dependent types

logical verification

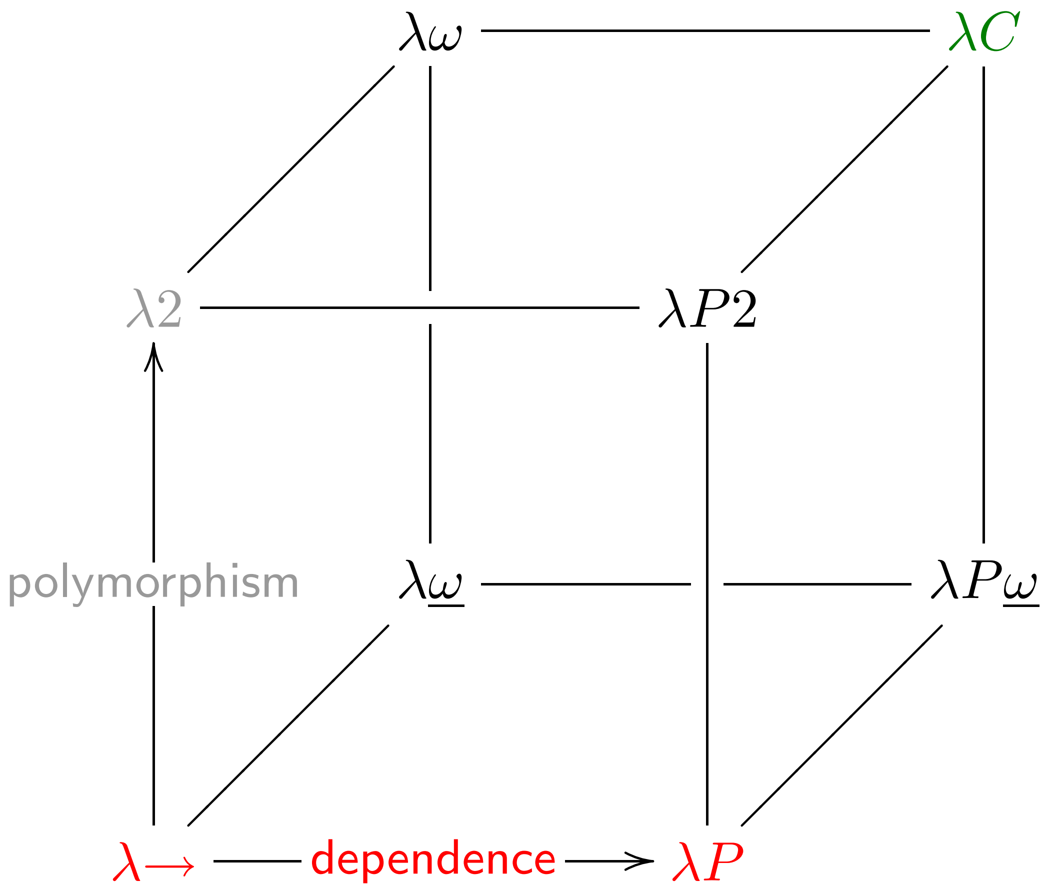
week 8

2004 11 03

Curry-Howard-de Bruijn continued

propositional logic	$\leftrightarrow$	**simple** type theory
		$\lambda{\to}$
predicate logic	$\leftrightarrow$	type theory with **dependent types**
		λP
2nd order propositional logic	$\leftrightarrow$	**polymorphic** type theory
		$\lambda 2$

Barendregt's lambda cube



recap predicate logic

syntax

terms

- $f(M_1, \ldots, M_n)$
- x

formulas

- $P(M_1, \ldots, M_n)$
- $\top$
- $\bot$
- $\neg A$
- $A \to B$
- $A \land B$
- $A \lor B$
- $\forall x.\, A$
- $\exists x.\, A$

rules

introduction rules	**elimination rules**
$I\top$	
	$E\bot$
$I[x]\neg$	$E\neg$
$I[x]\rightarrow$	$E\rightarrow$
$I\wedge$	$El\wedge$ $Er\wedge$
$Il\vee$ $Ir\vee$	$E\vee$
$I\forall$	$E\forall$
$I\exists$	$E\exists$

example

$$(\forall x.\, P(x)) \rightarrow \neg \exists y.\, \neg P(y)$$

natlist

```
Inductive natlist : Set :=
    nil : natlist
  | cons : nat -> natlist -> natlist.
```

append & reverse

```
Fixpoint append (k l : natlist) {struct k} : natlist :=
  match k with
    nil => l
  | cons h t => cons h (append t l)
  end.

Fixpoint reverse (k : natlist) : natlist :=
  match k with
    nil => nil
  | cons h t => append (reverse t) (cons h nil)
  end.
```

lists of a given length

```
Fixpoint zeroes (n : nat) : natlist :=
  match n with
    O => nil
  | S n' => cons 0 (zeroes n')
  end.
```

$$
\begin{aligned}
0 &\mapsto \\
1 &\mapsto 0 \\
2 &\mapsto 0, 0 \\
3 &\mapsto 0, 0, 0 \\
4 &\mapsto 0, 0, 0, 0
\end{aligned}
$$

dependent lists

we will define a type for lists of a given length

```
a : (natlist_dep 6)
```

this corresponds in normal programming languages to something like

```
int a[6]
```

the type of a is called a **dependent** type

it **depends** on the natural number 6

```
natlist_dep : nat -> Set
natlist_dep 6 : Set
```

natlist_dep

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

$$3, 1, 4, 1, 5, 9$$

$$\downarrow$$

```
cons 5 3 (cons 4 1 (cons 3 4 (cons 2 1 (cons 1 5 (cons 0 9 nil)))))
                                                  : (natlist_dep 6)
```

zeroes for dependent lists

```
Fixpoint zeroes (n : nat) : natlist_dep n :=
  match n return natlist_dep n with
    O => nil
  | S n' => cons n' O (zeroes n')
  end.
```

the type of dependent zeroes

~~`zeroes : nat -> natlist_dep ?`~~

`zeroes : forall n : nat, natlist_dep n`

dependent product

generalizes the notion of function type

function types and dependent products

$$A \to B$$

$$\downarrow$$

$$\Pi x : A.\, B$$

```
A -> B
```

$$\downarrow$$

```
forall x : A, B
```

append for dependent lists

```
Fixpoint append (n m : nat)
  (k : natlist_dep n) (l : natlist_dep m) {struct k} :
    natlist_dep (plus n m) :=
  match k
    in natlist_dep n return natlist_dep (plus n m)
  with
    nil => l
  | cons n' h t => cons (plus n' m) h (append n' m t l)
  end.
```

reverse for dependent lists

```
Fixpoint reverse
  (n : nat) (k : natlist_dep n) {struct k} :
    natlist_dep n :=
  match k in natlist_dep n return natlist_dep n with
    nil => nil
  | cons n' h t =>
      eq_rec (plus n' 1) (fun n => natlist_dep n)
        (append n' 1 (reverse n' t) (cons 0 h nil))
        (S n') (plus_one n')
  end.
```

has type `natlist_dep (plus n' 1)`

but should have type `natlist_dep (S n')`

reduction of dependent types

```
append nil l  :  natlist_dep (plus O m)
```

$$\downarrow_{\beta\delta\iota}$$

```
           l  :  natlist_dep m
```

equality and dependent types

```
reverse (cons n' h t)  :  natlist_dep (S n')
```

is not interchangable with

```
append (reverse n' t)  :  natlist_dep (plus n' 1)
       (cons 0 h nil)
```

Brouwer-Heyting-Kolmogorov for implication

proof of $A \to B$

is defined to be

function that maps proofs of A to proofs of B

Brouwer-Heyting-Kolmogorov for universal quantification

proof of $\forall x.\, P(x)$

is defined to be

function that maps objects x to proofs of $P(x)$

unification of terms and types

$\lambda{\rightarrow}$ terms and types are separate worlds

$$\lambda x.\, x$$

$$A \rightarrow B$$

λP terms and types are the same kind of expression

$$\lambda x.\, A \rightarrow B$$

syntax

- **variables**

  $x,\ y,\ z,\ \ldots$

- **lambda abstraction**

  $\lambda x : M.\,N$

- **function application**

  $M N$

- **dependent product**

  $\Pi x : M.\,N$

- **two 'sorts'**

  $*$ and $\Box$

rules

next week

different notations for dependent products

syntactic alternatives

$$\Pi x : M.\, N$$

$$\forall x : M.\, N$$

just different ways of writing exactly the same term

coq syntax

```
forall x : M, N
```

Prop, Set and Type in λP

$$\begin{aligned} \texttt{Prop} &\longrightarrow * \\ \texttt{Set} &\longrightarrow * \\ \texttt{Type} &\longrightarrow \Box \end{aligned}$$

$$\begin{aligned} \texttt{Prop : Type} &\longrightarrow * : \Box \\ \texttt{Set : Type} &\longrightarrow * : \Box \end{aligned}$$

terms and formulas of minimal predicate logic in λP

$$
\begin{aligned}
f(M_1, \ldots, M_n) \quad &\to \quad f\, M_1 \ldots M_n \\
x \quad &\to \quad x
\end{aligned}
$$

$$
\begin{aligned}
P(M_1, \ldots, M_n) \quad &\to \quad P\, M_1 \ldots M_n \\
A \to B \quad &\to \quad \Pi u : A.\, B \\
\forall x. A \quad &\to \quad \Pi x : \mathtt{Terms}.\, A
\end{aligned}
$$

where

$$\mathtt{Terms} : *$$

$$
\begin{aligned}
f &: \Pi x_1 : \mathtt{Terms}.\, \ldots\, \Pi x_n : \mathtt{Terms}.\, \mathtt{Terms} \\
P &: \Pi x_1 : \mathtt{Terms}.\, \ldots\, \Pi x_n : \mathtt{Terms}.\, *
\end{aligned}
$$

proofs in λP: implication

$$\dfrac{\begin{array}{c}[A^u] \\ \vdots \\ B\end{array}}{A \to B}\ I[u]{\to} \qquad \lambda u : A.\, M \ : \ \Pi u : A.\, B$$

with $M \ : \ B$

$$\dfrac{\begin{array}{c}\vdots \\ A \to B\end{array} \qquad \begin{array}{c}\vdots \\ A\end{array}}{B}\ E{\to} \qquad M N \ : \ B$$

with $M \ : \ \Pi u : A.\, B$
$N \ : \ A$

proofs in λP: universal quantification

$$\dfrac{\begin{array}{c}\vdots\\ A\end{array}}{\forall x.\, A}\; I\forall \qquad\qquad \lambda x : \texttt{Terms}.\, M \;:\; \Pi x : \texttt{Terms}.\, A$$

with $\quad M \;:\; A$

$$\dfrac{\begin{array}{c}\vdots\\ \forall x.\, A\end{array}}{A[x := N]}\; E\forall \qquad\qquad M\, N \;:\; A[x := N]$$

with $\quad M \;:\; \Pi x : \texttt{Terms}.\, A$

$\quad N \;:\; \texttt{Terms}$

example 1

$$\forall x.\,(P(x) \to (\forall y.\,P(y) \to A) \to A)$$

example 2

$$(\forall x.\, P(x) \longrightarrow Q(x)) \longrightarrow (\forall x.\, P(x)) \longrightarrow \forall y.\, Q(y)$$

`printf`

what is the type of `printf`?

```
printf("hello\n")
printf("%d\n", 3)
printf(s, ...)
```

$$\begin{aligned}\texttt{printf} &: \texttt{string} \to \texttt{unit}\\ \texttt{printf} &: \texttt{string} \to \texttt{int} \to \texttt{unit}\\ \texttt{printf} &: \Pi s : \texttt{string}\,.\,\texttt{printftype}\ s\end{aligned}$$

$$\texttt{printftype} : \texttt{string} \to \mathsf{Set}$$

printftype

```
Fixpoint printftype (s : string) : Set :=
  match s with
    nil => unit
  | cons '%' (cons 'd' t) => int -> printftype t
  | cons '%' (cons 'c' t) => char -> printftype t
  | cons '%' (cons 'f' t) => float -> printftype t
  | cons '%' (cons 's' t) => string -> printftype t
  | ...
  | cons _ t => printftype t
  end.
```

dependently typed functional programming languages

- **cayenne**

  'dependent haskell'

  Lennart Augustsson

- **dependent ML**

  Hongwei Xi

- **epigram**

  Conor McBride

the religion of dependent types

Why do dependent types matter? Types matter. That's what they're for – to classify data with respect to criteria which matter: how they should be stored in memory, whether they can be safely passed as inputs to a given operation, even who is allowed to see them. Dependent types are types expressed in terms of data, explicitly relating their inhabitants to that data. As such, they enable you to express more of what matters about data. Dependent types are better at mattering on your behalf, and that is why I hope they might matter to you.

– Conor McBride

the three main things we have seen today

- **dependent types in programming**
- **generalizing function types to dependent products**

$$A \to B$$

$$\downarrow$$

$$\Pi x : A.\, B$$

- **Curry-Howard-de Bruijn for predicate logic**