□ Simple Churches

$$\begin{aligned} \text{zero} &\triangleq \lambda f.\lambda x.x \\ \text{one} &\triangleq \lambda f.\lambda x.f\,x \\ &\vdots \end{aligned}$$

Simply Typed Versions

$$\text{Nat} \triangleq (() \to ()) \to () \to ()$$

We don't care what the carrier value is, so use ().

$$\begin{aligned} \text{zero} &\triangleq \lambda f_{()\to()}.\lambda x_{()}.x \\ \text{one} &\triangleq \lambda f_{()\to()}.\lambda x_{()}.f\,x \\ &\vdots \end{aligned}$$

□ Simple Functions

$$\begin{array}{rcl} \mathsf{plus} & \triangleq & \lambda m_{\mathsf{Nat}}.\lambda n_{\mathsf{Nat}}.\lambda f_{()\to()}.\lambda x_{()}.m\ f\ (n\ f\ x) \\ \mathsf{plus} & :: & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \end{array}$$

$$\begin{array}{rcl} \mathsf{mult} & \triangleq & \lambda m_{\mathsf{Nat}}.\lambda n_{\mathsf{Nat}}.\lambda f_{()\to()}.\lambda x_{()}.m\ (n\ f)\ x \\ \mathsf{mult} & :: & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \end{array}$$

$$\begin{array}{rcl} \mathsf{mult}' & \triangleq & \lambda m_{\mathsf{Nat}}.\lambda n_{\mathsf{Nat}}.\lambda f_{()\to()}.\lambda x_{()}.m\ (\mathsf{plus}\ n)\ \mathsf{zero}\ f\ x \\ \mathsf{mult}' & \not:: & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \end{array}$$

$$\begin{array}{rcl} \mathsf{exp} & \triangleq & \lambda m_{\mathsf{Nat}}.\lambda n_{\mathsf{Nat}}.\lambda f_{()\to()}.\lambda x_{()}.m\ n\ f\ x \\ \mathsf{exp} & \not:: & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \end{array}$$

$$\begin{array}{rcl} \mathsf{exp}' & \triangleq & \lambda m_{\mathsf{Nat}}.\lambda n_{\mathsf{Nat}}.\lambda f_{()\to()}.\lambda x_{()}.m\ (\mathsf{mult}\ n)\ \mathsf{one}\ f\ x \\ \mathsf{exp}' & \not:: & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \end{array}$$

$$\mathsf{Nat} \triangleq ((() \to ()) \to () \to ())$$

□ Polymorphic Churches

$$\begin{aligned} \text{zero} &\triangleq \lambda f.\lambda x.x \\ \text{one} &\triangleq \lambda f.\lambda x.f\,x \\ &\vdots \end{aligned}$$

Polymorphically typed versions

$$\text{Nat} \triangleq \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$$

$$\begin{aligned} \text{zero} &\triangleq \Lambda\alpha.\lambda f_{\alpha\to\alpha}.\lambda x_{\alpha}.x \\ \text{one} &\triangleq \Lambda\alpha.\lambda f_{\alpha\to\alpha}.\lambda x_{\alpha}.f\,x \\ &\vdots \end{aligned}$$

□ Polymorphic Functions

$$\begin{array}{rcl}
\mathsf{plus} & \triangleq & \lambda m_{\mathsf{Nat}}.\lambda n_{\mathsf{Nat}}.\Lambda\alpha.\lambda f_{\alpha\to\alpha}.\lambda x_{\alpha}.m\,[\alpha]\ f\,(n\,[\alpha]\,f\,x) \\
\mathsf{plus} & :: & \mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}
\end{array}$$

$$\begin{array}{rcl}
\mathsf{mult} & \triangleq & \lambda m_{\mathsf{Nat}}.\lambda n_{\mathsf{Nat}}.\Lambda\alpha.\lambda f_{\alpha\to\alpha}.\lambda x_{\alpha}.m\,[\alpha]\,(n\,[\alpha]\,f)\,x \\
\mathsf{mult} & :: & \mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}
\end{array}$$

$$\begin{array}{rcl}
\mathsf{mult}' & \triangleq & \lambda m_{\mathsf{Nat}}.\lambda n_{\mathsf{Nat}}.\Lambda\alpha.\lambda f_{\alpha\to\alpha}.\lambda x_{\alpha}.m\,[\mathsf{Nat}]\,(\mathsf{plus}\,n)\,\mathsf{zero}\,[\alpha]\,f\,x \\
\mathsf{mult}' & :: & \mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}
\end{array}$$

$$\begin{array}{rcl}
\mathsf{exp} & \triangleq & \lambda m_{\mathsf{Nat}}.\lambda n_{\mathsf{Nat}}.\Lambda\alpha.\lambda f_{\alpha\to\alpha}.\lambda x_{\alpha}.m\,[\alpha\to\alpha]\,(n\,[\alpha])\,f\,x \\
\mathsf{exp} & :: & \mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}
\end{array}$$

$$\begin{array}{rcl}
\mathsf{exp}' & \triangleq & \lambda m_{\mathsf{Nat}}.\lambda n_{\mathsf{Nat}}.\Lambda\alpha.\lambda f_{\alpha\to\alpha}.\lambda x_{\alpha}.m\,[\mathsf{Nat}]\ (\mathsf{mult}\,n)\,\mathsf{one}\,[\alpha]\,f\,x \\
\mathsf{exp}' & :: & \mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Nat}
\end{array}$$

$$\mathsf{Nat} \triangleq \forall\alpha.(\alpha\to\alpha)\to\alpha\to\alpha$$

□ Some Results and Observations

- Evaluation preserves types:

$$\forall e, \tau, v. \quad \cdot \vdash e :: \tau \text{ and } e \hookrightarrow v \quad \text{implies} \quad \cdot \vdash v :: \tau$$

- In absence of fix, evaluation of well-typed functions terminates.

$$\forall e, \tau. \quad \cdot \vdash e :: \tau \quad \text{implies} \quad \exists v.\ e \hookrightarrow v$$

- Church numbers don't use fix.
- Polymorphic types are more powerful than simple types.
- Pure (no fix) polymorphic lambda calculus can encode non primitive recursive functions, such as Ackermann's function, and they are guaranteed to terminate.

□ Implicit Types and Type Inference

- People don't like to see types.
- People don't like to be forced to write types.
- Modern languages try to hide types as much as possible.
- Type inference algorithms reconstruct types.
  We know

$$\lambda x.\lambda f.x + f\,x \quad :: \quad \mathsf{Int} \to (\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int}$$

  from type of constant $+$ (i.e. $+ :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$).

□ Implicit Polymorphism

- Keep polymorphic types.
- Drop polymorphic expression syntax.
- Polymorphic typing rules:

$$\frac{\Gamma \vdash e :: \tau}{\Gamma \vdash e :: \forall \alpha.\tau}\ \text{tp\_tlam} \qquad \frac{\Gamma \vdash e :: \forall \alpha.\tau}{\Gamma \vdash e :: \tau[\alpha := \tau']}\ \text{tp\_tapp}$$

  where α is not free in Γ in tp_tlam.

- Type inference impossible for general polymorphic types.
- Restrict to more tractable polymorphic types.

□ Prenex Polymorphism

- Don't allow embedded quantifiers.

$$\forall\alpha.(\alpha \to \text{Int}) \to \text{Int} \qquad \text{ok}$$
$$(\forall\alpha.\alpha \to \text{Int}) \to \text{Int} \qquad \text{bad}$$

- Allows for type inference.
- Only defined constants (and locally defined) are polymorphic.

```
let map f x = case x of [] -> []; h:t -> f h : map f t
map :: forall a b.(a -> b) -> [a] -> b

map (map (+ 1)) [[1,2],[3,4]] :: [[Int]]

(\m -> m (m (+ 1)) [[1,2],[3,4]]) map :: type error
```

□ GHC Extensions

- Standard Haskell (i.e. Haskell 98) restricts to prenex polymorphism.

- GHC allows for arbitrary polymorphism.

- Sometimes GHC requires help (explicit type annotations) to infer types.

```
(\ (m::forall a.(a -> a) -> [a] -> [a]) -> m (m (+ 1)) [[1,2],[3,4]]) map
:: [[Int]]
```

- GHC has several other interesting extensions...

□ Disjunctive Types

Types to express a choice of two things.

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{inl}\, e \mid \mathsf{inr}\, e \mid (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e \mid \mathsf{inr}\, x \Rightarrow e) \\ \tau & ::= & \ldots \mid \tau \oplus \tau \end{array}$$

$$\frac{\Gamma \vdash e :: \tau_1}{\Gamma \vdash \mathsf{inl}\, e :: \tau_1 \oplus \tau_2}\ \mathsf{tp\_inl} \qquad \frac{\Gamma \vdash e :: \tau_2}{\Gamma \vdash \mathsf{inr}\, e :: \tau_1 \oplus \tau_2}\ \mathsf{tp\_inr}$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \oplus \tau_2 \quad \Gamma \vdash e_2 :: \tau_1 \to \tau_3 \quad \Gamma \vdash e_3 :: \tau_2 \to \tau_3}{\Gamma \vdash \mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_2 \mid \mathsf{inr}\, x \Rightarrow e_3 :: \tau_3}\ \mathsf{tp\_case}$$

$$\frac{e_1 \hookrightarrow \mathsf{inl}\, v_1 \quad e_2\, v_1 \hookrightarrow v_2}{\mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_2 \mid \mathsf{inr}\, x \Rightarrow e_3 \hookrightarrow v_2}\ \mathsf{ev\_caseL}$$

$$\frac{e_1 \hookrightarrow \mathsf{inr}\, v_1 \quad e_3\, v_1 \hookrightarrow v_2}{\mathsf{case}\ e_1\ \mathsf{of}\ \mathsf{inl}\, x \Rightarrow e_2 \mid \mathsf{inr}\, x \Rightarrow e_3 \hookrightarrow v_2}\ \mathsf{ev\_caseR}$$

□ Datatypes

We've already seen disjunctive types.

- Bool $\triangleq () \oplus ()$ tt $\triangleq$ inl() ff $\triangleq$ inr()
- List $\triangleq \forall\alpha.()\oplus(\alpha, \mathsf{List}\alpha)$ nil $\triangleq$ inl() (:) $\triangleq \lambda h.\lambda t.\mathsf{inr}(h,t)$
  (if we allow recursive types)

Can generalize from binary disjuncts to labelled n-ary disjuncts.

Combine with recursive types and we get datatypes.

```
data Bool = True | False
data List a = Nil | Cons a (List a)
data Maybe a = Nothing | Just a
```

□ Case Analysis and Pattern Matching

Datatype constructors may be matched and their arguments implicitly bound.

```
map f x = case x of [] -> []; h:t -> f h : map f t
```

`[]` and `:` are the built-in constructors for lists.
`h` and `t` are bound variables in second alternative.

Functions can be defined by cases:

```
div (Just x) (Just y) = if y == 0 then Nothing else Just (x / y)
div _ _ = Nothing
```

Haskell can check to see if all cases are covered:

```
head (h : t) = h
```

`head` doesn't cover the case for `[]`.

□ Homework 3

Here are some different datatypes for implementing the lambda calculus:

Named variables:

```
data Exp = Var String | Lam String Exp | App Exp Exp
```

Represent variables as strings.

DeBruijn Indices:

```
data Exp = Var Int | Lam Exp | App Exp Exp
```

Variables are pointers to their binding site:

$\lambda f.\lambda g.\lambda x.f\,x\,(g\,x) \equiv \Lambda\Lambda\Lambda\,2\,0\,(1\,0)$

Write a normalization function, and a pretty printer, for each of these representations of the lambda calculus.

Extra credit:

Higher-Order Abstract Syntax:

```
data Exp = Lam (Exp -> Exp) | App Exp Exp
```

Variables are directly represented by Haskell variables.