motivated by computational behaviour, constructive philosophy. Types of the theory end up not behaving like familiar classical sets.

# Homotopy Type Theory

Problem: a mismatch! Original conception: a theory of something like sets. Formulation largely motivated by computational behaviour, constructive philosophy. Types of the theory end up not behaving like familiar classical sets.

One solution: add more axioms — “equality reflection”, etc. Problem: destroys computational content, makes typechecking undecidable, etc.

# Homotopy Type Theory

Problem: a mismatch! Original conception: a theory of something like sets. Formulation largely motivated by computational behaviour, constructive philosophy. Types of the theory end up not behaving like familiar classical sets.

One solution: add more axioms — "equality reflection", etc. Problem: destroys computational content, makes typechecking undecidable, etc.

Alternative: see types as being something more like *spaces* — topological spaces, (higher) groupoids, etc. **Change our idea of what this is a theory of.**

Precise statements: models of the theory in **Top**, **SSet**, *n*-**Gpd**, nice Quillen model categories… (Awodey, Warren, Garner, van en Berg, etc.); conversely, higher categories, wfs's, etc. from theory (Garner, Gambino, van den Berg, PLL).

# Homotopy Type Theory

Idea: work with dependent type theory as a theory of *homotopy types*.

`Id x y` not just proposition of “equality”, but *space of paths* from `x` to `y`.
Notation: write `x ~~> x'` for `Id A x x'`.

Dep. type `Y : X -> Type` — a fibration $\begin{array}{c} Y \\ \big\downarrow{\scriptstyle p} \\ X \end{array}$

Term `f : forall x:X, (Y x)` — a section $\begin{array}{c} Y \\ {\scriptstyle f}\big\uparrow\big\downarrow{\scriptstyle p} \\ X \end{array}$.

# Homotopy Type Theory

Programme (Voevodsky et al): develop homotopy theory axiomatically within this logic.

So far, enough to start making definitions: contractibility, loop spaces, equivalence…

But: how to start building interesting spaces? Circles, spheres, … ?

# Inductive types

Main standard type-construction principle: *inductive types*.

```
Inductive Nat : Type where
  | zero : Nat
  | suc : Nat -> Nat.
```

"Let `Nat` be the type freely generated by an element `zero : Nat` and a map `suc : Nat -> Nat`."

From this specification, Coq automatically generates *induction principle* (aka *recursor*, *eliminator*) for `Nat`:

```
forall (P : Nat -> Type)
       (d_zero : P zero)
       (d_suc : forall (n:Nat), P n -> P (suc n)),
forall (n : Nat), P n.
```

# Higher Inductive Types

Extend this principle: allow constructors to produce paths.

```
Inductive Circle : Type where
  | base : Circle
  | loop : base ~~> base.
```

"Let `Circle` be the type freely generated by an element `base : Circle` and a path `loop : base ~~> base`."

Can't actually type this definition into Coq (yet). What should its induction principle be?

# Circle

Type of non-dependent eliminator is clear:

```
forall (X : Type)
       (d_base : X)
       (d_loop : d_base ~~> d_base),
Circle -> X
```

Not powerful enough to do much with. Need to be able to eliminate into *dependent* type. How about:

```
forall (P : Circle -> Type)
       (d_base : P base)
       (d_loop : d_base ~~> d_base),
forall (x:Circle), P x.
```

# Circle

Type of non-dependent eliminator is clear:

```
forall (X : Type)
       (d_base : X)
       (d_loop : d_base ~~> d_base),
Circle -> X
```

Not powerful enough to do much with. Need to be able to eliminate into *dependent* type. How about:

```
forall (P : Circle -> Type)
       (d_base : P base)
       (d_loop : d_base ~~> d_base),
forall (x:Circle), P x.
```

# Interval

Digression: axiomatise the interval, as warmup.

```
Inductive Interval : Type where
  | src : Interval
  | tgt : Interval
  | seg : src ~~> tgt.
```

Induction principle?

Given fibration `P : Interval -> Type`, how to produce section?

Need points `d_src:(P src)`, `d_tgt:(P tgt)`, and a path `d_seg` between them.

# Interval

Digression: axiomatise the interval, as warmup.

```
Inductive Interval : Type where
  | src : Interval
  | tgt : Interval
  | seg : src ~~> tgt.
```

Induction principle?

Given fibration `P : Interval -> Type`, how to produce section?

Need points `d_src:(P src)`, `d_tgt:(P tgt)`, and a path `d_seg` between them.

Problem: `d_src ~~> d_tgt` doesn't typecheck — `d_src`, `d_tgt` have different types. How to get type for `d_seg`?

# Interval

Answer: *transport* between fibers of a fibration, derivable in the type theory:

```
transport {X : Type} {P : X -> Type}
          {x y : X} (u : x ~~> y) (a : P x)
          : P y
```

So, induction principle for interval:

```
forall (P : Interval -> Type)
   (d_src : P src) (d_tgt : P tgt)
   (d_seg : (transport seg d_src) ~~> d_tgt),
forall (x:Interval), P x.
```

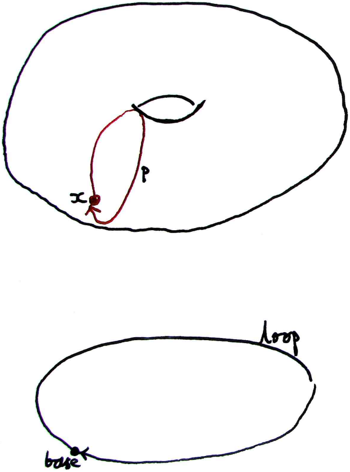
# Circle

In induction principle, the case for a constructor of path type should *lie over* that path.

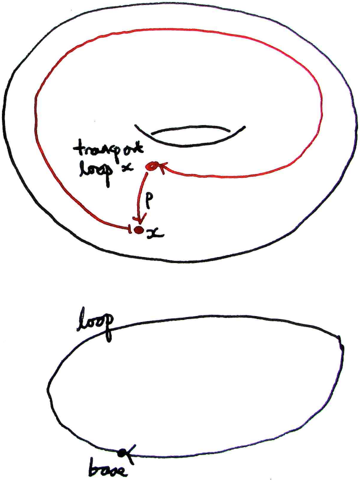
Correct induction principle for the circle:

```
forall (P : Circle -> Type)
   (d_base : P base)
   (d_loop : (transport loop d_base) ~~> d_base),
forall (x:Circle), P x.
```

Not a section.




**Section!**



# Consequences

What can we prove with these?

- `Interval` is contractible.

# Consequences

What can we prove with these?

- `Interval` is contractible.
- `Interval` implies functional extensionality.

# Consequences

What can we prove with these?

- `Interval` is contractible.
- `Interval` implies functional extensionality.
- `Circle` is contractible iff all path types are trivial (i.e. in a **Sets**-like model).

# Consequences

What can we prove with these?

- `Interval` is contractible.
- `Interval` implies functional extensionality.
- `Circle` is contractible iff all path types are trivial (i.e. in a **Sets**-like model).
- "$\pi_1(S^1) \cong \mathbb{Z}$." Assuming Univalence ("equality between types is homotopy equivalence"), loop space of `Circle` is homotopy-equivalent to `Int`.

# Models

Can interpret `Circle` (and the other HIT's below) in:

- **Set**: trivially, 0-truncated.
- **Gpd**: 1-truncated; but with a good enough univalent universe that the above theorem applies.
- **str**-$n$-**Gpd**, for $n \leq \omega$.

Hopefully also $\mathbf{Sets}^{\Delta^{\mathrm{op}}}$, **Top**?

# More Higher Inductive Types

- Familiar spaces with good cell complex structures: higher spheres, tori, Klein bottle, . . .
- Maps between these: universal covers, Hopf fibration, . . .
- Mapping cylinders. From these, wfs's as for a Quillen model structure.
- Truncations, homotopy groups: $\mathrm{tr}_{-1} = \pi_{-1}$, $\mathrm{tr}_0 = \pi_0$, $\mathrm{tr}_1$, $\pi_1$, . . .

# Truncations

By using *proper recursion* (like `suc` for `Nat`), can construct *truncations* as higher inductive types:

```
Inductive isInhab (X:Type) : Type where
  | incl : X -> isInhab X
  | contr : forall (y y' : isInhab X),
                    y ~~> y'.
```

Gives the *support* of a type, aka $-1$-*truncation* $\mathrm{tr}_{-1} = \pi_{-1}$, *homotopy-proposition reflection*, *bracket types* (Awodey, Bauer).

Gives an alternate "homotopy-proposition" interpretation of logic in the DTT, besides Curry-Howard. So may even have *classical* logic existing inside a completely constructive type theory!

Thank you!

References, related reading, Coq files, and much more at:
http://homotopytypetheory.org