Section 1

System F_ω

Abstraction within types

We have defined types that are parametric in other types:

$$\phi \vee \psi = \forall\theta.\,(\phi \to \theta) \to (\psi \to \theta) \to \theta$$
$$\sigma \times \rho = \forall\alpha.\,(\sigma \to \rho \to \alpha) \to \alpha$$
$$\mathbf{list}\ \sigma = \forall\alpha.\,\alpha \to (\sigma \to \alpha \to \alpha) \to \alpha$$

Abstraction within types

We have defined types that are parametric in other types:

$$\phi \vee \psi = \forall\theta.\, (\phi \to \theta) \to (\psi \to \theta) \to \theta$$
$$\sigma \times \rho = \forall\alpha.\, (\sigma \to \rho \to \alpha) \to \alpha$$
$$\mathbf{list}\ \sigma = \forall\alpha.\, \alpha \to (\sigma \to \alpha \to \alpha) \to \alpha$$

What if we could define them as type-level functions:

$$\vee = \Lambda\alpha, \beta.\, \forall\theta.\, (\alpha \to \theta) \to (\beta \to \theta) \to \theta$$
$$\times = \Lambda\beta, \gamma.\, \forall\alpha.\, (\beta \to \gamma \to \alpha) \to \alpha$$
$$\mathbf{list} = \Lambda\beta.\, \forall\alpha.\, \alpha \to (\beta \to \alpha \to \alpha) \to \alpha$$

Idea

We want to allow abstraction within types — type level functions.

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

$$\frac{}{(\lambda \alpha. \sigma) \rho \to_\beta \sigma[\rho/\alpha]}$$

Idea

We want to allow abstraction within types — type level functions.

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

$$\frac{}{(\lambda \alpha.\, \sigma)\rho \to_\beta \sigma[\rho/\alpha]}$$

We want to constrain our types; they should be strongly normalising.

Kinds of Types

Idea: our types themselves can be a simply-typed lambda calculus. The "types" of types are called *kinds*. We don't have kind variables, but we do have a constant base kind $*$ (pronounced "type").

$$\kappa ::= * \mid \kappa_1 \to \kappa_2$$

Kinds of Types

Idea: our types themselves can be a simply-typed lambda calculus. The "types" of types are called *kinds*. We don't have kind variables, but we do have a constant base kind $*$ (pronounced "type").

$$\kappa ::= * \mid \kappa_1 \to \kappa_2$$

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

Kinding Rules

$$\textsc{Start}\ \frac{\alpha \notin \mathrm{FV}(\Gamma)}{\Gamma, \alpha : \kappa \vdash \alpha : \kappa} \qquad \to\ \textsc{formation}\ \frac{\Gamma \vdash \sigma : * \qquad \Gamma \vdash \rho : *}{\Gamma \vdash \sigma \to \rho : *} \qquad \forall\ \textsc{formation}\ \frac{\Gamma, \alpha : \kappa \vdash \sigma : *}{\Gamma \vdash \forall \alpha : \kappa . \sigma : *}$$

$$\to\ \textsc{introduction}\ \frac{\Gamma, \alpha : \kappa_1 \vdash \sigma : \kappa_2}{\Gamma \vdash \lambda \alpha : \kappa_1 . \sigma : \kappa_1 \to \kappa_2}$$

$$\to\ \textsc{elimination}\ \frac{\Gamma \vdash \sigma : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash \rho : \kappa_1}{\Gamma \vdash \sigma\rho : \kappa_2}$$

Terms

$$\kappa ::= * \mid \kappa_1 \to \kappa_2$$
$$\sigma, \rho ::= \alpha \mid \sigma \to \rho \mid \forall \alpha : \kappa. \sigma \mid \lambda \alpha : \kappa. \sigma \mid \sigma \rho$$
$$M, N ::= x \mid \lambda x : \sigma. M \mid MN \mid \lambda \alpha : \kappa. M \mid M\sigma$$

Typing Rules

$$\textsc{Start}\ \frac{\Gamma \vdash \sigma : * \qquad x \notin \mathrm{FV}(\Gamma)}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\rightarrow\ \textsc{introduction}\ \frac{\Gamma, x : \sigma \vdash M : \rho \qquad \Gamma \vdash \sigma \rightarrow \rho : *}{\Gamma \vdash \lambda x : \sigma.\, M : \sigma \rightarrow \rho}$$

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

$$\forall\ \textsc{introduction}\ \frac{\Gamma, \alpha : \kappa \vdash M : \rho \qquad \Gamma \vdash \forall \alpha : \kappa.\, \rho : *}{\Gamma \vdash \lambda \alpha : \kappa.\, M : \forall \alpha : \kappa.\, \rho}$$

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

$$\textsc{Conversion}\ \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma' : * \qquad \sigma =_\beta \sigma'}{\Gamma \vdash M : \sigma'}$$

Contexts

Previously, type variables didn't occur in contexts. (They all had kind $*$.)

Contexts

Previously, type variables didn't occur in contexts. (They all had kind $*$.)
Now they can have different kinds, so we need to keep track of them in the context.
Moreover, since the types of term variables can contain type variables, the order is significant: the type variable should be kinded before it is used.

Contexts

Previously, type variables didn't occur in contexts. (They all had kind $*$.)
Now they can have different kinds, so we need to keep track of them in the context.
Moreover, since the types of term variables can contain type variables, the order is significant: the type variable should be kinded before it is used.
Thus a context is a *list* of pairs $\alpha : \kappa$ and $x : \sigma$.

Weakening Rules

$$\frac{\Gamma \vdash \sigma : \kappa \qquad \alpha \notin \mathrm{FV}(\Gamma)}{\Gamma, \alpha : \kappa' \vdash \sigma : \kappa}$$

$$\frac{\Gamma \vdash \sigma : \kappa \qquad \Gamma \vdash \alpha : \kappa' \qquad x \notin \mathrm{FV}(\Gamma)}{\Gamma, x : \alpha \vdash \sigma : \kappa}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \alpha \notin \mathrm{FV}(\Gamma)}{\Gamma, \alpha : \kappa' \vdash M : \sigma}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \alpha : \kappa' \qquad x \notin \mathrm{FV}(\Gamma)}{\Gamma, x : \alpha \vdash M : \sigma}$$

Lists

We can define a type constructor for lists:

$$\mathbf{list} = \lambda\beta : *. \forall\alpha : *. \alpha \to (\beta \to \alpha \to \alpha) \to \alpha$$

Kindedness Checking **list**

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\beta : *, \alpha : * \vdash \alpha : *}\,\textsc{St}}{\beta : *, \alpha : * \vdash \alpha : *}\,\textsc{St}}{\beta : *, \alpha : * \vdash \beta : *}\,\textsc{Wk} \quad \dfrac{\dfrac{}{\beta : *, \alpha : * \vdash \alpha : *}\,\textsc{St}}{\beta : *, \alpha : * \vdash \alpha \to \alpha : *}\,{\to}\textsc{F}}{\beta : *, \alpha : * \vdash \beta \to \alpha \to \alpha : *}\,{\to}\textsc{F} \quad \dfrac{\dfrac{}{\beta : *, \alpha : * \vdash \alpha : *}\,\textsc{St}}{\beta : *, \alpha : * \vdash (\beta \to \alpha \to \alpha) \to \alpha : *}\,{\to}\textsc{F}}{\beta : *, \alpha : * \vdash \alpha \to (\beta \to \alpha \to \alpha) \to \alpha : *}\,{\to}\textsc{F}}{\beta : * \vdash \forall \alpha : *.\, \alpha \to (\beta \to \alpha \to \alpha) \to \alpha : *}\,\forall\textsc{F}}{\vdash \lambda \beta : *.\, \forall \alpha : *.\, \alpha \to (\beta \to \alpha \to \alpha) \to \alpha : * \to *}\,{\to}\textsc{I}
$$

Mapping Over Lists

A natural operation on a list is to apply a function on each element.

Mapping Over Lists

A natural operation on a list is to apply a function on each element.

$$\mathsf{map} : \forall \alpha : *, \beta : *. (\alpha \to \beta) \to \mathbf{list}\ \alpha \to \mathbf{list}\ \beta$$

Mapping Over Lists

A natural operation on a list is to apply a function on each element.

$$\mathsf{map} : \forall \alpha : *, \beta : *. \, (\alpha \to \beta) \to \mathbf{list}\ \alpha \to \mathbf{list}\ \beta$$

$$\mathsf{map} = \lambda \alpha : *, \beta : *. \, \lambda f : \alpha \to \beta, l : \mathbf{list}\ \alpha.$$
$$\lambda \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma. \, l \gamma n (\lambda x : \alpha. \, c (f x))$$

The idea behind map

A term $l : \mathbf{list}\ \alpha$ behaves like a fold over the list it represents.

The idea behind map

A term $l : \mathbf{list}\ \alpha$ behaves like a fold over the list it represents.
In principle $l\ (\mathbf{list}\ \alpha)$ nil cons reconstructs the list.

The idea behind map

A term $l : \mathbf{list}\ \alpha$ behaves like a fold over the list it represents.
In principle $l\ (\mathbf{list}\ \alpha)$ nil cons reconstructs the list.
We want to reconstruct the list, but transmuting all the elements with $f : \alpha \to \beta$. Instead of cons we thus want $\lambda x : \alpha.\, \mathsf{cons}(fx)$.

The idea behind map

A term $l : \mathbf{list}\ \alpha$ behaves like a fold over the list it represents.
In principle $l\ (\mathbf{list}\ \alpha)$ nil cons reconstructs the list.
We want to reconstruct the list, but transmuting all the elements with $f : \alpha \to \beta$. Instead of cons we thus want $\lambda x : \alpha.\, \mathsf{cons}(fx)$.
To construct a list, we abstract over the representation type and the constructors:

$$\lambda\gamma : *, n : \gamma, c : (\beta \to \gamma \to \gamma).\, l\gamma n(\lambda x : \alpha.\, c(fx))$$

The idea behind map

A term $l : \mathbf{list}\ \alpha$ behaves like a fold over the list it represents.
In principle $l\ (\mathbf{list}\ \alpha)$ nil cons reconstructs the list.
We want to reconstruct the list, but transmuting all the elements with $f : \alpha \to \beta$. Instead of cons we thus want $\lambda x : \alpha.\, \mathsf{cons}(fx)$.
To construct a list, we abstract over the representation type and the constructors:

$$\lambda\gamma : *, n : \gamma, c : (\beta \to \gamma \to \gamma).\, l\gamma n(\lambda x : \alpha.\, c(fx))$$

$$\mathsf{map} = \lambda\alpha : *, \beta : *.\, \lambda f : \alpha \to \beta, l : \mathbf{list}\ \alpha.$$
$$\lambda\gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma.\, l\gamma n(\lambda x : \alpha.\, c(fx))$$

Typing map

Goal:

$$\vdash \lambda\alpha : *, \beta : *.\, \lambda f : \alpha \to \beta, l : \mathbf{list}\ \alpha.\, \lambda\gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma.\, l\gamma n(\lambda x : \alpha.\, c(fx))$$
$$: \forall\alpha : *, \beta : *.\, (\alpha \to \beta) \to \mathbf{list}\ \alpha \to \mathbf{list}\ \beta$$

Typing map

Goal:

$$\vdash \lambda\alpha : *, \beta : *.\, \lambda f : \alpha \to \beta, l : \mathbf{list}\ \alpha.\, \lambda\gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma.\, l\gamma n(\lambda x : \alpha.\, c(fx))$$
$$: \forall\alpha : *, \beta : *.\, (\alpha \to \beta) \to \mathbf{list}\ \alpha \to \mathbf{list}\ \beta$$

1. Apply $\forall$ INTRODUCTION twice

Typing map

Goal:

$$\alpha : *, \beta : * \vdash \lambda f : \alpha \to \beta, l : \mathbf{list}\ \alpha.\, \lambda\gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma.\, l\gamma n(\lambda x : \alpha.\, c(fx)) \\ : (\alpha \to \beta) \to \mathbf{list}\ \alpha \to \mathbf{list}\ \beta$$

Typing map

Goal:

$$\alpha : *, \beta : * \vdash \lambda f : \alpha \to \beta, l : \mathbf{list}\ \alpha.\, \lambda \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma.\, l \gamma n (\lambda x : \alpha.\, c(fx)) \\ : (\alpha \to \beta) \to \mathbf{list}\ \alpha \to \mathbf{list}\ \beta$$

2. Apply $\to$ INTRODUCTION twice

Typing map

Goal:

$$\alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha \vdash \lambda \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma.\, l \gamma n (\lambda x : \alpha.\, c (f x)) : \mathbf{list}\ \beta$$

Typing map

Goal:

$$\alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha \vdash \lambda\gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma.\, l\gamma n(\lambda x : \alpha.\, c(fx)) : \mathbf{list}\ \beta$$

3. Apply Conversion

$$\begin{aligned}\mathbf{list}\ \beta &= (\lambda\beta : *.\, \forall\alpha : *.\, \alpha \to (\beta \to \alpha \to \alpha) \to \alpha)\beta \\ &=_\beta \forall\gamma.\, \gamma \to (\beta \to \gamma \to \gamma) \to \gamma\end{aligned}$$

Typing map

Goal:

$$\alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha \vdash \lambda\gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma.\, l\gamma n(\lambda x : \alpha.\, c(fx))$$
$$: \forall\gamma.\, \gamma \to (\beta \to \gamma \to \gamma) \to \gamma$$

Typing map

Goal:

$$\alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha \vdash \lambda\gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma.\, l\gamma n(\lambda x : \alpha.\, c(fx))$$
$$: \forall\gamma.\, \gamma \to (\beta \to \gamma \to \gamma) \to \gamma$$

4. Apply $\forall$ INTRODUCTION; then $\to$ INTRODUCTION twice

Typing map

Goal:

$$\alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma \vdash l\gamma n(\lambda x : \alpha.\, c(fx)) : \gamma$$

Typing map

Goal:

$$\alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma \vdash l\gamma n(\lambda x : \alpha.\, c(fx)) : \gamma$$

5. Apply $\to$ ELIMINATION

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l\gamma n : ?_1 \to \gamma \tag{A}$$

$$\Gamma \vdash \lambda x : \alpha.\, c(fx) : ?_1 \tag{B}$$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l\gamma n : ?_1 \to \gamma \tag{A}$$

$$\Gamma \vdash \lambda x : \alpha.\, c(fx) : ?_1 \tag{B}$$

A.1 Apply $\to$ ELIMINATION

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l\gamma : ?_2 \to ?_1 \to \gamma \tag{A.A}$$

$$\Gamma \vdash n : ?_2 \tag{A.B}$$

$$\Gamma \vdash \lambda x : \alpha.\, c(fx) : ?_1 \tag{B}$$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l\gamma : ?_2 \to ?_1 \to \gamma \tag{A.A}$$

$$\Gamma \vdash n : ?_2 \tag{A.B}$$

$$\Gamma \vdash \lambda x : \alpha.\, c(fx) : ?_1 \tag{B}$$

A.B.1 Apply Weakening and Start: $?_2 = \gamma$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l\gamma : \gamma \to ?_1 \to \gamma \tag{A.A}$$

$$\Gamma \vdash \lambda x : \alpha.\, c(fx) : ?_1 \tag{B}$$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l\gamma : \gamma \to ?_1 \to \gamma \tag{A.A}$$

$$\Gamma \vdash \lambda x : \alpha.\, c(fx) : ?_1 \tag{B}$$

A.A.1 Apply $\forall$ ELIMINATION

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : ?_3.\, ?_4 \tag{A.A.A}$$

$$\Gamma \vdash \gamma : ?_3 \tag{A.A.B}$$

$$\Gamma \vdash \lambda x : \alpha.\, c(fx) : ?_1 \tag{B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to ?_1 \to \gamma$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : ?_3.\, ?_4 \tag{A.A.A}$$

$$\Gamma \vdash \gamma : ?_3 \tag{A.A.B}$$

$$\Gamma \vdash \lambda x : \alpha.\, c(fx) : ?_1 \tag{B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to ?_1 \to \gamma$
A.A.B.1 Weakening (twice) and Start: $?_3 = *$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *.\, ?_4 \tag{A.A.A}$$

$$\Gamma \vdash \lambda x : \alpha.\, c(f x) : ?_1 \tag{B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to ?_1 \to \gamma$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *.\, ?_4 \tag{A.A.A}$$

$$\Gamma \vdash \lambda x : \alpha.\, c(f x) : ?_1 \tag{B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to ?_1 \to \gamma$
B.1 Apply $\to$ INTRODUCTION: $?_1 = \alpha \to ?_5$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *.\, ?_4 \tag{A.A.A}$$

$$\Gamma, x : \alpha \vdash c(fx) : ?_5 \tag{B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to (\alpha \to ?_5) \to \gamma$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *.\, ?_4 \tag{A.A.A}$$

$$\Gamma, x : \alpha \vdash c(fx) : ?_5 \tag{B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to (\alpha \to ?_5) \to \gamma$

B.2 Apply $\to$ ELIMINATION

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *.\, ?_4 \tag{A.A.A}$$

$$\Gamma, x : \alpha \vdash c : ?_6 \to ?_5 \tag{B.A}$$

$$\Gamma, x : \alpha \vdash f x : ?_6 \tag{B.B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to (\alpha \to ?_5) \to \gamma$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *.\, ?_4 \tag{A.A.A}$$

$$\Gamma, x : \alpha \vdash c : ?_6 \to ?_5 \tag{B.A}$$

$$\Gamma, x : \alpha \vdash fx : ?_6 \tag{B.B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to (\alpha \to ?_5) \to \gamma$
B.A.1 Apply Weakening and Start: $?_5 = \gamma \to \gamma$, $?_6 = \beta$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *. ?_4 \tag{A.A.A}$$

$$\Gamma, x : \alpha \vdash f x : \beta \tag{B.B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to (\alpha \to \gamma \to \gamma) \to \gamma$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *. ?_4 \tag{A.A.A}$$

$$\Gamma, x : \alpha \vdash f x : \beta \tag{B.B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to (\alpha \to \gamma \to \gamma) \to \gamma$

B.B.1 Apply $\to$ ELIMINATION

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *. ?_4 \tag{A.A.A}$$

$$\Gamma, x : \alpha \vdash f : ?_7 \to \beta \tag{B.B.A}$$

$$\Gamma, x : \alpha \vdash x : ?_7 \tag{B.B.B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to (\alpha \to \gamma \to \gamma) \to \gamma$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \textbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *. ?_4 \tag{A.A.A}$$

$$\Gamma, x : \alpha \vdash f : ?_7 \to \beta \tag{B.B.A}$$

$$\Gamma, x : \alpha \vdash x : ?_7 \tag{B.B.B}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to (\alpha \to \gamma \to \gamma) \to \gamma$
B.B.A.1 Apply Weakening and Start: $?_7 = \alpha$
B.B.B.1 Apply Start: $?_7 = \alpha$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \textbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *. ?_4 \qquad \text{(A.A.A)}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to (\alpha \to \gamma \to \gamma) \to \gamma$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \textbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *.\, ?_4 \qquad \text{(A.A.A)}$$

Constraint: $?_4[\gamma/\delta] = \gamma \to (\alpha \to \gamma \to \gamma) \to \gamma$
A.A.A.1 Instantiate $?_4 = \delta \to (\alpha \to \delta \to \delta) \to \delta$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *. \delta \to (\alpha \to \delta \to \delta) \to \delta \qquad \text{(A.A.A)}$$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \mathbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \forall \delta : *. \delta \to (\alpha \to \delta \to \delta) \to \delta \tag{A.A.A}$$

A.A.A.2 Apply CONVERSION:

$$\forall \delta : *. \delta \to (\alpha \to \delta \to \delta) \to \delta =_\beta (\lambda \beta : *. \forall \alpha : *. \alpha \to (\beta \to \alpha \to \alpha) \to \alpha)\alpha = \mathbf{list}\ \alpha$$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \textbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \textbf{list}\ \alpha \qquad \text{(A.A.A)}$$

Typing map

$$\Gamma = \alpha : *, \beta : *, f : \alpha \to \beta, l : \textbf{list}\ \alpha, \gamma : *, n : \gamma, c : \beta \to \gamma \to \gamma$$

Goals:

$$\Gamma \vdash l : \textbf{list}\ \alpha \tag{A.A.A}$$

A.A.A.2 Apply Weakening and Start

Generalised map

We could define map-like functions for other type constructors. For instance, a function on **tree** that applies the given function to all leaves of the tree. Or on $\lambda\alpha : *. \alpha$, we could define it to be

$$\lambda\beta : *, \gamma : *, f : \beta \to \gamma, a : ((\lambda\alpha : *. \alpha)\beta). fa$$

Generalised map

We could define map-like functions for other type constructors. For instance, a function on **tree** that applies the given function to all leaves of the tree. Or on $\lambda\alpha : *. \alpha$, we could define it to be

$$\lambda\beta : *, \gamma : *, f : \beta \to \gamma, a : ((\lambda\alpha : *. \alpha)\beta). fa$$

We can define a higher-order type constructor that generates the type of map-like functions for a type constructor of kind $* \to *$:

$$\mathbf{Map} = \lambda\mathbf{A} : (* \to *). (\forall\alpha : *, \beta : *. (\alpha \to \beta) \to \mathbf{A}\ \alpha \to \mathbf{A}\ \beta)$$

$$\vdash \mathbf{Map} : (* \to *) \to *$$

Composing maps

$$\mathbf{Map} = \lambda \mathbf{A} : (* \to *).\, (\forall \alpha : *, \beta : *.\, (\alpha \to \beta) \to \mathbf{A}\ \alpha \to \mathbf{A}\ \beta)$$

$$\mathsf{mapComp} : \forall \mathbf{A} : (* \to *).\, \mathbf{Map}\ \mathbf{A} \to \forall \mathbf{B} : (* \to *).\, \mathbf{Map}\ \mathbf{B} \to \mathbf{Map}\ (\lambda \alpha : *.\, \mathbf{A}\ (\mathbf{B}\ \alpha))$$

$$\mathsf{mapComp} = \lambda \mathbf{A} : * \to *, \mathsf{mapA} : \mathbf{Map}\ \mathbf{A}, \mathbf{B} : * \to *, \mathsf{mapB} : \mathbf{Map}\ \mathbf{B}.$$
$$\lambda \alpha : *, \beta : *, f : \alpha \to \beta.\, \mathsf{mapA}\ (\mathbf{B}\ \alpha)\ (\mathbf{B}\ \beta)\ (\mathsf{mapB}\ \alpha\ \beta\ f)$$

$\mathsf{F}_0, \mathsf{F}_1, \mathsf{F}_2, \ldots, \mathsf{F}_\omega$

Why F_ω?
System $\mathsf{F} = \mathsf{F}_0$ has only one kind:

$$\mathbb{K}_0 \ni \kappa^0 ::= *$$

$\mathsf{F}_0, \mathsf{F}_1, \mathsf{F}_2, \ldots, \mathsf{F}_\omega$

Why F_ω?
System $\mathsf{F} = \mathsf{F}_0$ has only one kind:

$$\mathbb{K}_0 \ni \kappa^0 ::= *$$

System F_1 has kinds:

$$\mathbb{K}_1 \ni \kappa^1 ::= * \mid \kappa^0 \to \kappa^1$$

$\mathsf{F}_0, \mathsf{F}_1, \mathsf{F}_2, \ldots, \mathsf{F}_\omega$

Why F_ω?
System $\mathsf{F} = \mathsf{F}_0$ has only one kind:

$$\mathbb{K}_0 \ni \kappa^0 ::= *$$

System F_1 has kinds:

$$\mathbb{K}_1 \ni \kappa^1 ::= * \mid \kappa^0 \to \kappa^1$$

System F_{n+1} has kinds:

$$\mathbb{K}_{n+1} \ni \kappa^{n+1} ::= * \mid \kappa^n \to \kappa^{n+1}$$

$\mathsf{F}_0, \mathsf{F}_1, \mathsf{F}_2, \ldots, \mathsf{F}_\omega$

Why F_ω?
System $\mathsf{F} = \mathsf{F}_0$ has only one kind:

$$\mathbb{K}_0 \ni \kappa^0 ::= *$$

System F_1 has kinds:

$$\mathbb{K}_1 \ni \kappa^1 ::= * \mid \kappa^0 \to \kappa^1$$

System F_{n+1} has kinds:

$$\mathbb{K}_{n+1} \ni \kappa^{n+1} ::= * \mid \kappa^n \to \kappa^{n+1}$$

System F_ω has kinds:

$$\mathbb{K}_\omega = \bigcup_{n \in \omega} \mathbb{K}_n$$

A Collapsed Presentation

Our presentation of F_ω separated *terms*, *types* and *kinds* into separate syntactic categories. An alternative presentation is to treat all three as one syntactic category, and use the type system to distinguish them.

A Collapsed Presentation

Our presentation of F_ω separated *terms, types* and *kinds* into separate syntactic categories. An alternative presentation is to treat all three as one syntactic category, and use the type system to distinguish them.

$$M, N, \sigma, \rho, \kappa ::= x \mid \mathbf{s} \mid MN \mid \lambda x : M.\, N \mid M \to N \mid \forall x : M.\, N$$

where $\mathbf{s} \in \{*, \Box\}$ ranges over *sorts*.
$*$ is the sort of types; $\Box$ is the sort of kinds.

$$\textsc{Axiom}\ \frac{}{\vdash * : \Box}$$

$$\textsc{Start}\ \frac{\Gamma \vdash \sigma : \mathbf{s} \qquad x \notin \mathrm{FV}(\Gamma)}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\textsc{Weakening}\ \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \rho : \mathbf{s} \qquad x \notin \mathrm{FV}(\Gamma)}{\Gamma, x : \rho \vdash M : \sigma}$$

$$\to \textsc{formation}\ \frac{\Gamma \vdash \sigma : \mathbf{s} \qquad \Gamma \vdash \rho : \mathbf{s}}{\Gamma \vdash \sigma \to \rho : \mathbf{s}}$$

$$\forall \textsc{formation}\ \frac{\Gamma \vdash \kappa : \Box \qquad \Gamma, \alpha : \kappa \vdash \sigma : *}{\Gamma \vdash \forall \alpha : \kappa . \sigma : *}$$

$$\to \textsc{introduction}\ \frac{\Gamma, x : \sigma \vdash M : \rho \qquad \Gamma \vdash \sigma \to \rho : \mathbf{s}}{\Gamma \vdash \lambda x : \sigma . M : \sigma \to \rho}$$

$$\forall \textsc{introduction}\ \frac{\Gamma, \alpha : \kappa \vdash M : \sigma \qquad \Gamma \vdash \forall \alpha : \kappa . \sigma : \mathbf{s}}{\Gamma \vdash \lambda \alpha : \kappa . M : \forall \alpha : \kappa . \sigma}$$

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

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

$$\textsc{Conversion}\ \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma' : \mathbf{s} \qquad \sigma =_\beta \sigma'}{\Gamma \vdash M : \sigma'}$$

Properties of F_ω

Theorem. System F_ω is strongly normalising (with respect to $\rightarrow_\beta$).

Theorem. Type-checking in System F_ω is decidable.

Section 2

CoC

Calculus of Constructions

So far, we have seen:

- terms parametrised by terms $\lambda x : \sigma. M$, in simply-typed lambda calculus;
- terms parametrised by types $\Lambda\alpha. M$ (or $\lambda\alpha : *. M$), in System F; and
- types parametrised by types $\lambda\alpha : *. \sigma$, in System F_ω.

Calculus of Constructions

So far, we have seen:

- terms parametrised by terms $\lambda x : \sigma . M$, in simply-typed lambda calculus;
- terms parametrised by types $\Lambda \alpha . M$ (or $\lambda \alpha : * . M$), in System F; and
- types parametrised by types $\lambda \alpha : * . \sigma$, in System F_ω.

The Calculus of Constructions (Coquand and Huet, 1985) adds

- types parametrised by terms.

Calculus of Constructions

So far, we have seen:

- terms parametrised by terms $\lambda x : \sigma. M$, in simply-typed lambda calculus;
- terms parametrised by types $\Lambda\alpha. M$ (or $\lambda\alpha : *. M$), in System F; and
- types parametrised by types $\lambda\alpha : *. \sigma$, in System F_ω.

The Calculus of Constructions (Coquand and Huet, 1985) adds

- types parametrised by terms.

The terms of CoC are:

$$M, N, \sigma, \rho, \kappa ::= x \mid \mathbf{s} \mid MN \mid \lambda x : M. N \mid \forall x : M. N$$

where x ranges over syntactic variables and $\mathbf{s} \in \{*, \Box\}$ ranges over sorts.

Calculus of Constructions

So far, we have seen:

- terms parametrised by terms $\lambda x : \sigma . M$, in simply-typed lambda calculus;
- terms parametrised by types $\Lambda \alpha . M$ (or $\lambda \alpha : * . M$), in System F; and
- types parametrised by types $\lambda \alpha : * . \sigma$, in System F_ω.

The Calculus of Constructions (Coquand and Huet, 1985) adds

- types parametrised by terms.

The terms of CoC are:

$$M, N, \sigma, \rho, \kappa ::= x \mid \mathbf{s} \mid MN \mid \lambda x : M . N \mid \forall x : M . N$$

where x ranges over syntactic variables and $\mathbf{s} \in \{*, \Box\}$ ranges over sorts.
Notably, $M \to N$ is absent from the syntax; we treat it as syntactic sugar (i.e. a notational convention) for $\forall x : M . N$, where $x \notin \mathrm{FV}(N)$.

Axiom

$$\frac{}{\vdash * : \Box}$$

Start

$$\frac{\Gamma \vdash \sigma : \mathbf{s} \qquad x \notin \mathrm{FV}(\Gamma)}{\Gamma, x : \sigma \vdash x : \sigma}$$

Weakening

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \rho : \mathbf{s} \qquad x \notin \mathrm{FV}(\Gamma)}{\Gamma, x : \rho \vdash M : \sigma}$$

$\forall$ formation

$$\frac{\Gamma \vdash \sigma : \mathbf{s}_1 \qquad \Gamma, x : \sigma \vdash \rho : \mathbf{s}_2}{\Gamma \vdash \forall x : \sigma.\, \rho : \mathbf{s}_2}$$

$\forall$ introduction

$$\frac{\Gamma, x : \rho \vdash M : \sigma \qquad \Gamma \vdash \forall x : \rho.\, \sigma : \mathbf{s}}{\Gamma \vdash \lambda x : \rho.\, M : \forall x : \rho.\, \sigma}$$

$\forall$ elimination

$$\frac{\Gamma \vdash M : \forall x : \rho.\, \sigma \qquad \Gamma \vdash N : \rho}{\Gamma \vdash MN : \sigma[N/x]}$$

Conversion

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma' : \mathbf{s} \qquad \sigma =_\beta \sigma'}{\Gamma \vdash M : \sigma'}$$

$$\textsc{Axiom}\ \frac{}{\vdash * : \Box}$$

$$\textsc{Start}\ \frac{\Gamma \vdash \sigma : \mathbf{s} \qquad x \notin \mathrm{FV}(\Gamma)}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\textsc{Weakening}\ \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \rho : \mathbf{s} \qquad x \notin \mathrm{FV}(\Gamma)}{\Gamma, x : \rho \vdash M : \sigma}$$

$$\forall\ \textsc{formation}\ \frac{\Gamma \vdash \sigma : \mathbf{s}_1 \qquad \Gamma, x : \sigma \vdash \rho : \mathbf{s}_2}{\Gamma \vdash \forall x : \sigma.\, \rho : \mathbf{s}_2}$$

$$\forall\ \textsc{introduction}\ \frac{\Gamma, x : \rho \vdash M : \sigma \qquad \Gamma \vdash \forall x : \rho.\, \sigma : \mathbf{s}}{\Gamma \vdash \lambda x : \rho.\, M : \forall x : \rho.\, \sigma}$$

$$\forall\ \textsc{elimination}\ \frac{\Gamma \vdash M : \forall x : \rho.\, \sigma \qquad \Gamma \vdash N : \rho}{\Gamma \vdash MN : \sigma[N/x]}$$

$$\textsc{Conversion}\ \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma' : \mathbf{s} \qquad \sigma =_\beta \sigma'}{\Gamma \vdash M : \sigma'}$$

In F_ω, the $\forall$ FORMATION rule restricts $\mathbf{s}_1 = \Box$ and $\mathbf{s}_2 = *$.
When $\mathsf{s}_1 = \mathsf{s}_2$, the $\forall$ FORMATION rule generalises the $\to$ FORMATION rule.

Types of Abstractions 1

The type of a term parametrised by a term, by choosing $\mathbf{s}_1 = *$ and $\mathbf{s}_2 = *$:

$$\forall \mathrm{F} \; \frac{\Gamma \vdash \sigma : * \qquad \Gamma, x : \sigma \vdash \rho : *}{\Gamma \vdash \forall x : \sigma . \rho : *}$$

Types of Abstractions 1

The type of a term parametrised by a term, by choosing $\mathbf{s}_1 = *$ and $\mathbf{s}_2 = *$:

$$\forall \mathrm{F} \ \frac{\Gamma \vdash \sigma : * \qquad \Gamma, x : \sigma \vdash \rho : *}{\Gamma \vdash \forall x : \sigma . \rho : *}$$

Previously, this would have been the type $\alpha \to \sigma$, formed as

$$\to \mathrm{F} \ \frac{\Gamma \vdash \sigma : * \qquad \Gamma \vdash \rho : *}{\Gamma \vdash \sigma \to \rho : *}$$

This can be seen as a special case, when $x \notin \mathrm{FV}(\rho)$. However, we now allow types (in this case ρ) to depend on term variables (x).

Types of Abstractions 1

The type of a term parametrised by a term, by choosing $\mathbf{s}_1 = *$ and $\mathbf{s}_2 = *$:

$$\forall \mathrm{F}\ \frac{\Gamma \vdash \sigma : * \qquad \Gamma, x : \sigma \vdash \rho : *}{\Gamma \vdash \forall x : \sigma.\, \rho : *}$$

Previously, this would have been the type $\alpha \to \sigma$, formed as

$$\to \mathrm{F}\ \frac{\Gamma \vdash \sigma : * \qquad \Gamma \vdash \rho : *}{\Gamma \vdash \sigma \to \rho : *}$$

This can be seen as a special case, when $x \notin \mathrm{FV}(\rho)$. However, we now allow types (in this case ρ) to depend on term variables (x).
Example: $\forall n : nat.\, \mathrm{vec} n$.
This is new in the Calculus of Constructions.

Types of Abstractions 2

The type of a term parametrised by a type, by choosing $\mathbf{s}_1 = \Box$ and $\mathbf{s}_2 = *$:

$$\forall \mathrm{F}\ \frac{\Gamma \vdash \kappa : \Box \qquad \Gamma, \alpha : \kappa \vdash \rho : *}{\Gamma \vdash \forall \alpha : \kappa . \rho : *}$$

Types of Abstractions 2

The type of a term parametrised by a type, by choosing $\mathbf{s}_1 = \Box$ and $\mathbf{s}_2 = *$:

$$\forall \mathrm{F}\ \frac{\Gamma \vdash \kappa : \Box \qquad \Gamma, \alpha : \kappa \vdash \rho : *}{\Gamma \vdash \forall \alpha : \kappa . \rho : *}$$

This is the version of the rule seen in F_ω. It is also essentially the same as for System F, except that we must check that κ is a well-formed kind, since we do not have syntactic stratification.

Types of Abstractions 2

The type of a term parametrised by a type, by choosing $\mathbf{s}_1 = \Box$ and $\mathbf{s}_2 = *$:

$$\forall \mathrm{F}\ \frac{\Gamma \vdash \kappa : \Box \qquad \Gamma, \alpha : \kappa \vdash \rho : *}{\Gamma \vdash \forall \alpha : \kappa . \rho : *}$$

This is the version of the rule seen in F_ω. It is also essentially the same as for System F, except that we must check that κ is a well-formed kind, since we do not have syntactic stratification. Example: $\forall \alpha : *. \mathrm{list} \alpha$.

Types of Abstractions 3

The kind of a type parametrised by a type, by choosing $\mathbf{s}_1 = \Box$ and $\mathbf{s}_2 = \Box$:

$$\forall \mathrm{F}\ \frac{\Gamma \vdash \kappa_1 : \Box \qquad \Gamma, \alpha : \kappa_1 \vdash \kappa_2 : \Box}{\Gamma \vdash \forall \alpha : \kappa_1 . \kappa_2 : \Box}$$

Types of Abstractions 3

The kind of a type parametrised by a type, by choosing $\mathbf{s}_1 = \Box$ and $\mathbf{s}_2 = \Box$:

$$\forall\mathrm{F}\ \frac{\Gamma \vdash \kappa_1 : \Box \qquad \Gamma, \alpha : \kappa_1 \vdash \kappa_2 : \Box}{\Gamma \vdash \forall \alpha : \kappa_1 . \kappa_2 : \Box}$$

This is a generalisation of the $\rightarrow$ FORMATION rule:

$$\rightarrow\mathrm{F}\ \frac{\Gamma \vdash \kappa_1 : \Box \qquad \Gamma \vdash \kappa_2 : \Box}{\Gamma \vdash \kappa_1 \rightarrow \kappa_2 : \Box}$$

Types of Abstractions 3

The kind of a type parametrised by a type, by choosing $\mathbf{s}_1 = \Box$ and $\mathbf{s}_2 = \Box$:

$$\forall\mathrm{F}\ \frac{\Gamma \vdash \kappa_1 : \Box \qquad \Gamma, \alpha : \kappa_1 \vdash \kappa_2 : \Box}{\Gamma \vdash \forall \alpha : \kappa_1 . \kappa_2 : \Box}$$

This is a generalisation of the $\rightarrow$ FORMATION rule:

$$\rightarrow\mathrm{F}\ \frac{\Gamma \vdash \kappa_1 : \Box \qquad \Gamma \vdash \kappa_2 : \Box}{\Gamma \vdash \kappa_1 \rightarrow \kappa_2 : \Box}$$

Example: $\forall \alpha : *. (\alpha \rightarrow *)$.

Types of Abstractions 4

The kind of a type parametrised by a term, by choosing $\mathbf{s}_1 = *$ and $\mathbf{s}_2 = \Box$:

$$\forall \mathrm{F} \; \frac{\Gamma \vdash \sigma : * \qquad \Gamma, x : \sigma \vdash \kappa : \Box}{\Gamma \vdash \forall x : \sigma . \kappa : \Box}$$

Types of Abstractions 4

The kind of a type parametrised by a term, by choosing $\mathbf{s}_1 = *$ and $\mathbf{s}_2 = \Box$:

$$\forall \mathrm{F}\ \frac{\Gamma \vdash \sigma : * \qquad \Gamma, x : \sigma \vdash \kappa : \Box}{\Gamma \vdash \forall x : \sigma . \kappa : \Box}$$

$\forall n : nat . *^n$. This is new in the Calculus of Constructions.

The Lambda Cube

We have seen three extensions to the basic lambda calculus:

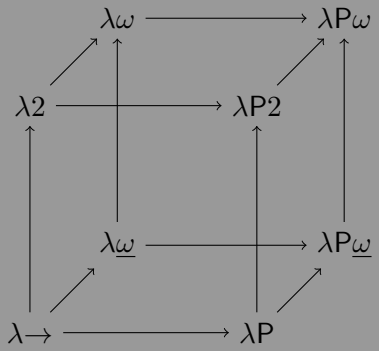
- terms depending on types
- types depending on types
- types depending on terms

These three extensions are orthogonal and correspond to allowing pairs of sorts in the $\forall$ formation rule:

$$\frac{\Gamma \vdash \sigma : \mathbf{s}_1 \qquad \Gamma, x : \sigma \vdash \rho : \mathbf{s}_2}{\Gamma \vdash \forall x : \sigma.\, \rho : \mathbf{s}_2} \; \forall \text{ FORMATION}$$

This gives rise to Barendregt's Lambda Cube

The Lambda Cube



$\lambda\!\rightarrow$: simply typed lambda calculus
$\lambda 2$: system F
$\lambda\omega$: F_ω
$\lambda\mathrm{P}\omega$: calculus of constructions.

Dependent Product

The type $\forall x : \rho. \sigma$ is called the *dependent product*.

Dependent Product

The type $\forall x : \rho. \sigma$ is called the *dependent product*.
In the literature, it is often written as $\prod x : \rho. \sigma$.

Dependent Product

The type $\forall x : \rho. \sigma$ is called the *dependent product*.
In the literature, it is often written as $\prod x : \rho. \sigma$.
It is a product because a term of type $\forall x : \rho. \sigma$ consists of "ρ copies of σ": i.e. for each term t of type ρ, we get a term of type $\sigma[t/x]$.

Dependent Types: $\mathbf{vector}$

A vector is like a list, but has a defined length.

$$\textbf{vector} : * \to \mathbf{nat} \to *$$

It has two constructors:

$$\text{vnil} : \mathbf{vector}\ \sigma\ 0$$
$$\text{vcons} : \forall n : \mathbf{nat}.\, \sigma \to \mathbf{vector}\ \sigma\ n \to \mathbf{vector}\ \sigma\ (\mathsf{S}n)$$

Dependent Types: **vector**

A vector is like a list, but has a defined length.

$$\textbf{vector} : * \to \textbf{nat} \to *$$

It has two constructors:

$$\begin{aligned} \text{vnil} &: \textbf{vector}\ \sigma\ 0 \\ \text{vcons} &: \forall n : \textbf{nat}.\, \sigma \to \textbf{vector}\ \sigma\ n \to \textbf{vector}\ \sigma\ (\mathsf{S}n) \end{aligned}$$

$$\textbf{vector} = \lambda\sigma : *, n : \textbf{nat}.\, \forall v : \textbf{nat} \to *.\, v0 \to (\forall n' : \textbf{nat}.\, \sigma \to vn' \to v(\mathsf{S}n')) \to vn$$

Propositions and Predicates

We can consider $*$ as the type of propositions.
A predicate is a parametrised proposition.

$$\text{Zero} : \mathbf{nat} \to *$$

Propositions and Predicates

We can consider $*$ as the type of propositions.
A predicate is a parametrised proposition.

$$\text{Zero} : \mathbf{nat} \to *$$

We would like Zero 0 to hold (i.e. be inhabited), but not Zero (Sn) for any n.
We can view this as an inductive type with a single constructor z0 : Zero 0.

Propositions and Predicates

We can consider $*$ as the type of propositions.
A predicate is a parametrised proposition.

$$\text{Zero} : \mathbf{nat} \to *$$

We would like Zero 0 to hold (i.e. be inhabited), but not Zero (Sn) for any n.
We can view this as an inductive type with a single constructor z0 : Zero 0.

$$\text{Zero} = \lambda n : \mathbf{nat}. \forall Z : \mathbf{nat} \to *. Z0 \to Zn$$

Leibniz Equality

$$\text{Zero} = \lambda n : \mathbf{nat}. \forall Z : \mathbf{nat} \to *. Z0 \to Zn$$

We can interpret Zero n as meaning: any property that 0 has, n also has.

Leibniz Equality

$$\mathsf{Zero} = \lambda n : \mathbf{nat}. \forall Z : \mathbf{nat} \to *. Z0 \to Zn$$

We can interpret Zero n as meaning: any property that 0 has, n also has.
We can generalise this into a notion of equality:

$$\mathsf{Id} = \lambda \alpha : *, x : \alpha, y : \alpha. \forall P : \alpha \to *. Px \to Py$$

This is called *Leibniz equality*.

Properties of Equality

$$\mathsf{Id} = \lambda \alpha : *, x : \alpha, y : \alpha. \forall P : \alpha \to *. Px \to Py$$

Properties of Equality

$$\mathsf{Id} = \lambda\alpha : *, x : \alpha, y : \alpha. \forall P : \alpha \to *. Px \to Py$$

Reflexivity:

$$\alpha : *, x : \alpha \vdash \lambda P : \alpha \to *. \lambda p : Px. p : \mathsf{Id}\ \alpha\ x\ x$$

Properties of Equality

$$\mathsf{Id} = \lambda \alpha : *, x : \alpha, y : \alpha. \forall P : \alpha \to *. Px \to Py$$

Reflexivity:

$$\alpha : *, x : \alpha \vdash \lambda P : \alpha \to *. \lambda p : Px. p : \mathsf{Id}\ \alpha\ x\ x$$

Symmetry:

$$\alpha : *, x : \alpha, y : \alpha, H : \mathsf{Id}\ \alpha\ x\ y \vdash H(\lambda z : \alpha. \mathsf{Id}\ \alpha\ z\ x)(\lambda P : \alpha \to *. \lambda p : Px. p) : \mathsf{Id}\ \alpha\ y\ x$$

Properties of Equality

$$\mathsf{Id} = \lambda \alpha : *, x : \alpha, y : \alpha. \forall P : \alpha \to *. Px \to Py$$

Reflexivity:

$$\alpha : *, x : \alpha \vdash \lambda P : \alpha \to *. \lambda p : Px. p : \mathsf{Id}\ \alpha\ x\ x$$

Symmetry:

$$\alpha : *, x : \alpha, y : \alpha, H : \mathsf{Id}\ \alpha\ x\ y \vdash H(\lambda z : \alpha. \mathsf{Id}\ \alpha\ z\ x)(\lambda P : \alpha \to *. \lambda p : Px. p) : \mathsf{Id}\ \alpha\ y\ x$$

Transitivity (exercise):

$$\alpha : *, x : \alpha, y : \alpha, z : \alpha, H_1 : \mathsf{Id}\ \alpha\ x\ y, H_2 : \mathsf{Id}\ \alpha\ y\ z \vdash ? : \mathsf{Id}\ \alpha\ x\ z$$

Functional Extensionality

In set theory, if functions f and g satisfy $\forall x.\, f(x) = g(x)$ then $f = g$.

Functional Extensionality

In set theory, if functions f and g satisfy $\forall x.\, f(x) = g(x)$ then $f = g$.

In CoC, this can be embodied by a term of type

$$\forall \alpha : *, \beta : *, f : \alpha \to \beta, g : \alpha \to \beta.\, (\forall x : \alpha.\, \mathsf{Id}\ \beta\ (f x)\ (g x)) \to \mathsf{Id}\ (\alpha \to \beta)\ f\ g$$

Functional Extensionality

In set theory, if functions f and g satisfy $\forall x.\, f(x) = g(x)$ then $f = g$.

In CoC, this can be embodied by a term of type

$$\forall \alpha : *, \beta : *, f : \alpha \to \beta, g : \alpha \to \beta. (\forall x : \alpha. \mathsf{Id}\ \beta\ (f x)\ (g x)) \to \mathsf{Id}\ (\alpha \to \beta)\ f\ g$$

However, there is no closed term of this type.

Functional Extensionality

In set theory, if functions f and g satisfy $\forall x.\, f(x) = g(x)$ then $f = g$.

In CoC, this can be embodied by a term of type

$$\forall \alpha : *, \beta : *, f : \alpha \to \beta, g : \alpha \to \beta.\, (\forall x : \alpha.\, \mathsf{Id}\ \beta\ (f x)\ (g x)) \to \mathsf{Id}\ (\alpha \to \beta)\ f\ g$$

However, there is no closed term of this type.
We can, however, postulate such a term as an *axiom* without introducing inconsistency.

Properties of CoC

Theorem. The Calculus of Constructions is strongly normalising (with respect to $\rightarrow_\beta$).

Theorem. Type-checking in the Calculus of Constructions is decidable.

Why not $* : *$?

In CoC we have the axiom $* : \Box$. But why don't we just get rid of $\Box$ and have the axiom $* : *$?

Why not $* : *$?

In CoC we have the axiom $* : \Box$. But why don't we just get rid of $\Box$ and have the axiom $* : *$? Unfortunately, such a system is inconsistent: all types are inhabited, and some well-typed terms do not have normal forms (Girard's Paradox).

Why not $* : *$?

In CoC we have the axiom $* : \Box$. But why don't we just get rid of $\Box$ and have the axiom $* : *$?
Unfortunately, such a system is inconsistent: all types are inhabited, and some well-typed terms do not have normal forms (Girard's Paradox).
Girard's Paradox is not trivial to show. It is related to paradoxes in naïve set theory.

Russell's Paradox

In naïve set theory, we can define "the set of all sets that do not contain themselves":

$$R = \{x \mid x \notin x\}$$

Russell's Paradox

In naïve set theory, we can define "the set of all sets that do not contain themselves":

$$R = \{x \mid x \notin x\}$$

Is $R \in R$? If so, then by definition $R \notin R$. But if not, then it must be that $R \in R$. A contradiction!

Russell's Paradox

In naïve set theory, we can define "the set of all sets that do not contain themselves":

$$R = \{x \mid x \notin x\}$$

Is $R \in R$? If so, then by definition $R \notin R$. But if not, then it must be that $R \in R$. A contradiction!
This paradox arises from considering the totality of sets to be itself a set.

Russell's Paradox

In naïve set theory, we can define "the set of all sets that do not contain themselves":

$$R = \{x \mid x \notin x\}$$

Is $R \in R$? If so, then by definition $R \notin R$. But if not, then it must be that $R \in R$. A contradiction!
This paradox arises from considering the totality of sets to be itself a set.
Analogously, $* : *$ (type in type) leads to a paradox.