Outline

Section 1

Recursion and Fixed-point Combinators

Recursion

A recursive function is one that is defined in terms of itself.

$$fac = \lambda x.\, \textbf{if } (\textbf{isZero } x)\ 1\ (x \times fac(x-1))$$

Recursion

A recursive function is one that is defined in terms of itself.

$$fac = \lambda x.\, \textbf{if } (\textbf{isZero } x)\ 1\ (x \times fac(x-1))$$

We could use such an equation to define a function that produces a factorial function given a factorial function:

$$f = \lambda fac.\, \lambda x.\, \textbf{if } (\textbf{isZero } x)\ 1\ (x \times fac(x-1))$$

Recursion

A recursive function is one that is defined in terms of itself.

$$fac = \lambda x.\, \textbf{if } (\textbf{isZero } x)\ 1\ (x \times fac(x-1))$$

We could use such an equation to define a function that produces a factorial function given a factorial function:

$$f = \lambda fac.\, \lambda x.\, \textbf{if } (\textbf{isZero } x)\ 1\ (x \times fac(x-1))$$

Suppose that we have a function fac that satisfies

$$\begin{aligned} fac &=_\beta f(fac) \\ &=_\beta (\lambda fac.\, \lambda x.\, \textbf{if } (\textbf{isZero } x)\ 1\ (x \times fac(x-1)))(fac) \\ &=_\beta \lambda x.\, \textbf{if } (\textbf{isZero } x)\ 1\ (x \times fac(x-1)) \end{aligned}$$

It satisfies the defining equation, and so is a factorial function!

Fixed Points

If x is such that $fx = x$, we say that x is a *fixed point* of f.

Fixed Points

If x is such that $fx = x$, we say that x is a *fixed point* of f.
If we can find fixed points of arbitrary functions, then we can exploit them to define recursive functions:

- Take defining equation: $g = M$
- Convert to defining function: $f = \lambda g.\, M$
- Take fixed point g' of f: $g' = fg'$.

Fixed-point Combinators

What we need is a function $\mathbf{Y}$ that computes fixed-points, i.e.:

$$\mathbf{Y}g =_\beta g(\mathbf{Y}g)$$

Fixed-point Combinators

What we need is a function $\mathbf{Y}$ that computes fixed-points, i.e.:

$$\mathbf{Y}g =_\beta g(\mathbf{Y}g)$$

One such function is Curry's $\mathbf{Y}$ combinator:

$$\mathbf{Y} = \lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx))$$

Y combinator

$$\lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx))$$

Y combinator

$$\begin{aligned}\lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx)) \\ \rightarrow_\beta \lambda f.\,f((\lambda x.\,f(xx))(\lambda x.\,f(xx)))\end{aligned}$$

Y combinator

$$\begin{aligned}
&\lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx)) \\
&\quad \to_\beta \lambda f.\,f((\lambda x.\,f(xx))(\lambda x.\,f(xx))) \\
&\quad \to_\beta \lambda f.\,f(f((\lambda x.\,f(xx))(\lambda x.\,f(xx))))
\end{aligned}$$

Y combinator

$$
\begin{aligned}
&\lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx)) \\
&\quad \to_\beta \lambda f.\,f((\lambda x.\,f(xx))(\lambda x.\,f(xx))) \\
&\quad \to_\beta \lambda f.\,f(f((\lambda x.\,f(xx))(\lambda x.\,f(xx)))) \\
&\quad \to_\beta \lambda f.\,f(f(f((\lambda x.\,f(xx))(\lambda x.\,f(xx)))))
\end{aligned}
$$

Y combinator

$$
\begin{aligned}
&\lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx)) \\
&\quad \to_\beta \lambda f.\,f((\lambda x.\,f(xx))(\lambda x.\,f(xx))) \\
&\quad \to_\beta \lambda f.\,f(f((\lambda x.\,f(xx))(\lambda x.\,f(xx)))) \\
&\quad \to_\beta \lambda f.\,f(f(f((\lambda x.\,f(xx))(\lambda x.\,f(xx))))) \\
&\quad \to_\beta \ldots
\end{aligned}
$$

Y combinator

$$
\begin{aligned}
&\lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx)) \\
&\quad \to_\beta \lambda f.\,f((\lambda x.\,f(xx))(\lambda x.\,f(xx))) \\
&\quad \to_\beta \lambda f.\,f(f((\lambda x.\,f(xx))(\lambda x.\,f(xx)))) \\
&\quad \to_\beta \lambda f.\,f(f(f((\lambda x.\,f(xx))(\lambda x.\,f(xx))))) \\
&\quad \to_\beta \ldots
\end{aligned}
$$

Exercise: check that $\mathbf{Y}g =_\beta g(\mathbf{Y}g)$.

Primitive Recursion

A *primitive recursive* function h (on the natural numbers) is defined by giving cases f and g so that

$$h\,0 = f$$
$$h\,(\mathsf{S}y) = g(hy)$$

Predecessor

Suppose we want to define a predecessor function:

$$\text{pred } n = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

Predecessor

Suppose we want to define a predecessor function:

$$\text{pred } n = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

If we try to define it directly by primitive recursion:

- pred $0 = 0$, so $f = 0$;
- pred $(\mathsf{S}n) = \mathsf{S}(\text{pred } n) \ldots$

Predecessor

Suppose we want to define a predecessor function:

$$\text{pred } n = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

If we try to define it directly by primitive recursion:

- pred $0 = 0$, so $f = 0$;
- pred $(\mathsf{S}n) = \mathsf{S}(\text{pred } n) \ldots$
- $\ldots$ except when $n = 0$

Predecessor

Suppose we want to define a predecessor function:

$$\text{pred } n = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

If we try to define it directly by primitive recursion:

- pred $0 = 0$, so $f = 0$;
- pred $(\mathsf{S}n) = \mathsf{S}(\text{pred } n)\ldots$
- $\ldots$ except when $n = 0$

If we only know pred n, we cannot compute pred $(\mathsf{S}n)$; we need to also know if $n = 0$.

Predecessor

So let's define a helper function pred' that returns a pair $\langle \mathsf{pred}\ n, \mathsf{isZero}\ n \rangle$:

$$\begin{aligned} \mathsf{pred}'\ 0 &= \langle 0, \mathbf{true} \rangle \\ \mathsf{pred}'\ (\mathsf{S}n) &= (\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ 0\ (\mathsf{S}(\pi_1 p)), \mathbf{false} \rangle)(\mathsf{pred}'\ n) \end{aligned}$$

We can then define

$$\mathsf{pred} = \lambda n.\, \pi_1(\mathsf{pred}' n)$$

Primitive Recursion with Church Numerals

Suppose that h is defined by primitive recursion with f as the base case and g as the recursive case:

$$h\,0 = f$$
$$h\,(\mathsf{S}y) = g(hy)$$

We can define this function on Church numerals simply as:

$$h = \lambda n.\, nfg$$

So, for instance,

$$h3 = (\lambda n.\, nfg)(\lambda o, s.\, s(s(so))) =_\beta g(g(gf))$$

Predecessor with Church Numerals

$$\mathsf{pred} = \lambda n.\, \pi_1(\mathsf{pred}' n)$$

Predecessor with Church Numerals

$$\begin{aligned}\mathsf{pred} &= \lambda n.\, \pi_1(\mathsf{pred}' n) \\ &= \lambda n.\, \pi_1((\lambda n.\, n\,(\langle 0, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ 0\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle))n)\end{aligned}$$

Predecessor with Church Numerals

$$
\begin{aligned}
\mathsf{pred} &= \lambda n.\, \pi_1(\mathsf{pred}' n) \\
&= \lambda n.\, \pi_1((\lambda n.\, n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle))n) \\
&=_\beta \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle))
\end{aligned}
$$

Predecessor with Church Numerals

$$
\begin{aligned}
\mathsf{pred} &= \lambda n.\, \pi_1(\mathsf{pred}' n) \\
&= \lambda n.\, \pi_1((\lambda n.\, n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle))n) \\
&=_\beta \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle)) \\
&= \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \lambda t, f.\, t\rangle) \\
&\qquad\qquad (\lambda p.\, \langle (\lambda c, a, b.\, cab)\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \lambda t, f.\, f\rangle))
\end{aligned}
$$

Predecessor with Church Numerals

$$
\begin{aligned}
\mathsf{pred} &= \lambda n.\, \pi_1(\mathsf{pred}' n) \\
&= \lambda n.\, \pi_1((\lambda n.\, n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle))n) \\
&=_\beta \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle)) \\
&= \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \lambda t, f.\, t\rangle) \\
&\qquad\qquad (\lambda p.\, \langle (\lambda c, a, b.\, cab)\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \lambda t, f.\, f\rangle)) \\
&=_\beta \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \lambda t, f.\, t\rangle) \\
&\qquad\qquad (\lambda p.\, \langle (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \lambda t, f.\, f\rangle))
\end{aligned}
$$

Predecessor with Church Numerals

$$
\begin{aligned}
\mathsf{pred} &= \lambda n.\, \pi_1(\mathsf{pred}' n) \\
&= \lambda n.\, \pi_1((\lambda n.\, n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle))n) \\
&=_\beta \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle)) \\
&= \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \lambda t, f.\, t\rangle) \\
&\qquad\qquad (\lambda p.\, \langle (\lambda c, a, b.\, cab)\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \lambda t, f.\, f\rangle)) \\
&=_\beta \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \lambda t, f.\, t\rangle) \\
&\qquad\qquad (\lambda p.\, \langle (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \lambda t, f.\, f\rangle)) \\
&= \lambda n.\, (\lambda p.\, p(\lambda a, b.\, a))(n\,(\lambda q.\, q\,\mathsf{0}\,(\lambda t, f.\, t)) \\
&\qquad\qquad (\lambda p.\, \lambda q.\, q\,(((\lambda p.\, p(\lambda a, b.\, b))p)\ \mathsf{0}\ (\mathsf{S}((\lambda p.\, p(\lambda a, b.\, a))p))) \\
&\qquad\qquad\qquad (\lambda t, f.\, f)))
\end{aligned}
$$

Predecessor with Church Numerals

$$
\begin{aligned}
\mathsf{pred} &= \lambda n.\, \pi_1(\mathsf{pred}' n) \\
&= \lambda n.\, \pi_1((\lambda n.\, n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle))n) \\
&=_\beta \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle)) \\
&= \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \lambda t, f.\, t\rangle) \\
&\qquad\qquad (\lambda p.\, \langle (\lambda c, a, b.\, cab)\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \lambda t, f.\, f\rangle)) \\
&=_\beta \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \lambda t, f.\, t\rangle) \\
&\qquad\qquad (\lambda p.\, \langle (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \lambda t, f.\, f\rangle)) \\
&= \lambda n.\, (\lambda p.\, p(\lambda a, b.\, a))(n\,(\lambda q.\, q\,\mathsf{0}\,(\lambda t, f.\, t)) \\
&\qquad\qquad (\lambda p.\, \lambda q.\, q\,(((\lambda p.\, p(\lambda a, b.\, b))p)\ \mathsf{0}\ (\mathsf{S}((\lambda p.\, p(\lambda a, b.\, a))p))) \\
&\qquad\qquad\qquad (\lambda t, f.\, f))) \\
&=_\beta \lambda n.\, n\,(\lambda q.\, q\,\mathsf{0}\,(\lambda t, f.\, t)) \\
&\qquad\qquad (\lambda p, q.\, q(p(\lambda a, b.\, b)\mathsf{0}(\mathsf{S}(p(\lambda a, b.\, a)))))\,(\lambda a, b.\, a)
\end{aligned}
$$

Predecessor with Church Numerals

$$
\begin{aligned}
\mathsf{pred} &= \lambda n.\, \pi_1(\mathsf{pred}' n) \\
&= \lambda n.\, \pi_1((\lambda n.\, n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle))n) \\
&=_\beta \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \mathbf{true}\rangle)\,(\lambda p.\, \langle \mathbf{if}\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \mathbf{false}\rangle)) \\
&= \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \lambda t, f.\, t\rangle) \\
&\qquad (\lambda p.\, \langle (\lambda c, a, b.\, cab)\ (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \lambda t, f.\, f\rangle)) \\
&=_\beta \lambda n.\, \pi_1(n\,(\langle \mathsf{0}, \lambda t, f.\, t\rangle) \\
&\qquad (\lambda p.\, \langle (\pi_2 p)\ \mathsf{0}\ (\mathsf{S}(\pi_1 p)), \lambda t, f.\, f\rangle)) \\
&= \lambda n.\, (\lambda p.\, p(\lambda a, b.\, a))(n\,(\lambda q.\, q\,\mathsf{0}\,(\lambda t, f.\, t)) \\
&\qquad (\lambda p.\, \lambda q.\, q\,(((\lambda p.\, p(\lambda a, b.\, b))p)\ \mathsf{0}\ (\mathsf{S}((\lambda p.\, p(\lambda a, b.\, a))p))) \\
&\qquad\qquad (\lambda t, f.\, f))) \\
&=_\beta \lambda n.\, n\,(\lambda q.\, q\,\mathsf{0}\,(\lambda t, f.\, t)) \\
&\qquad (\lambda p, q.\, q(p(\lambda a, b.\, b)\mathsf{0}(\mathsf{S}(p(\lambda a, b.\, a)))))\,(\lambda a, b.\, a) \\
&=_\beta \lambda n.\, n\,(\lambda q.\, q\,(\lambda o, s.\, o)\,(\lambda t, f.\, t)) \\
&\qquad (\lambda p, q.\, q(p\,(\lambda a, b.\, b)\,(\lambda o, s.\, o)\,(\lambda o, s.\, s(p(\lambda a, b.\, a)os)))) \\
&\qquad (\lambda a, b.\, a)
\end{aligned}
$$

On Pairs

pred included subterms $p(\lambda a, b.\, a)$ and $p(\lambda a, b.\, b)$ — the first and second projections of p. When p is a pair, the term

$$q(p(\lambda a, b.\, a))(p(\lambda a, b.\, b))$$

is equivalent to

$$pq$$

However, if p is *not* a pair, then this equivalence does not hold.

Primitive Recursion vs. Recursion

Primitive recursion

- guarantees that functions will terminate (when called on a numeral)
- cannot define *all* computable functions e.g.

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

General recursion

- can define non-terminating functions
- can define *all* computable functions

Section 2

Simply-Typed Lambda Calculus

Types

Terms in the lambda calculus can represent different kinds of things, e.g.:

- a natural number
- a boolean
- a function from natural numbers to booleans
- a pair of a boolean and a natural number
- a pair of a natural number and a boolean

While we might expect a function to take a number as an argument, we could equally well call it with a pair instead. If we do so, it is difficult to predict what will happen. Essentially, we want to avoid this kind of situation.

Simply-typed Lambda Calculus

The *simply-typed lambda calculus* introduces *types* for terms to the lambda calculus. We let $\tau, \alpha, \beta, \gamma$ range over an infinite set of *type variables*. Types are:

$$\sigma, \rho ::= \tau \mid \sigma \to \rho$$

By convention $\alpha \to \beta \to \gamma$ means $\alpha \to (\beta \to \gamma)$.

Simply-typed Lambda Calculus

The *simply-typed lambda calculus* introduces *types* for terms to the lambda calculus. We let $\tau, \alpha, \beta, \gamma$ range over an infinite set of *type variables*. Types are:

$$\sigma, \rho ::= \tau \mid \sigma \to \rho$$

By convention $\alpha \to \beta \to \gamma$ means $\alpha \to (\beta \to \gamma)$.
Terms are as before, except that we attach types to lambda abstractions to indicate the type of the argument:

$$M, N ::= x \mid MN \mid \lambda x : \sigma . M$$

Typing Terms

We define a judgement that determines when a term M has a type σ. To type a term, we need a context Γ that tells us the type of the free variables of M. A context is a set of pairs of variables and types $x : \sigma$, where each x occurs at most once.
The typing judgement is defined by the rules:

$$\textsc{Start}\ \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \to \textsc{elimination}\ \frac{\Gamma \vdash M : \sigma \to \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \rho} \qquad \to \textsc{introduction}\ \frac{\Gamma, x : \sigma \vdash M : \rho}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \rho}$$

Examples

$$\rightarrow\mathrm{I}\ \frac{}{\vdash \lambda x : \alpha.\, x : \alpha \rightarrow \alpha}$$

Examples

$$\to\mathrm{I}\ \dfrac{\mathrm{St}\ \dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x : \alpha.\, x : \alpha \to \alpha}$$

$$\to\mathrm{I}\ \dfrac{\mathrm{St}\ \dfrac{}{x : \alpha \to \beta \vdash x : \alpha \to \beta}}{\vdash \lambda x : \alpha \to \beta.\, x : (\alpha \to \beta) \to (\alpha \to \beta)}$$

Examples

$$\to\mathrm{I}\ \frac{\mathrm{St}\ \dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x : \alpha.\, x : \alpha \to \alpha}$$

$$\to\mathrm{I}\ \frac{\mathrm{St}\ \dfrac{}{x : \alpha \to \beta \vdash x : \alpha \to \beta}}{\vdash \lambda x : \alpha \to \beta.\, x : (\alpha \to \beta) \to (\alpha \to \beta)}$$

$$\to\mathrm{I}\ \frac{\mathrm{St}\ \dfrac{}{x : (\alpha \to \beta) \to \gamma \vdash x : (\alpha \to \beta) \to \gamma}}{\vdash \lambda x : (\alpha \to \beta) \to \gamma.\, x : ((\alpha \to \beta) \to \gamma) \to (\alpha \to \beta) \to \gamma}$$

Examples

$$\to\mathrm{I}\ \frac{}{\vdash \lambda x : \alpha, y : \beta.\, x : \alpha \to \beta \to \alpha}$$

Examples

$$\dfrac{\dfrac{}{x : \alpha \vdash \lambda y : \beta . x : \beta \to \alpha}\,{\to}\mathrm{I}}{\vdash \lambda x : \alpha, y : \beta . x : \alpha \to \beta \to \alpha}\,{\to}\mathrm{I}$$

Examples

$$\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \beta \vdash x : \alpha}\,\textsc{St}}{x : \alpha \vdash \lambda y : \beta.\, x : \beta \to \alpha}\,{\to}\mathrm{I}}{\vdash \lambda x : \alpha, y : \beta.\, x : \alpha \to \beta \to \alpha}\,{\to}\mathrm{I}$$

Examples

$$\rightarrow\mathrm{I}\ \frac{}{\vdash \lambda x \qquad\qquad\qquad , y \qquad\qquad , z \qquad . xz(yz)}$$

Examples

$$\rightarrow\mathrm{I}\ \frac{}{\vdash \lambda x \qquad\qquad , y \qquad\qquad , z \quad . xz(yz) : \qquad\qquad \rightarrow \qquad\qquad \rightarrow \quad \rightarrow}$$

Examples

$$\to\mathrm{I}\ \frac{}{\vdash \lambda x : \alpha \to \beta \to \gamma, y \qquad\qquad , z \qquad . xz(yz) : (\alpha \to \beta \to \gamma) \to \qquad\qquad \to \quad \to \gamma}$$

Examples

$$\to\mathrm{I}\ \frac{}{\vdash \lambda x : \alpha \to \beta \to \gamma, y \qquad\qquad , z : \alpha.\, xz(yz) : (\alpha \to \beta \to \gamma) \to \qquad\qquad \to \alpha \to \gamma}$$

Examples

$$\to\mathrm{I}\ \frac{}{\vdash \lambda x : \alpha \to \beta \to \gamma, y : \alpha \to \beta, z : \alpha.\, xz(yz) : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma}$$

Examples

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : \alpha \to (\beta \to \gamma)}\,\text{St} \quad \dfrac{}{\Gamma \vdash z : \alpha}\,\text{St}}{\Gamma \vdash xz : \beta \to \gamma}\,{\to}\text{E} \quad \dfrac{\dfrac{}{\Gamma \vdash y : \alpha \to \beta}\,\text{St} \quad \dfrac{}{\Gamma \vdash z : \alpha}\,\text{St}}{\Gamma \vdash yz : \beta}\,{\to}\text{E}}{x : \alpha \to \beta \to \gamma, y : \alpha \to \beta, z : \alpha \vdash xz(yz) : \gamma}\,{\to}\text{E}}{x : \alpha \to \beta \to \gamma, y : \alpha \to \beta \vdash \lambda z : \alpha.\, xz(yz) : \alpha \to \gamma}\,{\to}\text{I}}{x : \alpha \to \beta \to \gamma \vdash \lambda y : \alpha \to \beta, z : \alpha.\, xz(yz) : (\alpha \to \beta) \to \alpha \to \gamma}\,{\to}\text{I}}{\vdash \lambda x : \alpha \to \beta \to \gamma, y : \alpha \to \beta, z : \alpha.\, xz(yz) : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma}\,{\to}\text{I}$$

$$\Gamma = [x : \alpha \to \beta \to \gamma, y : \alpha \to \beta, z : \alpha]$$

Typing Terms

Note that typing is *entirely* directed by the syntax of terms:

- x — use $\textsc{Start}$
- $\lambda x : \sigma.\, M$ — use $\to \textsc{Introduction}$
- MN — use $\to \textsc{Elimination}$

This means that any term has at most one type.
Moreover, it is decidable whether a term has a type.

Untypeable Terms

Not every term is typeable.
Consider xx, and suppose that $\Gamma \vdash xx : \sigma$.
The derivation must have the following form:

$$\rightarrow\mathrm{E}\ \dfrac{\mathrm{St}\ \dfrac{(x : \rho \rightarrow \sigma) \in \Gamma}{\Gamma \vdash x : \rho \rightarrow \sigma} \qquad \mathrm{St}\ \dfrac{(x : \rho) \in \Gamma}{\Gamma \vdash x : \rho}}{\Gamma \vdash xx : \sigma}$$

Since a variable can have at most one type in a context, we must have $\rho = (\rho \rightarrow \sigma)$, but there is no such type!

Untypeable Terms

Not every term is typeable.
Consider xx, and suppose that $\Gamma \vdash xx : \sigma$.
The derivation must have the following form:

$$\dfrac{\dfrac{(x : \rho \to \sigma) \in \Gamma}{\Gamma \vdash x : \rho \to \sigma}\,\textsc{St} \qquad \dfrac{(x : \rho) \in \Gamma}{\Gamma \vdash x : \rho}\,\textsc{St}}{\Gamma \vdash xx : \sigma}\,{\to}\textsc{E}$$

Since a variable can have at most one type in a context, we must have $\rho = (\rho \to \sigma)$, but there is no such type!
Note that this means $\mathbf{Y}$ cannot be typed in STLC

$$\mathbf{Y} = \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$$

Properties of Typing

Lemma (Weakening)**.** If $\Gamma \vdash M : \sigma$ and Γ, Γ' is a well-defined typing context, then $\Gamma, \Gamma' \vdash M : \sigma$.

Properties of Typing

Lemma (Weakening)**.** If $\Gamma \vdash M : \sigma$ and Γ, Γ' is a well-defined typing context, then $\Gamma, \Gamma' \vdash M : \sigma$.

Lemma. If $\Gamma, \Gamma' \vdash M : \sigma$ and none of the variables in Γ' occur free in M, then $\Gamma \vdash M : \sigma$.

Substitution and Preservation

Lemma. Suppose that $\Gamma, x : \sigma \vdash M : \rho$ and $\Gamma \vdash N : \sigma$. Then $\Gamma \vdash M[N/x] : \rho$.

Theorem (Preservation)**.** Suppose that $\Gamma \vdash M : \sigma$ and $M \to_\beta N$. Then $\Gamma \vdash N : \sigma$.

Strong Normalisation

An important property of well-typed terms is that they are strongly normalising: reduction is guaranteed to terminate.

Theorem (Strong Normalisation)**.** If $\Gamma \vdash M : \sigma$ then M is strongly normalising with respect to β-reduction.

Strong Normalisation

An important property of well-typed terms is that they are strongly normalising: reduction is guaranteed to terminate.

Theorem (Strong Normalisation)**.** If $\Gamma \vdash M : \sigma$ then M is strongly normalising with respect to β-reduction.

N.B. This means the simply-typed lambda calculus is *not* Turing complete, since we cannot represent non-terminating Turing machines with simply-typed lambda terms.

Allowing Recursion

We can permit recursion (at the expense of strong normalisation) by adding a construct such as

$$fix_\rho : (\rho \to \rho) \to \rho$$

to the language.

Section 3

Logic

Logic

You may recall from formal logic the validity judgement $\Gamma \vdash \phi$: from assumptions Γ, ϕ is provable.
Proof systems for (propositional) logic are built on the proof rule *modus ponens*:

$$\frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

Logic

You may recall from formal logic the validity judgement $\Gamma \vdash \phi$: from assumptions Γ, ϕ is provable.
Proof systems for (propositional) logic are built on the proof rule *modus ponens*:

$$\frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

This should look very familiar — it is just the same as $\to$ ELIMINATION, but without λ-terms:

$$\frac{\Gamma \vdash M : \sigma \to \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \rho} \to \text{ ELIMINATION}$$

Hilbert-Style Propositional Logic

A logic typically takes this rule together with a system of axioms, such as:

$$\frac{}{\vdash A \to A} \qquad \frac{}{\vdash A \to (B \to A)}$$

$$\frac{}{\vdash (A \to (B \to C)) \to ((A \to B) \to (A \to C))}$$

$$\frac{}{\vdash ((A \to B) \to A) \to A}$$

Together, these axiomatise (implicational) propositional logic. Without the third, they give implicational intuitionistic logic.

λ-terms as Proof Terms

We can realise the intuitionistic axioms with λ-terms:

$$\vdash \lambda x : A.\, x : A \to A \qquad\qquad \vdash \lambda x : A, y : B.\, x : A \to (B \to A)$$

$$\vdash \lambda x : (A \to (B \to C)), y : (A \to B), z : A.\, xz(yz)$$
$$: (A \to (B \to C)) \to ((A \to B) \to (A \to C))$$

(There is no term of type $((A \to B) \to A) \to A$.)

Sequent Calculus

Hilbert systems do not use the context; an alternative formulation that does is sequent calculus.
Instead of a system of axioms, sequent calculus uses introduction and elimination rules for each connective, such as:

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \qquad \frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi} \qquad \frac{}{\Gamma, \phi \vdash \phi}$$

These are just the typing rules of STLC!

Curry-Howard Isomorphism

If we interpret a logical proposition P as a type, then a term M of type P witnesses the validity of P. That is, from M we can construct a proof of P.

In fact, we may identify propositions with types and proofs (of propositions) with programs (of types). This is called the *Curry-Howard isomorphism* (or correspondence).

Curry-Howard Isomorphism

If we interpret a logical proposition P as a type, then a term M of type P witnesses the validity of P. That is, from M we can construct a proof of P.

In fact, we may identify propositions with types and proofs (of propositions) with programs (of types). This is called the *Curry-Howard isomorphism* (or correspondence).

In fact, this correspondence goes beyond propositional logic (as we have seen) and forms the basis for proof assistants such as Coq.

Curry-Howard and Recursion

Note that with fix_σ we can define terms of arbitrary type:

$$\vdash fix_\sigma\ id_\sigma : \sigma$$

Under Curry-Howard, this corresponds to every proposition being provable, i.e. an inconsistent logic. Therefore, for a type system to support a logic, it has to avoid general recursion.

Section 4

Curry and Church

Curry and Church-style Type Systems

The approach we have taken to simply-typed lambda calculus is to associate a type with every abstraction — Church's approach.
Curry's approach does not associate types directly with abstractions — terms are just the terms of untyped λ-calculus.

Curry and Church-style Type Systems

The approach we have taken to simply-typed lambda calculus is to associate a type with every abstraction — Church's approach.
Curry's approach does not associate types directly with abstractions — terms are just the terms of untyped λ-calculus.

$\to$ INTRODUCTION (CURRY)

$$\frac{\Gamma, x : \sigma \vdash M : \rho}{\Gamma \vdash (\lambda x.\, M) : \sigma \to \rho}$$

Curry and Church-style Type Systems

The approach we have taken to simply-typed lambda calculus is to associate a type with every abstraction — Church's approach. Curry's approach does not associate types directly with abstractions — terms are just the terms of untyped λ-calculus.

$\to$ introduction (Curry)

$$\frac{\Gamma, x : \sigma \vdash M : \rho}{\Gamma \vdash (\lambda x.\, M) : \sigma \to \rho}$$

$\to$ introduction (Church)

$$\frac{\Gamma, x : \sigma \vdash M : \rho}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \rho}$$

Curry-style STLC

Remember that in Church's STLC every typeable term has a unique type? That's not the case for Curry's system.

$$\to\mathrm{I}\ \dfrac{\mathrm{St}\ \dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.\, x : \alpha \to \alpha}$$

Curry-style STLC

Remember that in Church's STLC every typeable term has a unique type? That's not the case for Curry's system.

$$\to\mathrm{I}\ \frac{\mathrm{St}\ \dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.\, x : \alpha \to \alpha} \qquad \to\mathrm{I}\ \frac{\mathrm{St}\ \dfrac{}{x : \beta \vdash x : \beta}}{\vdash \lambda x.\, x : \beta \to \beta}$$

Curry-style STLC

Remember that in Church's STLC every typeable term has a unique type? That's not the case for Curry's system.

$$\to\mathrm{I}\ \frac{\mathrm{St}\ \dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.\, x : \alpha \to \alpha} \qquad \to\mathrm{I}\ \frac{\mathrm{St}\ \dfrac{}{x : \beta \vdash x : \beta}}{\vdash \lambda x.\, x : \beta \to \beta} \qquad \to\mathrm{I}\ \frac{\mathrm{St}\ \dfrac{}{x : \alpha \to \beta \vdash x : \alpha \to \beta}}{\vdash \lambda x.\, x : (\alpha \to \beta) \to (\alpha \to \beta)}$$

Curry-style STLC

Remember that in Church's STLC every typeable term has a unique type? That's not the case for Curry's system.

$$\to\mathrm{I}\,\frac{\mathrm{St}\,\dfrac{}{x:\alpha\vdash x:\alpha}}{\vdash \lambda x.\,x:\alpha\to\alpha}\qquad \to\mathrm{I}\,\frac{\mathrm{St}\,\dfrac{}{x:\beta\vdash x:\beta}}{\vdash \lambda x.\,x:\beta\to\beta}\qquad \to\mathrm{I}\,\frac{\mathrm{St}\,\dfrac{}{x:\alpha\to\beta\vdash x:\alpha\to\beta}}{\vdash \lambda x.\,x:(\alpha\to\beta)\to(\alpha\to\beta)}$$

However, every typeable term has a *principle type*, from which all other types can be derived by substitution. (Principle types can be determined by *type inference*.)

Curry and Church

If a term is typeable in the Church style, then we can erase the types and it will be typeable with the same type in the Curry style.

Curry and Church

If a term is typeable in the Church style, then we can erase the types and it will be typeable with the same type in the Curry style.

If a term is typeable in the Curry style, then we can insert types (according to how $\rightarrow$I is used in the derivation) and get a term that is typeable with the same type in the Church style.

Curry and Church

If a term is typeable in the Church style, then we can erase the types and it will be typeable with the same type in the Curry style.

If a term is typeable in the Curry style, then we can insert types (according to how $\rightarrow$I is used in the derivation) and get a term that is typeable with the same type in the Church style.

We will look at more complex type systems in the Church style. Some can be reformulated in the Curry style, others cannot.

Section 5

System F

Encoding Booleans

It is possible to extend STLC with additional types and terms for various datatypes, e.g. numbers, booleans, pairs (Ex. 11). But what about the Church encodings?

Encoding Booleans

It is possible to extend STLC with additional types and terms for various datatypes, e.g. numbers, booleans, pairs (Ex. 11). But what about the Church encodings?

$$\mathsf{true} = \lambda x, y.\, x \qquad\qquad \mathsf{false} = \lambda x, y.\, y$$

Encoding Booleans

It is possible to extend STLC with additional types and terms for various datatypes, e.g. numbers, booleans, pairs (Ex. 11). But what about the Church encodings?

$$\mathsf{true} = \lambda x, y.\, x \qquad\qquad \mathsf{false} = \lambda x, y.\, y$$

We can add type annotations to type these as $\alpha \to \alpha \to \alpha$:

$$\vdash \lambda x : \alpha, y : \alpha.\, x : \alpha \to \alpha \to \alpha \qquad\qquad \vdash \lambda x : \alpha, y : \alpha.\, y : \alpha \to \alpha \to \alpha$$

But is $\mathbf{bool} = \alpha \to \alpha \to \alpha$ a suitable type to represent booleans?

Conditional

We would like to have a term 'if' such that

$$\text{if true } M\ N \to_\beta^* M \qquad\qquad \text{if false } M\ N \to_\beta^* N$$

If M and N have type τ then if should have type $\mathbf{bool} \to \tau \to \tau \to \tau$.

Conditional

We would like to have a term 'if' such that

$$\text{if true } M\ N \to_\beta^* M \qquad\qquad \text{if false } M\ N \to_\beta^* N$$

If M and N have type τ then if should have type $\mathbf{bool} \to \tau \to \tau \to \tau$.

$$\vdash \lambda b : \mathbf{bool}, x : \tau, y : \tau.\, bxy : \mathbf{bool} \to \tau \to \tau \to \tau$$

Conditional

We would like to have a term 'if' such that

$$\text{if true } M\ N \to_\beta^* M \qquad\qquad \text{if false } M\ N \to_\beta^* N$$

If M and N have type τ then if should have type $\mathbf{bool} \to \tau \to \tau \to \tau$.

$$\dfrac{\dfrac{\dfrac{}{b : \alpha \to \alpha \to \alpha, x : \tau, y : \tau \vdash bxy : \tau}}{\vdots}}{\vdash \lambda b : \mathbf{bool}, x : \tau, y : \tau.\, bxy : \mathbf{bool} \to \tau \to \tau \to \tau}$$

Conditional

We would like to have a term 'if' such that

$$\text{if true } M\ N \to^*_\beta M \qquad\qquad \text{if false } M\ N \to^*_\beta N$$

If M and N have type τ then if should have type $\mathbf{bool} \to \tau \to \tau \to \tau$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash b : \tau \to \tau \to \tau \qquad \dfrac{}{\Gamma \vdash x : \tau}}{\Gamma \vdash bx : \tau \to \tau} \qquad \dfrac{}{\Gamma \vdash y : \tau}}{b : \alpha \to \alpha \to \alpha, x : \tau, y : \tau \vdash bxy : \tau}}{\vdots}}{\vdash \lambda b : \mathbf{bool}, x : \tau, y : \tau.\, bxy : \mathbf{bool} \to \tau \to \tau \to \tau}$$

$$\Gamma = [b : \alpha \to \alpha \to \alpha, x : \tau, y : \tau]$$

The Problem

The issue is that the type α is fixed (albeit arbitrarily). The type system will not let us use a term of type $\alpha \to \alpha \to \alpha$ as if it were of type $\tau \to \tau \to \tau$, or any other type.

The Problem

The issue is that the type α is fixed (albeit arbitrarily). The type system will not let us use a term of type $\alpha \to \alpha \to \alpha$ as if it were of type $\tau \to \tau \to \tau$, or any other type.

This kind of restriction also prevents us from e.g. pred (Exercise 10).

The Solution: Parametric Polymorphism

Idea: parametrise terms by types.

$$\mathsf{true} = \Lambda\alpha.\, \lambda x : \alpha, y : \alpha.\, x$$

The Solution: Parametric Polymorphism

Idea: parametrise terms by types.

$$\mathsf{true} = \Lambda\alpha.\, \lambda x : \alpha, y : \alpha.\, x$$

This gives us a polymorphic true:

$$(\Lambda\alpha.\, \lambda x : \alpha, y : \alpha.\, x)\sigma \quad \rightarrow_\beta \quad \lambda x : \sigma, y : \sigma.\, x$$

The Solution: Parametric Polymorphism

Idea: parametrise terms by types.

$$\mathsf{true} = \Lambda\alpha.\, \lambda x : \alpha, y : \alpha.\, x$$

This gives us a polymorphic true:

$$(\Lambda\alpha.\, \lambda x : \alpha, y : \alpha.\, x)\sigma \quad \to_\beta \quad \lambda x : \sigma, y : \sigma.\, x$$

We reflect the abstraction over the type α by quantifying over it in the type:

$$\mathbf{bool} = \forall\alpha.\, \alpha \to \alpha \to \alpha$$

System F — Second-order Polymorphic Lambda Calculus

Terms are now extended with type application and type abstraction:

$$M, N ::= x \mid MN \mid \lambda x : \sigma. M \mid M\sigma \mid \Lambda\alpha. M$$

System F — Second-order Polymorphic Lambda Calculus

Terms are now extended with type application and type abstraction:

$$M, N ::= x \mid MN \mid \lambda x : \sigma. M \mid M\sigma \mid \Lambda\alpha. M$$

Types are extended with quantification:

$$\sigma, \rho ::= \alpha \mid \sigma \to \rho \mid \forall\alpha. \sigma$$

System F — Second-order Polymorphic Lambda Calculus

Terms are now extended with type application and type abstraction:

$$M, N ::= x \mid MN \mid \lambda x : \sigma. M \mid M\sigma \mid \Lambda\alpha. M$$

Types are extended with quantification:

$$\sigma, \rho ::= \alpha \mid \sigma \to \rho \mid \forall\alpha. \sigma$$

Beta reduction also applies to type application:

$$\frac{}{(\Lambda\alpha. M)\sigma \to_\beta M[\sigma/\alpha]}$$

Binders and Substitution

Just like λ binds a term variable (e.g. x) in a term, Λ binds a type variable (e.g. α) in a term and $\forall$ binds a type variable in a type.
The rules of α-conversion extend to these binders, allowing us to rename bound type variables.

Binders and Substitution

Just like λ binds a term variable (e.g. x) in a term, Λ binds a type variable (e.g. α) in a term and $\forall$ binds a type variable in a type.
The rules of α-conversion extend to these binders, allowing us to rename bound type variables.

Substitution also extends to types, allowing us to substitute a type for a free type variable. As before, substitution respects α-equivalence and does not substitute bound variables or cause variable capture.

Typing Rules

$$\textsc{Start}\ \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\to \textsc{elimination}\ \frac{\Gamma \vdash M : \sigma \to \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \rho}$$

$$\to \textsc{introduction}\ \frac{\Gamma, x : \sigma \vdash M : \rho}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \rho}$$

Typing Rules

$$\textsc{Start}\quad \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\to \textsc{ elimination}\quad \frac{\Gamma \vdash M : \sigma \to \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \rho}$$

$$\to \textsc{ introduction}\quad \frac{\Gamma, x : \sigma \vdash M : \rho}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \rho}$$

$$\forall \textsc{ elimination}\quad \frac{\Gamma \vdash M : \forall \alpha.\, \sigma}{\Gamma \vdash M\rho : \sigma[\rho/\alpha]}$$

$$\forall \textsc{ introduction}\quad \frac{\Gamma \vdash M : \sigma \qquad \alpha \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash \Lambda\alpha.\, M : \forall \alpha.\, \sigma}$$

Section 6

Data types

Back to Booleans

Now we can give a type of booleans:

$$\mathbf{bool} = \forall\alpha.\, \alpha \to \alpha \to \alpha$$

Back to Booleans

Now we can give a type of booleans:

$$\mathbf{bool} = \forall\alpha.\, \alpha \to \alpha \to \alpha$$

$$\text{true} : \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \alpha \vdash x : \alpha}\,\text{ST}}{x : \alpha \vdash \lambda y : \alpha.\, x : \alpha \to \alpha}\,{\to}\text{I}}{\vdash \lambda x : \alpha, y : \alpha.\, x : \alpha \to \alpha \to \alpha}\,{\to}\text{I}}{\vdash \Lambda\alpha.\, \lambda x : \alpha, y : \alpha.\, x : \forall\alpha.\, \alpha \to \alpha \to \alpha}\,\forall\text{I}\quad \alpha \notin \emptyset$$

$$\text{false} : \vdash \Lambda\alpha.\, \lambda x : \alpha, y : \alpha.\, y : \forall\alpha.\, \alpha \to \alpha \to \alpha$$

Typing Conditional

$$\mathsf{if} = \Lambda\beta.\, \lambda b : \mathbf{bool}, x : \beta, y : \beta.\, b\beta xy$$

Typing Conditional

$$\mathsf{if} = \Lambda\beta.\, \lambda b : \mathbf{bool}, x : \beta, y : \beta.\, b\beta xy$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash b : \forall\alpha.\, \alpha \to \alpha \to \alpha}\,\textsc{St}}{\Gamma \vdash b\beta : \beta \to \beta \to \beta}\,\forall\mathrm{E} \quad \dfrac{}{\Gamma \vdash x : \beta}\,\textsc{St}}{\Gamma \vdash b\beta x : \beta \to \beta}\,{\to}\mathrm{E} \quad \dfrac{}{\Gamma \vdash y : \beta}\,\textsc{St}}{\Gamma \vdash b\beta xy : \beta}\,{\to}\mathrm{E}}{\vdots}}{\vdash \lambda b : \mathbf{bool}, x : \beta, y : \beta.\, b\beta xy : \beta}\,{\to}\mathrm{I}}{\vdash \Lambda\beta.\, \lambda b : \mathbf{bool}, x : \beta, y : \beta.\, b\beta xy : \forall\beta.\, \mathbf{bool} \to \beta \to \beta \to \beta}\,\forall\mathrm{I}$$

$$\Gamma = [b : (\forall\alpha.\, \alpha \to \alpha \to \alpha), x : \beta, y : \beta]$$

Church Numerals

$$\mathbf{nat} = \forall \nu.\, \nu \to (\nu \to \nu) \to \nu$$

$$0 = \Lambda \nu.\, \lambda o : \nu, s : \nu \to \nu.\, o$$

$$1 = \Lambda \nu.\, \lambda o : \nu, s : \nu \to \nu.\, so$$

$$2 = \Lambda \nu.\, \lambda o : \nu, s : \nu \to \nu.\, s(so)$$

$$3 = \Lambda \nu.\, \lambda o : \nu, s : \nu \to \nu.\, s(s(so))$$

Church Numerals

$$\mathbf{nat} = \forall \nu.\, \nu \to (\nu \to \nu) \to \nu$$

$$0 = \Lambda \nu.\, \lambda o : \nu, s : \nu \to \nu.\, o$$

$$1 = \Lambda \nu.\, \lambda o : \nu, s : \nu \to \nu.\, so$$

$$2 = \Lambda \nu.\, \lambda o : \nu, s : \nu \to \nu.\, s(so)$$

$$3 = \Lambda \nu.\, \lambda o : \nu, s : \nu \to \nu.\, s(s(so))$$

In System F, we can define primitive recursive functions in a well-typed fashion.

Church Encoding

Idea: for an inductively defined datatype:

- abstract over representation type;
- abstract over constructors into this type.

Church Encoding

Idea: for an inductively defined datatype:

- abstract over representation type;
- abstract over constructors into this type.

Example: $\mathbf{list}\ \sigma$

$$\mathsf{nil} : \mathbf{list}\ \sigma$$
$$\mathsf{cons} : \sigma \to \mathbf{list}\ \sigma \to \mathbf{list}\ \sigma$$

Church Encoding

Idea: for an inductively defined datatype:

- abstract over representation type;
- abstract over constructors into this type.

Example: $\mathbf{list}\ \sigma$

$$\begin{aligned} \mathsf{nil} &: \mathbf{list}\ \sigma \\ \mathsf{cons} &: \sigma \to \mathbf{list}\ \sigma \to \mathbf{list}\ \sigma \end{aligned}$$

$$\mathbf{list}\ \sigma = \forall \alpha.\, \alpha \to (\sigma \to \alpha \to \alpha) \to \alpha$$

Section 7

Logic

Logic

We can encode logical connectives using System F.
For example, $\bot$ can be represented as $\forall\alpha.\,\alpha$. $\bot$ satisfies the (elimination) axiom

$$\bot \to \phi$$

Logic

We can encode logical connectives using System F.
For example, $\bot$ can be represented as $\forall\alpha.\,\alpha$. $\bot$ satisfies the (elimination) axiom

$$\bot \to \phi$$

$$\to\mathrm{I}\ \frac{\forall\mathrm{E}\ \dfrac{\mathrm{St}\ \dfrac{}{f : (\forall\alpha.\,\alpha) \vdash f : (\forall\alpha.\,\alpha)}}{f : (\forall\alpha.\,\alpha) \vdash f\psi : \psi}}{\vdash \lambda f : (\forall\alpha.\,\alpha).\, f\psi : (\forall\alpha.\,\alpha) \to \psi}$$

Logic

We can encode logical connectives using System F.
For example, $\bot$ can be represented as $\forall\alpha.\,\alpha$. $\bot$ satisfies the (elimination) axiom

$$\bot \to \phi$$

$$\to\mathrm{I}\ \frac{\forall\mathrm{E}\ \dfrac{\mathrm{St}\ \dfrac{}{f : (\forall\alpha.\,\alpha) \vdash f : (\forall\alpha.\,\alpha)}}{f : (\forall\alpha.\,\alpha) \vdash f\psi : \psi}}{\vdash \lambda f : (\forall\alpha.\,\alpha).\, f\psi : (\forall\alpha.\,\alpha) \to \psi}$$

We can think of $\bot$ as an inductive type with no constructors.

Disjunction

Disjunction $\phi \vee \psi$ has two introduction axioms:

$$\phi \to (\phi \vee \psi) \qquad\qquad \psi \to (\phi \vee \psi)$$

It also has one elimination axiom:

$$(\phi \to \theta) \to (\psi \to \theta) \to (\phi \vee \psi) \to \theta$$

Disjunction

Disjunction $\phi \vee \psi$ has two introduction axioms:

$$\phi \to (\phi \vee \psi) \qquad\qquad \psi \to (\phi \vee \psi)$$

It also has one elimination axiom:

$$(\phi \to \theta) \to (\psi \to \theta) \to (\phi \vee \psi) \to \theta$$

We can think of the introduction axioms as constructors. Thus we can define

$$\phi \vee \psi = \forall \theta. (\phi \to \theta) \to (\psi \to \theta) \to \theta$$

Disjunction Axioms

$$\begin{aligned}\vdash \lambda f : (\phi \to \theta), g : (\psi \to \theta), h : (\phi \vee \psi).\, h\theta f g \\ : (\phi \to \theta) \to (\psi \to \theta) \to (\phi \vee \psi) \to \theta\end{aligned}$$

$$\vdash \lambda a : \phi.\, \Lambda\theta.\, \lambda f : (\phi \to \theta), g : (\psi \to \theta).\, fa : \phi \to (\phi \vee \psi)$$

$$\vdash \lambda b : \psi.\, \Lambda\theta.\, \lambda f : (\phi \to \theta), g : (\psi \to \theta).\, gb : \psi \to (\phi \vee \psi)$$

Recall that:

$$\phi \vee \psi = \forall\theta.\, (\phi \to \theta) \to (\psi \to \theta) \to \theta$$

Hence the structure $\lambda\_\Lambda\_\lambda\_, \_ : \phi \to (\phi \vee \psi)$.