Last time on **Types**...

- Results on reduction (semantics) of PLC

Properties of PLC beta-reduction on typeable expressions

Suppose $\Gamma \vdash M : \tau$ is provable in the PLC type system. Then the following properties hold:

Subject Reduction. If $M \to M'$, then $\Gamma \vdash M' : \tau$ is also a provable typing.

Church Rosser Property. If $M \to^* M_1$ and $M \to^* M_2$, then there is M' with $M_1 \to^* M'$ and $M_2 \to^* M'$.

Strong Normalisation Property. There is no infinite chain $M \to M_1 \to M_2 \to \ldots$ of beta-reductions starting from M.

Last time on **Types**...

- Results on reduction (semantics) of PLC
- Encoding data types in PLC (part 1), *bool*

Polymorphic booleans

$$bool \stackrel{\text{def}}{=} \forall \alpha\, (\alpha \to (\alpha \to \alpha))$$

$$True \stackrel{\text{def}}{=} \Lambda \alpha\, (\lambda\, x_1 : \alpha, x_2 : \alpha\, (x_1))$$

$$False \stackrel{\text{def}}{=} \Lambda \alpha\, (\lambda\, x_1 : \alpha, x_2 : \alpha\, (x_2))$$

$$if \stackrel{\text{def}}{=} \Lambda \alpha\, (\lambda\, b : bool, x_1 : \alpha, x_2 : \alpha\, (b\, \alpha\, x_1\, x_2))$$

This time on **Types**...

- Encoding data types in PLC (part 2), *list*

This time on **Types**...

- Encoding data types in PLC (part 2), *list*
- Dependent type theory

A tautology checker

$$f : \underbrace{bool \to bool \to \cdots bool \to}_{n \text{ arguments}} bool$$

$$\texttt{fun}\ taut\ x\ f = \texttt{if}\ x = 0\ \texttt{then}\ f\ \texttt{else}$$
$$(taut(x-1)(f\ \texttt{true}))$$
$$\texttt{andalso}\ (taut(x-1)(f\ \texttt{false}))$$

Defining types $n\,AryBoolOp$ for each natural number $n \in \mathbb{N}$

$$\begin{cases} 0\,AryBoolOp & \stackrel{\text{def}}{=} bool \\ (n+1)\,AryBoolOp & \stackrel{\text{def}}{=} bool \to (n\,AryBoolOp) \end{cases}$$

then $taut\,n$ has type $(n\,AryBoolOp) \to bool$, i.e. the result type of the function *taut depends upon the value of its argument*.

The tautology checker in Agda

```
data Bool : Set where
  True : Bool
  False : Bool

_and_ : Bool -> Bool -> Bool
True and True = True
True and False = False
False and _ = False

data Nat : Set where
  Zero : Nat
  Succ : Nat -> Nat

_AryBoolOp : Nat -> Set
Zero AryBoolOp = Bool
(Succ n) AryBoolOp = Bool -> n AryBoolOp

taut : (n : Nat) -> n AryBoolOp -> Bool
taut Zero f = f
taut (Succ n) f = taut n (f True) and taut n (f False)
```

Dependent function types $(x : \tau) \to \tau'$

$$\frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda\, x : \tau\, (M) : (x : \tau) \to \tau'} \quad \text{if } x \notin \mathit{dom}(\Gamma) \cup \mathit{fv}(\Gamma)$$

$$\frac{\Gamma \vdash M : (x : \tau) \to \tau' \qquad \Gamma \vdash M' : \tau}{\Gamma \vdash M\, M' : \tau'[M'/x]}$$

τ' may 'depend' on x, i.e. have free occurrences of x.

(Free occurrences of x in τ' are bound in $(x : \tau) \to \tau'$.)

Dependent type systems feature rules like

$$\frac{\Gamma \vdash M : \tau \qquad \tau \approx \tau'}{\Gamma \vdash M : \tau'}$$

(E.g. $(1+1)\,\mathrm{AnyBoolOp} \approx 2\,\mathrm{AnyBoolOp}$)

For decidability, need $\tau \approx \tau'$ to be a decidable relation between type expressions.

Polymorphic lists

$$\alpha\ \mathit{list} \stackrel{\text{def}}{=} \forall\,\beta\,(\beta \to (\alpha \to \beta \to \beta) \to \beta)$$

$$\mathit{Nil} \stackrel{\text{def}}{=} \Lambda\,\alpha,\beta\,(\lambda\,n : \beta, c : \alpha \to \beta \to \beta\,(n))$$

$$\begin{aligned}\mathit{Cons} \stackrel{\text{def}}{=} \Lambda\alpha\,(\lambda x : \alpha, \ell : \alpha\ \mathit{list}\ (&\\ &\Lambda\beta\,(\lambda n : \beta, c : \alpha \to \beta \to \beta\ (\\ &\quad c\,x\,(\ell\,\beta\,n\,c)))))\end{aligned}$$

Iteratively defined functions on finite lists

$A^* \stackrel{\text{def}}{=}$ finite lists of elements of the type A
(with constructors *nil* and ::)

Given a type B, an element $n : B$, and a function $c : A \to B \to B$, the iteratively defined function $(\mathit{listIter}\, n\, c) : A^* \to B$ is the unique function satisfying:

$$\begin{aligned} \mathit{listIter}\ n\ c\ \mathit{nil} &\equiv n \\ \mathit{listIter}\ n\ c\ (x :: \ell) &\equiv c\, x\, (\mathit{listIter}\ n\ c\ \ell). \end{aligned}$$

for all $x : A$ and $\ell : A^*$.

$$\mathit{listIter} : \forall B. B \to (A \to B \to B) \to A^* \to B$$

List iteration in PLC

see List.agda on course web page

$$iter \stackrel{\text{def}}{=} \Lambda\alpha, \beta\ (\lambda n : \beta, c : \alpha \to \beta \to \beta(\\ \lambda \ell : \alpha\ list\ (\ell\ \beta\ n\ c)))$$

satisfies:

- $\vdash iter : \forall\, \alpha, \beta\, (\beta \to (\alpha \to \beta \to \beta) \to \alpha\ list \to \beta)$
- $iter\ \alpha\ \beta\ n\ c\ (Nil\ \alpha) \ =_\beta\ n$
- $iter\ \alpha\ \beta\ n\ c\ (Cons\ \alpha\ x\ \ell) \ =_\beta\ c\ x\ (iter\ \alpha\ \beta\ n\ c\ \ell)$

Understanding PLC encoding: abstract over the data constructors

$$\begin{array}{lccccccccc} \textit{e.g. for some list} = & x_1 & :: & (x_2 & :: & (x_3 & :: & \textit{nil} &)) \\ \text{abstract on constructors} & & \downarrow & & \downarrow & & \downarrow & \downarrow & \\ \Lambda\beta\,\lambda(n:\beta)\,(c:\alpha\to\beta\to\beta). & x_1 & c & (x_2 & c & (x_3 & c & n &)) \end{array}$$

Understanding PLC encoding: abstract over the data constructors

$$\begin{array}{lccccccccc}
\textit{e.g. for some list} = & x_1 & :: & (x_2 & :: & (x_3 & :: & \mathit{nil} &)) \\
\text{abstract on constructors} & & \downarrow & & \downarrow & & \downarrow & \downarrow & \\
\Lambda\beta\,\lambda(n:\beta)\,(c:\alpha\to\beta\to\beta). & x_1 & c & (x_2 & c & (x_3 & c & n &))
\end{array}$$

e.g.

$$\begin{array}{lccccccccc}
\texttt{sum = iter } \mathbb{N} \texttt{ + 0} & \Rightarrow & x_1 & + & (x_2 & + & (x_3 & + & 0 &)) \\
\texttt{len = iter } \mathbb{N} \texttt{ inc 0} & & & & & & & & & \\
\quad \textit{where } \texttt{inc = } (\lambda_\lambda r.r+1) & \Rightarrow & x_1 & \texttt{inc} & (x_2 & \texttt{inc} & (x_3 & \texttt{inc} & 0 &)) \\
\texttt{prod = iter } \mathbb{N} \times \texttt{ 1} & \Rightarrow & x_1 & \times & (x_2 & \times & (x_3 & \times & 1 &))
\end{array}$$

PLC encodings of ML algebraic datatypes

ML	PLC
$\alpha_1 * \alpha_2$	$\forall\alpha((\alpha_1 \to \alpha_2 \to \alpha) \to \alpha)$
datatype (α_1, α_2)sum = Inl of α_1 | Inr of α_2	$\forall\alpha((\alpha_1 \to \alpha) \to (\alpha_2 \to \alpha) \to \alpha)$
datatype nat = Zero | Succ of nat	$\forall\alpha(\alpha \to (\alpha \to \alpha) \to \alpha)$
datatype binTree = Leaf | Node of binTree * binTree	$\forall\alpha(\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha)$