Programming = proving?
The Curry-Howard correspondence today

Tenth lecture

What is equality?
From Leibniz to homotopy type theory

Xavier Leroy

Collège de France

2019-01-23

What do we mean?

In computing, when we write in a program

```
e1 = e2      e1 == e2      e1 === e2      e1.equals(e2)
```

In mathematics, when we write

$$\Delta = b^2 - 4ac \qquad \frac{2}{4} = \frac{1}{2} \qquad e^{i\pi} = -1$$

In philosophy, when we talk about object identity.

(Is Theseus' ship still the same after all of its parts were replaced with new ones?)

I

Notions of equality

The equality known as Leibniz equality

Two objects are equal
if and only if
every property that holds for one object also holds for the other

In higher-order logic, this principle provides a definition of equality known as “Leibniz equality”:

$$x = y \quad \overset{def}{=} \quad \forall P,\ P(x) \Leftrightarrow P(y)$$

Render unto Leibniz …

Principle of indiscernibility of identicals: two identical entities have the same properties.

> *A et B sont identiques signifie qu'ils peuvent être substitués l'un à l'autre dans toutes les propriétés* salva veritate.
>
> *G. W. Leibniz,* Échantillon de calcul universel

Principe of identity of indiscernibles: if two entities have the same properties, then they are identical.

> *[I]l n'est pas vrai que deux substances se ressemblent entièrement et soient différentes* solo numero.
>
> *G. W. Leibniz,* Discours de métaphysique

Variations on Leibniz equality

An axiomatization in first-order logic:

$$\begin{aligned} \text{Reflexivity axiom: } & \forall x,\ x = x \\ \text{Axiom schema: } & \forall x, y,\ x = y \Rightarrow (P(x) \Leftrightarrow P(y)) \\ \text{(one axiom per predicate } P\text{)} & \end{aligned}$$

From these axioms, the converse property follows:
if $P(x) \Leftrightarrow P(y)$ for every predicate P,
we take $\lambda z.\ (x = z)$ for P and we have $x = x \Leftrightarrow x = y$,
thus $x = y$.

Variations on Leibniz equality

We can replace the equivalence $P(x) \Leftrightarrow P(y)$ by an implication:

$$x = y \quad \stackrel{def}{=} \quad \forall P,\ P(x) \Rightarrow P(y)$$

Despite the apparent asymmetry, the definition is equivalent:
if $\forall P,\ P(x) \Rightarrow P(y)$, taking $P = \lambda z.\ P(z) \Rightarrow P(x)$ we get

$$(P(x) \Rightarrow P(x)) \Rightarrow (P(y) \Rightarrow P(x)) \qquad \text{and, therefore,} \qquad P(y) \Rightarrow P(x)$$

Hence $\forall P,\ P(y) \Rightarrow P(x)$.

Equivalence relations

A relation R is an equivalence relation if it is

$$\begin{array}{ll} \text{reflexive:} & \forall x,\ R(x,x) \\ \text{symmetric:} & \forall x,y,\ R(x,y) \Rightarrow R(y,x) \\ \text{transitive:} & \forall x,y,z,\ R(x,y) \wedge R(y,z) \Rightarrow R(x,z) \end{array}$$

Another definition of equality over a set A: it is the smallest of the equivalence relations over A, that is, the intersection of all these relations.

$$\begin{array}{rcl} x =_A y & \overset{def}{=} & (x,y) \in \bigcap\{R \mid R \text{ equivalence relation over } A\} \\ & \overset{def}{=} & R(x,y) \text{ for all equivalence relations } R \text{ over } A \end{array}$$

This definition is equivalent to Leibniz equality. (Exercise.)

Equality in type theory

In the 1973 version of his type theory, Per Martin-Löf introduced a type $x =_A y$ of identities between $x, y : A$.

An element of type $x =_A y$ is a proof of equality between x and y.

This type has one constructor $\texttt{refl}_A$ and one eliminator J_A.

$$\texttt{refl}_A : \forall x : A.\ x =_A x$$

$$\begin{aligned} J_A : \forall C : (\forall x, y : A.\ x =_A y \to Set). \\ (\forall z : A.\ C\ z\ z\ (\texttt{refl}_A\ z)) \to (\forall x, y : A.\ \forall s : x =_A y.\ C\ x\ y\ s) \end{aligned}$$

such that $J_A\ C\ d\ a\ a\ (\texttt{refl}_A(a)) = d\ a : C\ a\ a\ (\texttt{refl}_A(a))$.

Equality in type theory

$$
\begin{aligned}
J_A : \forall C : (\forall x, y : A.\ x =_A y \to Set).& \\
&(\forall z : A.\ C\ z\ z\ (\mathtt{refl}_A\ z)) \to (\forall x, y : A.\ \forall s : x =_A y.\ C\ x\ y\ s)
\end{aligned}
$$

In other words: let C be a three-place predicate, x and y of type A, and $s : x =_A y$ a proof of equality between x and y.

In order for C to be always true, it suffices that it is true in the case where x and y are the same variable z and s is the trivial equality $\mathtt{refl}_A\ z$.

Example: indiscernability of identicals!

Let $P : A \to Set$ be a predicate. We take $C\ x\ y\ s = (P\ x \to P\ y)$.
Clearly, $C\ z\ z\ (\mathtt{refl}_A\ z) = P\ z \to P\ z$ holds.
Hence, for any proof of $x =_A y$, we have $P\ x \to P\ y$.

Equality in type theory

The type $x =_A y$, interpreted via Curry-Howard as a proposition, is equivalent to Leibniz equality:

- Reflexivity: the type $x =_A x$ is inhabited by $\texttt{refl}_A\, x$.
- Indiscernability of identicals: if $x =_A y$ is inhabited, then $P\, x \to P\, y$ for all predicates $P : A \to Set$.

Moreover, we can "compute with equality proofs" in an effective way. For instance, using J we can define terms

$$\texttt{sym}_A : \forall x, y : A.\; x =_A y \to y =_A x$$

$$\texttt{trans}_A : \forall x, y, z : A.\; x =_A y \to y =_A z \to x =_A z$$

that satisfy $\texttt{trans}_A\, x\, y\, x\, s\, (\texttt{sym}_A\, x\, y\, s) \xrightarrow{*} \texttt{refl}_A\, x$.

Equality as an inductive predicate

If we have inductive families, that is, inductive predicates, as in Agda and Coq, we can define equality as an inductive predicate.

```
Inductive eq (A: Type): A -> A -> Prop :=
  | eq_refl: forall (x: A), eq A x x.
```

Coq uses the following variant, logically equivalent but sometimes easier to use:

```
Inductive eq (A: Type) (x: A): A -> Prop :=
  | eq_refl: eq A x x.
```

Equality as an inductive predicate

```
Inductive eq (A: Type) (x: A): A -> Prop :=
  | eq_refl: eq A x x.
```

The induction principle for this inductive predicate, automatically generated by Coq, is the principle of indiscernability of identicals:

```
eq_ind: forall (A : Type) (x : A) (P : A -> Prop),
        P x -> forall y : A, eq A x y -> P y
```

It follows that this `eq` predicate is equivalent to Leibniz equality:

```
forall (A: Type) (x y: A),
eq A x y <-> forall (P: A -> Prop), P x -> P y.
```

Equality as an inductive predicate

```
Inductive eq (A: Type) (x: A): A -> Prop :=
    eq_refl: eq A x x.
```

All the uses (“eliminations”) of an equality reduce to pattern-matching over terms of type `eq A x y`. For instance, the principle of indiscernability of identicals:

```
Definition F (A: Type) (P: A -> Prop) (x y: A) (s: eq A x y)
           : P x -> P y :=
  match s with
  | eq_refl _ _ => fun p => p
  end.
```

Exercise: define Martin-Löf’s eliminator J_A by pattern-matching.

II

The highs and lows of equality in Coq

Equality over simple inductive types

Equality from type theory behaves well over purely inductive types such as `nat` or `nat * bool` or `list nat`.

In particular, extensionality holds: two data structures are equal if and only if all their components are equal. For instance, in the case of two lists:

$$[x_1; \ldots; x_p] = [y_1; \ldots; y_q] \quad \text{iff} \quad p = q \text{ and } x_i = y_i \text{ for i = 1}, \ldots p$$

Moreover, equality is decidable: for a purely inductive type A, we can define a function $\texttt{beq} : A \to A \to \texttt{bool}$ such that $\texttt{beq}\ x\ y$ returns `true` iff $x = y$ and returns `false` iff $x \neq y$.

```
Fixpoint beq_nat (p q: nat) : bool :=
  match p, q with
  | O, O => true | S p, S q => beq_nat p q | _, _ => false
  end
```

Equality between functions

Two functions are equal if they are convertible. For instance, using Coq's definition for + :

$$\begin{array}{llll} \texttt{(fun x => 1 + x)} & =_\beta & \texttt{(fun x => S x)} & =_\eta \quad \texttt{S} \\ \texttt{(fun x => x + 1)} & \neq_{\beta\eta} & \texttt{S} & \end{array}$$

This is the only way to show that two functions are equal. In particular, we cannot prove an extensionality principle (two functions having the same graph are equal).

FE (*Function Extensionality*)

$$\forall A, B : \mathtt{Type}.\ \forall f, g : A \to B.\ (\forall x : A,\ f\ x = g\ x) \to f = g$$

DFE (*Dependent Function Extensionality*)

$$\forall A : \mathtt{Type}.\ \forall B : A \to \mathtt{Type}.\ \forall f, g : \Pi(x : A).\ B\ x.\ (\forall x : A,\ f\ x = g\ x) \to f = g$$

Equality between functions

Consider two functions $f, g : A \to B$ such that $\forall x : A.\ f\ x = g\ x$.

An argument based on logical relations shows contextual equivalence between f and g, from which it follows that $P\ f$ and $P\ g$ are logically equivalent for all $P : (A \to B) \to \texttt{Prop}$.

However, this is a “meta” argument that cannot be proved within the type theory!

Actually, **FE** and its extension **DFE** are independent from CC + universes:

- Set-based models validate **DFE**.
- A syntactic model by Boulier, Pédrot, Tabareau (2017) invalidates it.

Equality between coinductive types

As in the case of functions, equality over coinductive types is not extensional: two streams s_1, s_2 such that

$$\mathtt{hd}(\mathtt{tl}^n(s_1)) = \mathtt{hd}(\mathtt{tl}^n(s_2)) \quad \text{for all } n$$

do not satisfy $s_1 = s_2$ in general.

Usually, we reason not over equality between streams but over bisimilarity between streams, a notion defined as a coinductive predicate:

```
CoInductive bisim (A: Type): stream A -> stream A -> Prop :=
| bisim_intro: forall s1 s2,
    hd s1 = hd s2 -> bisim (tl s1) (tl s2) -> bisim s1 s2.
```

The extensionality axiom for streams $\forall s_1, s_2.\ \mathtt{bisim}\ s_1\ s_2 \Rightarrow s_1 = s_2$ is presumed independent from Coq's logic.

Equality between proof terms

A value of a (co-)inductive type can contain proof terms: values of a type P : `Prop` representing a logical proposition.

Example: the subset type $\{x : A \mid P(x)\}$, defined by

```
Inductive sig (A: Type) (P: A -> Prop) : Type :=
  | exist: forall (x: A), P x -> sig A P.
```

A value of type $\{x : A \mid P(x)\}$ is a pair of an $x : A$ and a proof of $P\,x$.

Equality between proof terms

To show that two values of type $\{x : A \mid P(x)\}$ are equal, we need to show not only that their first components x are equal, but also that the two proofs of $P\,x$ are equal. Equality between proof terms can be quite surprising indeed!

Example: we'd like to define $\mathbb{Z}$ as a quotient of $\mathbb{N} \times \mathbb{N}$.

```
Definition Z := { p: nat * nat | fst p = 0 ∨ snd p = 0 }
```

There are two proofs of $0 = 0 \vee 0 = 0$: the proof stating that the left claim is true, and the proof stating that the right claim is true. Hence two definitions for the zero of Z:

```
Definition zero  : Z := exist _ (0,0) (or_introl eq_refl).
Definition zero' : Z := exist _ (0,0) (or_intror eq_refl).
```

We cannot prove that `zero` $=$ `zero'`.
(Neither can we prove that `zero` $\neq$ `zero'`, by the way.)

Uniqueness of proof terms

We would like two values of type $\{x : A \mid P(x)\}$ to be equal as soon as their first components x are equal.

Three possibilities:

1. Show that proofs of $P(x)$ are unique:
   for all $p, q : P(x)$ we have $p = q$.
   Often impossible to prove; always difficult to prove.

2. Replace P by an equivalent predicate Q that has the unique proof property, typically a Boolean equality $f\ x = \mathtt{true}$.

3. Take proof irrelevance as an axiom:

   PI (*Proof Irrelevance*) $\quad \forall P : \mathtt{Prop}.\ \forall p, q : P.\ p = q$

Uniqueness of identity proofs

An important special case is uniqueness of identity proofs:

UIP(A) (*Uniqueness of Identity Proofs*) $\forall x, y : A.\ \forall p, q : x = y.\ p = q$

We can prove this property for several types A, in particular those where equality is decidable: $\forall x, y : A.\ x = y \vee x \neq y$.
This includes purely inductive types such as `bool` and `nat`.

(Hence the idea to replace $\{x \mid P\,x\}$ by $\{x \mid f\,x = \mathtt{true}\}$.)

We can also take UIP as axiom, for a given type, or for all types.

Summary

Equality as defined in type theory is perfect for purely inductive types, but does not allow us to identify object that we think are equal:

- two functions that have the same graph;
- two streams that are bisimilar;
- two proofs of the same proposition;
- two propositions P, Q that are equivalent $P \Leftrightarrow Q$.

Approach 1: the setoids

We can systematically work over types A equipped with equivalence relations eq_A that are the desired notions of equality, for instance

$$eq_{A \to B}\, f\ g = \forall x : A.\ eq_B\ (f\ x)\ (g\ x)$$

A pain: we must prove compatibility of every function or predicate definition.

$$\text{for all functions } f : A \to B \text{, show } \forall x, y : A.\ eq_A\ x\ y \to eq_B\ (f\ x)\ (f\ y)$$
$$\text{for all predicates } P : A \to \texttt{Prop} \text{, show } \forall x, y : A.\ eq_A\ x\ y \to P\ x \Leftrightarrow P\ y$$

Coq provides some notations and tactics to facilitate this style.

Approach 2: add axioms

The most common axioms:

$$\forall A, B : \texttt{Type}.\ \forall f, g : A \to B.\ (\forall x : A,\ f\ x = g\ x) \to f = g \tag{FE}$$

$$\forall A : \texttt{Type}.\ \forall B : A \to \texttt{Type}.\ \forall f, g : \Pi(x : A).\ B\ x.\ (\forall x : A,\ f\ x = g\ x) \to f = g \tag{DFE}$$

$$\forall P, Q : \texttt{Prop},\ (P \Leftrightarrow Q) \Rightarrow P = Q \tag{PE}$$

$$\forall P : \texttt{Prop}.\ \forall p, q : P.\ p = q \tag{PI}$$

$$\forall x, y : A.\ \forall p, q : x = y.\ p = q \tag{UIP}$$

We have good reasons to believe that these axioms are consistent with CC + universes. With all of Coq, it's less clear.

Also: **PE** and **PI** rely on the very special status of `Prop` in Coq, and make no sense in other type theories (e.g. Agda).

Approach 3: think equality differently

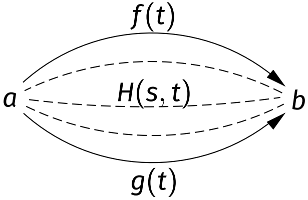
A fresh perspective on equality could enlighten us …

III

Equality and homotopy

Homotopy

A tool from algebraic topology that considers continuous deformations between two topological objects.

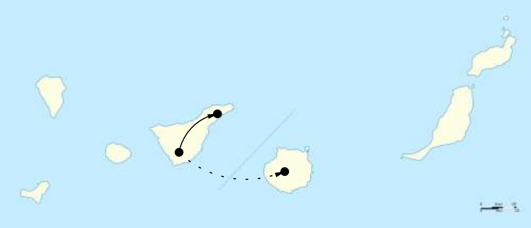


Example of topological object:
a path between two points a, b of space A is a continuous function $f : [0,1] \to A$ such that $f(0) = a$ and $f(1) = b$.

Example of continuous deformation:
two paths f, g from a to b are homotopic if there exists a continuous function $H : [0,1] \times [0,1] \to A$ such that $H(0,t) = f(t)$ and $H(1,t) = g(t)$ and $H(s,0) = a$ and $H(s,1) = b$.

Liberty, equality, connectedness

In archipelago *A*, tradition says that two inhabitants of *A* are equal if there exists a path (over land) that connects them.



Two inhabitants of the same island are equal.

Two inhabitants of different islands are different.
(Unless there exists a bridge between the islands.)

Operations over paths

$$\begin{tikzcd} a \arrow[loop right, "id_a"] \end{tikzcd} \qquad \begin{tikzcd} a \arrow[r, shift left, "f"] & b \arrow[l, shift left, "f^{-1}"] \end{tikzcd} \qquad \begin{tikzcd} a \arrow[r, "f"] \arrow[rr, bend right, "g \circ f"'] & b \arrow[r, "g"] & c \end{tikzcd}$$

Unit: for every point a we have a trivial paths id_a from a to a

$$id_a = t \in [0,1] \mapsto a$$

Inverse: for every path f from a to b, we have a path f^{-1} from b to a

$$f^{-1} = t \in [0,1] \mapsto f(1-t)$$

Composition: for every paths f from a to b and g from b to c, we have a path $g \circ f$ from a to c

$$g \circ f = \begin{cases} t \in [0, \frac{1}{2}] \mapsto f(2t) \\ t \in]\frac{1}{2}, 1] \mapsto g(2t-1) \end{cases}$$

Corollary

The relation “being connected by a path” is an equivalence relation.

A groupoid?

$$
\begin{aligned}
id &: Path(a, a) \\
\cdot^{-1} &: Path(a, b) \to Path(b, a) \\
\cdot \circ \cdot &: Path(b, c) \to Path(a, b) \to Path(a, c)
\end{aligned}
$$

We would like to see here a groupoid, that is,

- a group where the binary operator $\circ$ is partial;
- a category where every arrow has an inverse arrow.

For this, we would need the following identities:

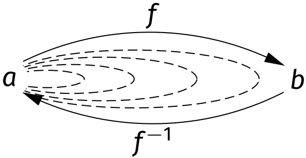
$$
\begin{array}{cc}
f \circ f^{-1} = id & f^{-1} \circ f = id \\
f \circ id = f & id \circ f = f \\
(f \circ g) \circ h = f \circ (g \circ h) &
\end{array}
$$

Equality between paths = homotopy

$$f^{-1} \circ f = id_a \quad \text{for all } f : Path(a, b)$$

Viewed as an equality between functions, this property is false:
id_a is a constant function, while $f^{-1} \circ f$ goes through $a \cdots b \cdots a$.

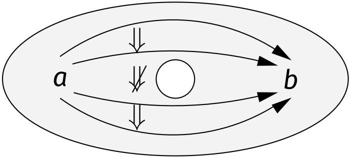
Viewed as an homotopy relation, this property is true:
the paths id_a and $f^{-1} \circ f$ are homotopic.

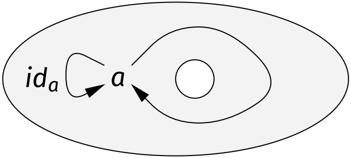


$$H(s,t) = \begin{cases} f(s \times 2t) & \text{if } t \leq \frac{1}{2} \\ f(s \times 2(1-t)) & \text{if } t > \frac{1}{2} \end{cases}$$

Homotopic paths, non-homotopic paths

In general, any two paths from a to b are not homotopic, and a loop (a path from a to a) is not homotopic to id_a.





However, the paths appearing in the groupoid equations are always homotopic, regardless of the topology of space A.
Reading $=$ as "is homotopic to":

$$f \circ f^{-1} = id \qquad f^{-1} \circ f = id$$

$$f \circ id = f \qquad id \circ f = f$$

$$(f \circ g) \circ h = f \circ (g \circ h)$$

Turtles all the way down

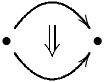
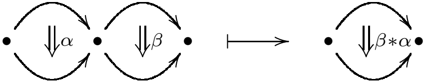
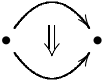
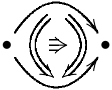
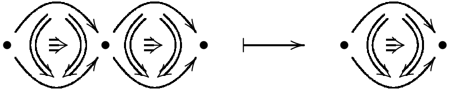
Homotopies between paths are themselves a groupoid, comprising

- unit homotopies id_f (where f is a given path);
- a composition law;
- an inverse.

These operations satisfy the groupoid laws provided equality between homotopies is interpreted as existence of a “level 2” homotopy:
a continuous function $\Phi : [0,1]^3 \to A$ such that $\Phi(0, \_, \_)$ is equal to the first homotopy and $\Phi(1, \_, \_)$ is equal to the second.

We can iterate this construction for every level k, obtaining an ω-groupoid or ∞-groupoid.

The first three levels

Level	Objects	Identities	Compositions (transitivity)
0	$\bullet$	$\bullet \longrightarrow \bullet$	$\bullet \xrightarrow{f} \bullet \xrightarrow{g} \bullet \quad \longmapsto \quad \bullet \xrightarrow{g\circ f} \bullet$
1	$\bullet \longrightarrow \bullet$		
2			

(Source: Cheng and Lauda, *Higher-Dimensional Categories: an illustrated guide book*.)

Type theory and homotopy theory

The presentation of type theory in type theory (type $x =_A y$, constructor $\mathtt{refl}_A$, eliminator J_A) naturally generates an ω-groupoid for every type A.
(van den Berg and Garner, 2008; Lumsdaine, 2009)

Conversely, ω-groupoids and higher-order homotopy provide models of type theory with equality.

For instance, Hofmann and Streicher (1998) used 1-groupoids to construct a model where UIP is false, that is, where there exists two different proofs for an equality $a = b$.

IV

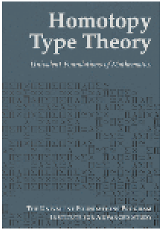
Homotopy type theory

Homotopy type theory

Very briefly: it is a type theory, close to that of Martin-Löf, but explained, revised and extended in the light of higher-order homotopy.

Origin: a recent (2005–2010) encounter between a mathematician (Voevodsky), category theorists (Awodey, Warren, …) and computer scientists (Streicher, Coquand, …).

Reference book: the collective book *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013, available on the Web.



Homotopy
Type Theory

The types of HoTT

HoTT starts with the same types as MLTT:

$$\begin{array}{rll} A, B ::= & \texttt{empty} \mid \texttt{unit} \mid \texttt{bool} & \text{enumerated types (0, 1, 2 elements)} \\ & \mid \Pi x : A.\, B \mid \Sigma x : A.\, B & \text{dependent products and sums} \\ & \mid A + B & \text{sums} \\ & \mid a =_A b & \text{identities (equality proofs)} \\ & \mid U & \text{names of universes} \end{array}$$

Standard abbreviations: $A \to B$ is $\Pi\_ : A.\, B$; $\quad A \times B$ is $\Sigma\_ : A.\, B$.

Classifying types according to their equalities

We distinguish two important families of types:

- Propositions (*mere propositions* in the book).
  These are the types A where all values are equal:
  $prop(A) \overset{def}{=} \Pi x, y : A.\, x =_A y.$
- Sets
  These are the types A such that identities are unique:
  $set(A) \overset{def}{=} \Pi x, y : A.\, prop(x =_A y) \overset{def}{=} \Pi x, y : A.\, \Pi p, q : x =_A y.\, p = q$

In other words: the sets are the types that already satisfy UIP, and the propositions are the types that already satisfy PI.

Examples of propositions

The following are propositions:

- `unit` ($\approx$ truth)
- `empty` ($\approx$ absurdity)
- $A \to B$ if B is a proposition (+ FE axiom)
- $A \to$ `empty` ($\approx$ negation) (+ FE axiom)
- $\Pi x : A.\, B(x)$ if $B(x)$ is a proposition for all $x : A$ (+ DFE axiom)
- $A \times B$ if A and B are propositions.
- $A + B$ if A and B are propositions and $A \to B \to$ `empty`.

$A + B$ is not a proposition in general:
e.g. `unit` $+$ `unit` has two different values, `inl tt` and `inr tt`.

$\Sigma x : A.\, B(x)$ is not a proposition in general.

Examples of sets

- Enumerated types: `empty`, `unit`, `bool`.
- $A \to B$ if B is a set (+ FE axiom)
- $\Pi x : A.\, B(x)$ if $B(x)$ is a set for all $x : A$ (+ DFE axiom)
- $A \times B$ and $A + B$ if A and B are sets.
- $\Sigma x : A.\, B(x)$ if A and $B(x)$ for all $x : A$ are sets
- A if A is a proposition.

A propositions as types correspondence

We'd like to represent propositions from higher-order logic as types that are propositions in the sense of HoTT, that is, types A such that $x = y$ for all $x, y : A$.

We use a propositional truncation operator:
a type $\|A\|$ that is a proposition, for all types A.

$$\begin{aligned}
[\top] &= \texttt{unit} & [\bot] &= \texttt{empty} \\
[P \Rightarrow Q] &= [P] \to [Q] & [\neg P] &= [P] \to \texttt{empty} \\
[P \wedge Q] &= [P] \times [Q] & [P \vee Q] &= \|[P] + [Q]\| \\
[\forall x : A.\, P] &= \Pi x : A.\, [P] & [\exists x : A.\, P] &= \|\Sigma x : A.\, [P]\|
\end{aligned}$$

Propositional truncation

Two operations over type $\|A\|$:

$$\begin{aligned} \texttt{img} &: A \to \|A\| \\ \texttt{lift} &: (A \to B) \to (\|A\| \to B) \quad \text{if } B \text{ is a proposition} \end{aligned}$$

such that $\texttt{img}\, x = \texttt{img}\, y$ for all $x, y : A$ and $\texttt{lift}\, f\, (\texttt{img}\, x) = f\, x$ for all $x : A$.

$\texttt{img}\, a$ erases all information on the value of a. Its result just witnesses that type A is inhabited.

If $f : A \to B$ and B is a proposition, function f returns the same result regardless of its argument $a : A$. All that matters to f is that A is inhabited. We can therefore transform it into a function $\texttt{lift}\, f : \|A\| \to B$.

Propositional truncation

$$\begin{aligned} \texttt{img} &: A \to \|A\| \\ \texttt{lift} &: (A \to B) \to (\|A\| \to B) \quad \text{if } B \text{ is a proposition} \end{aligned}$$

In the encoding of $P \vee Q$ by $\|[P] + [Q]\|$, we hide which of P or Q is true. We cannot, therefore, write a function $f : [P \vee Q] \to$ `bool` that is `true` in the P case and `false` otherwise.

However, `lift` still lets us do a case analysis “P true? Q true?” for the purpose of concluding a proposition R.

$$\frac{p : [P]}{\texttt{img}(\texttt{inl}\ p) : [P \vee Q]} \qquad \frac{q : [Q]}{\texttt{img}(\texttt{inr}\ q) : [P \vee Q]}$$

$$\frac{a : [P \vee Q] \qquad f : [P] \to [R] \qquad g : [Q] \to [R]}{\texttt{lift}\ (\lambda x.\ \texttt{match}\ x\ \texttt{with}\ \texttt{inl}\ p \Rightarrow f\ p \mid \texttt{inr}\ q \Rightarrow g\ q)\ a : [R]}$$

Higher-inductive types (HIT)

In a standard inductive type, constructors generate the values of the type:

```
Inductive nat: Type :=
| O: nat
| S: nat -> nat.
```

A higher-inductive type can also have constructors that generate paths between values of the type, that is, equalities beyond the default equality, and even higher-order paths, that is, equalities between equalities.

```
Inductive Z4: Type :=
| O: Z4
| S: Z4 -> Z4
| mod4: S(S(S(S O))) = O.
```

HIT = inductive types + equations

The definition of $\mathbb{Z}$ as a “free” inductive type:

```
Inductive Z :=
  | Z0: Z
  | Zpos: positive -> Z
  | Zneg: positive -> Z.
```

A definition “with two zeros” and an equation between them:

```
Inductive Z :=
  | Zpos: nat -> Z
  | Zneg: nat -> Z
  | Zzero: Zneg O = Zpos O.
```

$\mathbb{Z}$ generated by `O`, successor (`S`), and its inverse, the predecessor (`P`):

```
Inductive Z :=
  | O: Z    | S: Z -> Z    | P: Z -> Z
  | SP: forall z, S (P z) = z
  | PS: forall z, P (S z) = z.
```

Case analysis over a HIT

```
Inductive Z4: Type :=
| O: Z4
| S: Z4 -> Z4
| mod4: S(S(S(S O))) = O.
```

The declaration gives us an equality “for free”, `mod4`. Now, we must respect this equality in all computations that analyze a value of type `Z4`:

```
match (n: Z4) with O => a | S m => f m end
```

must produce the same result if $n = \mathtt{O}$ and if $n = \mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{S}\ \mathtt{O})))$, hence a proof obligation: $f\ (\mathtt{S}(\mathtt{S}(\mathtt{S}\ \mathtt{O}))) = a$.

```
Definition pred (n: Z4) :=
  match n with
  | O => S(S(S O)) ✔
  | S m => m
  end.
```

```
Definition pred (n: Z4) :=
  match n with
  | O => O   ✘
  | S m => m
  end.
```

Recursors over a HIT

This proof obligation appears in the type of the recursor (the higher-order function that performs case analysis and recursion).

For a standard inductive type such as `nat`, the recursor takes one argument per value constructor:

$$\mathtt{nat\_rec\ :\ }\forall\mathtt{X:\ Type.\ }\underset{\substack{\nearrow\\ \text{case }\mathtt{O}}}{\mathtt{X}}\to(\mathtt{X}\underset{\substack{\nwarrow\\ \text{case }\mathtt{S}}}{\to}\mathtt{X})\to\mathtt{nat}\to\mathtt{X}$$

For a HIT such as `Z4`, the recursor takes one argument per value constructor or path constructor, and in the latter case it's an equality proof.

$$\mathtt{Z4\_rec\ :\ }\forall\mathtt{X:\ Type.\ }\underset{\substack{\nearrow\\ \text{case }\mathtt{O}}}{\forall\mathtt{z}}\mathtt{:\ X.\ }\forall\underset{\substack{\uparrow\\ \text{case }\mathtt{S}}}{\mathtt{s}}\mathtt{:\ X}\to\mathtt{X.\ }\underset{\substack{\uparrow\\ \text{case }\mathtt{mod4}}}{\mathtt{s(s(s(s\ z)))\ =\ z}}\to\mathtt{Z4}\to\mathtt{X}$$

Recursors over a HIT

```
Z4_rec : ∀X: Type. ∀z: X. ∀s: X → X. s(s(s(s z))) = z → Z4 → X
```

Equipped with this recursor, it is easy to define the predecessor function:

```
Definition pred : Z4 → Z4 :=
  Z4_rec Z4 (S(S(S O))) (fun m => m) (eq_refl (S(S(S O)))).
```

Addition is just as easy:
(we follow the schema $\mathtt{add}\ m\ \mathtt{O} = m$ and $\mathtt{add}\ m\ (\mathtt{S}\ n) = \mathtt{S}(\mathtt{add}\ m\ n)$)

```
Definition add (m: Z4) : Z4 → Z4 :=
  Z4_rec Z4 m S (p m).
```

The term `p` must prove $\forall m,\ \mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{S}\ m))) = m$. This can be proved by induction over m, using a dependently-typed recursor that is slightly more complex.

(See the paper by Basold *et al* given in references.)

Truncation as a HIT

Propositional truncation is defined by a very simple HIT:

```
    Inductive tr (A: Type) : Type :=
      | img: A → tr A
      | tr_prop: ∀x y: tr A, x = y.
```

The `tr_prop` constructor asserts that `tr A` is a proposition.

The corresponding recursor is:

```
tr_rec (A: Type) :
   ∀X: Type. ∀i: A → X. (∀x y: X. x = y) → tr A → X
```

We see that it applies only to types `X` that are propositions. But, given a type `X` and a proof `pX: prop X`, we define easily the lifting of a function `A → X`:

```
lift (f: A → X) : tr A → X := tr_rec X f pX
```

Quotient types as a HIT

In set theory, the quotient A/R of a set A by an equivalence relation R over A is the set of all equivalence classes of R.

Given $A : \mathtt{Type}$ and $R : A \to A \to \mathtt{Type}$, we can define a quotient type A/R by the following HIT:

```
Inductive Q : Type :=
  | img: A → Q
  | img_eq: forall x y, R x y → img x = img y.
```

Assuming that R is an equivalence relation, we can prove the converse of `img_eq`, which shows that

```
forall x y,  img x = img y <-> R x y.
```

Quotient types as a HIT

```
Inductive Q : Type :=
  | img: A → Q
  | img_eq: forall x y, R x y → img x = img y.
```

Given a function $f : A \to B$ that is compatible with R (i.e. $R\,x\,y \Rightarrow f\ x = f\ y$), we want to construct a function $g : Q \to B$:

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "\mathtt{img}"'] & B \\ Q \arrow[ur, "g"'] & \end{tikzcd}$$

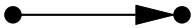
It suffices to take

```
Definition g (q: Q) := match q with img a => f a end
```

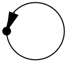
or, more exactly, the equivalent formulation using the recursor `Q_rec`. It follows that $g\ (\mathtt{img}\ a) = f\ a$ for all $a : A$, as expected.

HITs for homotopy

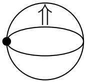
HITs make it possible to describe topological spaces in a purely synthetic manner:



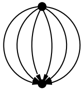
Interval



Circle



Sphere



Suspension

```
  0: I
  1: I
seg: 0=1
```

base: S2
surf: $\mathtt{refl}_{base}$ = $\mathtt{refl}_{base}$

```
base: S1
loop: base = base
```

```
    N: Susp
    S: Susp
merid: A -> N = S
```

V

Advanced topics

Equivalences and univalence

$f : A \to B$ is an equivalence if it is a bijection that "behaves well" with respect to equality paths:

$$\Pi y : B.\, \mathit{fibr}(y) \times \mathit{prop}(\mathit{fibr}(y)) \quad \text{with} \quad \mathit{fibr}(y) = \Sigma x : A.\, f\ x = y$$

We write $A \approxeq B$ if there exists an equivalence from A to B.

Voevodsky's univalence axiom:
the canonical function $A = B \to A \approxeq B$ is an equivalence.
Consequently, if $A \approxeq B$, then $A = B$.

Formalizes the intuitive idea of reasoning up to isomorphism
(not always valid in set theory).

Implies the usual extensionality axioms: **FE**, **DFE**, **PE**.

Computational content still unclear. (⇒ seminar by Th. Coquand)

HoTT for programming languages

The notion of equivalence as a very precise characterization of “good” representation changes: not just a bijection between two types, but a bijection that correctly “transports” equalities.

Higher-inductive types as a new tool to write “correct by construction” programs, in a ways that differs from but complements dependently-typed programming.

All this potential remains to be realized: no full implementation yet of HoTT + HIT; partial prototypes in Agda, Coq, and Lean.

VI

Further reading

Further reading

The reference book:

- *Homotopy Type Theory: Univalent Foundations of Mathematics*, The Univalent Foundations Program, Institute for Advanced Study, 2013, `https://homotopytypetheory.org/book/` To read first: chapters 1, 2, 3 + chapter 6 for HITs.

A rather short presentation of HITs with nice examples:

- Henning Basold, Herman Geuvers, Niels van der Weide: *Higher Inductive Types in Programming*. J. UCS 23(1): 63-88 (2017).

Libraries to work with HoTT:

- In Agda: `https://github.com/HoTT/HoTT-Agda`
- In Coq: `https://github.com/HoTT/HoTT`
- In Lean: `https://github.com/leanprover/lean2/`