polymorphism and impredicativity

logical verification

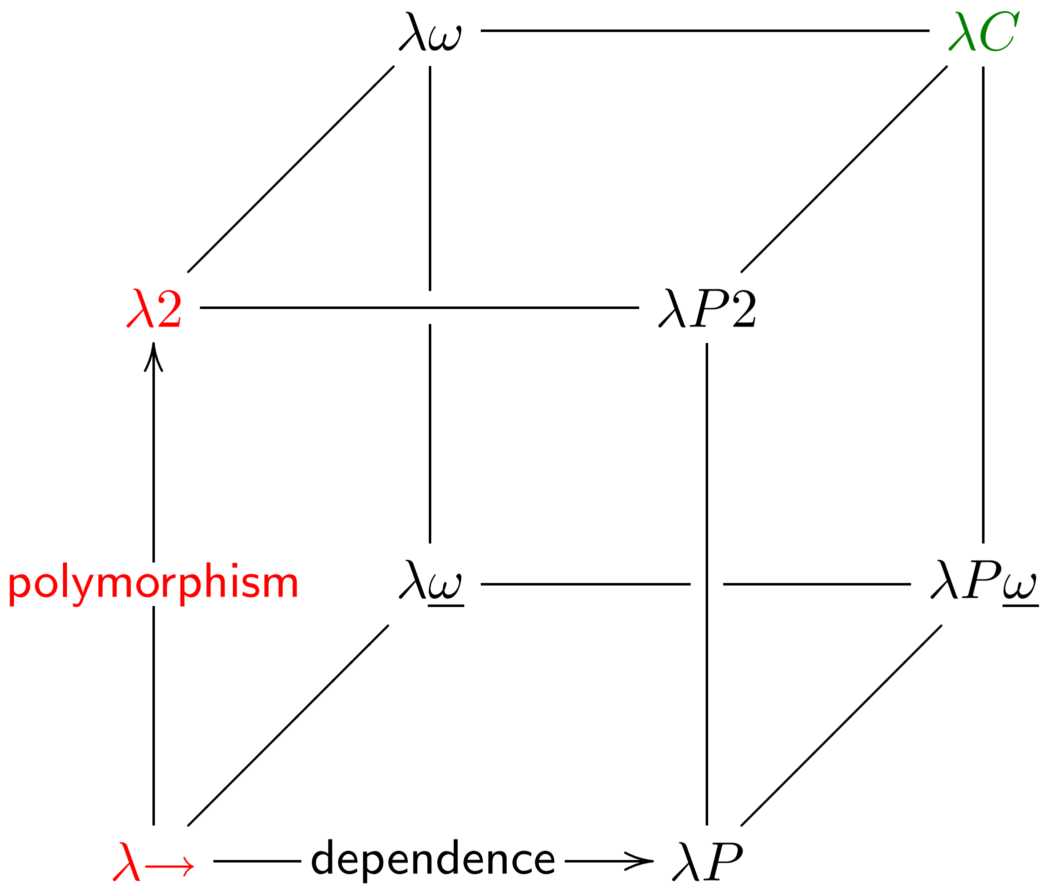
week 10

2004 11 17

the course

propositional logic	$\leftrightarrow$	**simple** type theory
		$\lambda\!\!\rightarrow$
predicate logic	$\leftrightarrow$	type theory with **dependent types**
		λP
2nd order propositional logic	$\leftrightarrow$	**polymorphic** type theory
		$\lambda 2$

the lambda cube



polymorphism

we had functions & quantification over the **elements** of a type

$$\lambda n : \texttt{nat} . \cdots$$

$$\forall n : \texttt{nat} . \cdots$$

```
   fun n : nat => ...
forall n : nat, ...
```

we now add functions & quantification over the **types** themselves

$$\lambda A : * . \cdots$$

$$\forall A : * . \cdots$$

```
   fun A : Set => ...
forall A : Set, ...
```

dependent types versus polymorphism

- **dependent types**

  types can take terms as an argument

- **polymorphism**

  terms can take types as an argument

the polymorphic identity

natid

identity function on the natural numbers

$$\lambda n : \mathsf{nat} \,.\, n$$

```
Definition natid : nat -> nat :=
  fun n : nat => n.
```

```
Check (natid 0).
Eval compute in (natid 0).
```

boolid

identity function on the booleans

$$\lambda b : \mathsf{bool} \,.\, b$$

```
Definition boolid : bool -> bool :=
  fun b : bool => b.
```

```
Check (boolid true).
Eval compute in (boolid true).
```

polyid

polymorphic identity function

$$\lambda A : *. \lambda x : A. x \quad : \quad \Pi A : *. A \to A$$

```
Definition polyid : forall A : Set, A -> A :=
  fun A : Set => fun x : A => x.
```

```
Check (polyid nat 0).
Check (polyid bool true).
```

```
Eval compute in (polyid nat 0).
Eval compute in (polyid bool true).
```

Notation

```
Check (polyid nat 0).
Check (polyid _ 0).


Notation id := (polyid _).


Check (id 0).
Check (id true).

Eval compute in (id 0).
Eval compute in (id true).
```

natlist

```
Inductive natlist : Set :=
  natnil : natlist
| natcons : nat -> natlist -> natlist.
```

$$3, 1, 4, 1, 5, 9, 2, 6$$

natlist_dep

generalizing natlist to a **dependent** type

```
Inductive natlist_dep : (nat -> Set) :=
  natnil_dep : (natlist_dep O)
| natcons_dep : forall n : nat,
    nat -> (natlist_dep n) -> (natlist_dep (S n)).
```

boollist

```
Inductive boollist : Set :=
  boolnil : boollist
| boolcons : bool -> boollist -> boollist.
```

$$F, T, F, F, F, T, T, F$$

polylist

generalizing `natlist` to a **polymorphic** type

```
Inductive polylist (A : Set) : Set :=
  polynil : (polylist A)
| polycons : A -> (polylist A) -> (polylist A).
```

```
polylist : forall A : Set, Set
polylist : Set -> Set
```

```
polynil  : forall A : Set, (polylist A)
polycons : forall A : Set, A -> (polylist A) -> (polylist A)
```

examples of polymorphic lists

$$3, 1$$

```
polycons nat 3 (polycons nat 1 (polynil nat))
```

$$F, T$$

```
polycons bool false (polycons bool true (polynil bool))
```

. . . and now in stereo!

```
Inductive polylist_dep (A : Set) : nat -> Set :=
  polynil_dep : (polylist_dep A O)
| polycons_dep : forall n : nat,
    A -> (polylist_dep A n) -> (polylist_dep A (S n)).
```

Notation

```
Notation ni := (polynil _).
Notation co := (polycons _).


Check (co 3 (co 1 ni)).
Check (co false (co true ni)).

Check (co 3 (co true ni)).
```

. . . and even more polymorphic

```
Inductive polylist' : Type :=
  polynil' : polylist'
| polycons' : forall A : Set, A -> polylist' -> polylist'.
```

```
polycons' nat 3 (polycons' bool true polynil')
```

length of a list

natlength

```
Fixpoint natlength (l : natlist) {struct l} : nat :=
  match l with
    natnil => O
  | natcons h t => S (natlength t)
  end.
```

```
natlength : natlist -> nat
```

polylength

```
Fixpoint polylength
    (A : Set) (l : polylist A) {struct l} : nat :=
  match l with
    polynil    => O
  | polycons   h t => S (polylength A t)
  end.
```

```
polylength : forall A : Set, polylist A -> nat
```

polylength

```
Fixpoint polylength' (l : polylist') {struct l} : nat :=
  match l with
    polynil' => O
  | polycons' A h t => S (polylength' t)
  end.
```

```
polylength' : polylist' -> nat
```

applying a function to each element of a list

natmap

```
Fixpoint natmap
    (f : nat -> nat) (l : natlist) {struct l} : natlist :=
  match l with
    natnil => natnil
  | natcons h t => natcons (f h) (natmap f t)
  end.
```

```
natmap : (nat -> nat) -> natlist -> natlist
```

`natmap` $(\lambda n.\, n + 10)\,(3, 1, 4, 1, 5, 9, 2, 6) = (13, 11, 14, 11, 15, 19, 12, 16)$

polymap

```
Fixpoint polymap (A : Set)
    (f : A -> A) (l : polylist A) {struct l} : polylist A :=
  match l with
    polynil => polynil A
  | polycons h t => polycons A (f h) (polymap A f t)
  end.
```

```
polymap :
  forall A : Set, (A -> A) -> polylist A -> polylist A
```

. . . and even more polymorphic

```
Fixpoint polymap' (A B : Set)
    (f : A -> B) (l : polylist A) {struct l} : polylist B :=
  match l with
    polynil => polynil B
  | polycons h t => polycons B (f h) (polymap' A B f t)
  end.
```

```
polymap' :
  forall A B : Set, (A -> B) -> polylist A -> polylist B
```

iterating an operation over a list

natfold

```
Fixpoint natfold (f : nat -> nat -> nat) (z : nat)
    (l : natlist) {struct l} : nat :=
  match l with
    natnil => z
  | natcons h t => f h (natfold f z t)
  end.
```

```
natfold : (nat -> nat -> nat) -> nat -> natlist -> nat
```

$$\text{natfold}\ f\ z\ (3, 1, 4, 1) = f\ 3\ (f\ 1\ (f\ 4\ (f\ 1\ z)))$$

$$\text{natfold} * z\ (3, 1, 4, 1) = 3 * 1 * 4 * 1 * z$$

sum

```
Definition sum := natfold plus 0.

sum : natlist -> nat


Eval compute in
  sum (natcons 3 (natcons 1 (natcons 4 (natcons 1 natnil)))).
```

$$\text{natfold}\ (+)\ 0\ (3, 1, 4, 1) = 3 + 1 + 4 + 1 + 0 = 9$$

polyfold

```
Fixpoint polyfold (A B : Set) (f : A -> B -> B) (z : B)
    (l : polylist A) struct l : B :=
  match l with
    polynil => z
  | polycons h t => f h (polyfold A B f z t)
  end.
```

```
polyfold :
  forall A B : Set, (A -> B -> B) -> B -> polylist A -> B
```

sum defined with polyfold

```
Definition sum' := polyfold nat nat plus O.

Definition sum' := polyfold _ _ plus O.


sum' : polylist nat -> nat


Eval compute in sum' (co 3 (co 1 (co 4 (co 1 ni)))).
```

Russell's paradox

Cantor: power set is bigger than the set itself

naive set theory

$$\mathcal{P}(\mathsf{Set}) \subseteq \mathsf{Set}$$

$$\{x \mid x \notin x\} \in \{x \mid x \notin x\}$$
$$\Updownarrow$$
$$\{x \mid x \notin x\} \notin \{x \mid x \notin x\}$$

inconsistent

impredicativity

$\lambda 2$ is impredicative

$$\mathsf{bool} : * \;\vdash\; * \to \mathsf{bool} \;:\; *$$

$$\mathcal{P}(\mathsf{Set}) \in \mathsf{Set}$$

$$\vdash\; (\Pi A : *.\, A) \;:\; *$$

Coq

`Prop` is impredicative, `Set` is predicative

```
(forall A : Prop, A) : Prop
(forall A : Set , A) : Type
```

$\lambda 2$ is inconsistent with classical mathematics

$$\mathsf{bool} : * \;\vdash\; \Pi A : *. (A \to \mathsf{bool}) \;:\; *$$

'set of functions that map **each** set into a subset of that set'

$$U := \prod_{A \in \mathsf{Set}} \mathcal{P}(A) \in \mathsf{Set}$$

$$U = \mathcal{P}(U) \times \ldots$$

the paradox

$$U := \prod_{A \in \mathsf{Set}} \mathcal{P}(A) \in \mathsf{Set}$$

if $u \in U$ then for each $A \in \mathsf{Set}$ holds that $u(A) \in \mathcal{P}(A)$

in particular $u(U) \in \mathcal{P}(U)$

$$X := \{u \in U \mid u \notin u(U)\} \in \mathcal{P}(U)$$

take some $x \in U$ such that $x(U) = X$

$$x \in x(U) \;\Leftrightarrow\; x \in X \;\Leftrightarrow\; x \notin x(U)$$