Functional Programming Lecture Notes - Part 3

Thomas Dinsdale-Young

15th September, 2017

5 System $\mathbf{F}_\omega$

In the previous section, we defined a number of types *in terms of* other types, such as **list** σ and $\sigma \times \tau$. However, **list** and $\times$ are not part of the language themselves; rather, they are notations defined outside the language. One way of extending the calculus would be to allow type-level functions, by allowing abstraction over types at the type level. We could thus define **list** and $\times$ as functions within the language.

Suppose that we wish to define something like

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

This is not itself a type. It is a function that takes a type and returns a type. By allowing abstraction and application at the type level, we need to be able to characterise terms that are well-formed types. We thus introduce a higher-level type of (basic) types, $*$. The idea then is that we should have $\mathbf{list} : * \to *$. We call these higher-level types *kinds*, which are defined as:

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

Now that we have $*$, we can dispense with Λ by considering $\Lambda\beta.\, M$ as syntax for $\lambda\beta : *.\, M$. Our terms now can abstract over variables of two sorts:

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

Correspondingly, our types include polymorphism at a given kind and type abstraction and application:

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

Since we have abstraction at the type level, we also have β-reduction at the type level. The reduction rules for redexes are:

$$\overline{(\lambda\alpha : \kappa.\, \sigma)\rho \to_\beta \sigma[\rho/\alpha]} \qquad \overline{(\lambda\alpha : \kappa.\, M)\rho \to_\beta M[\rho/\alpha]}$$

$$\overline{(\lambda x : \sigma.\, M)N \to_\beta M[N/x]}$$

We also have all contextual rules, permitting reduction of arbitrary subterms (at both the type and term level). For instance

$$\frac{\sigma \to_\beta \sigma'}{M\sigma \to_\beta M\sigma'}$$

Our type system consists of two judgements: $\Gamma \vdash M : \sigma$, that a term has a given type in a context; and $\Gamma \vdash \sigma : \kappa$, that a type-level term has a given kind in a context.

Since we can abstract over types at the type level, contexts contain both type judgements ($x : \sigma$) and kind judgements ($\alpha : \kappa$). Because the type of a term variable may include a type variable in the context, we require that the context is linearly ordered (i.e. it is a list). (We require that variables are not declared more than once in a given context, however.)

Since we have abstraction at the type level, we need typing rules to determine when type-formers are well-kinded. The rules are:

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

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

Note that this is essentially a simply-typed lambda calculus, but with additional formation rules for the type constructors $\to$ and $\forall$, which construct types of kind $*$. Note that we can only have $\to$ between types of kind $*$. Moreover, we can only $\forall$-quantify type variables, and only over types of kind $*$. In particular $(\lambda \alpha : \kappa. \alpha) \to \beta$ is not a well-kinded type in any context, since $\vdash (\lambda \alpha : \kappa. \alpha) : * \to *$ (and not $*$).

The typing rules for term-formers are:

$$\textsc{Start}\ \frac{\Gamma \vdash \sigma : * \qquad x \notin \mathrm{FV}(\Gamma)}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \to\ \textsc{introduction}\ \frac{\Gamma, x : \sigma \vdash M : \rho \qquad \Gamma \vdash \sigma \to \rho : *}{\Gamma \vdash \lambda x : \sigma. M : \sigma \to \rho}$$

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

$$\forall\ \textsc{elimination}\ \frac{\Gamma \vdash M : \forall \alpha : \kappa. \rho \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash M\sigma : \rho[\sigma/\alpha]} \qquad \textsc{Conversion}\ \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma' : * \qquad \sigma =_\beta \sigma'}{\Gamma \vdash M : \sigma'}$$

These rules (except for Conversion) are essentially the same as those for System F, except that the introduction rules check that the type being constructed has kind $*$. All of the types of System F have kind $*$, so there is no need for this check there. Note that the rules guarantee that the kind of the type of a term will always be $*$. So for instance, we can have terms of type **list nat**, but not of type $\lambda \alpha.$ **list** α.

The Conversion rule allows us to exploit type-level computation. Specifically, we can change the type of a term to any β-equivalent type. Note that this is the first typing rule that has not been syntax-directed: that is, the applicability of the Conversion rule does not depend on the syntax of the term being typed — it applies to every term. This complicates type-checking, i.e. determining whether $\Gamma \vdash M : \sigma$ holds. However, we can at least exploit the fact

that the β-reduction for types is strongly-normalising (as in the simply-typed lambda calculus). It is thus decidable whether β-equivalence holds by simply computing the (unique) normal forms of the two types and comparing them.

Since we have moved from unordered to ordered contexts, we need to add rules for manipulating the context, namely the following weakening rules:

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

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

5.1 Examples

We can define a type constructor for lists:

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

We want this to be well-kinded.

$$\dfrac{\dfrac{\dfrac{}{\beta : * \vdash \beta : *}}{\beta : *, \alpha : * \vdash \beta : *} \qquad \dfrac{\dfrac{}{\beta : *, \alpha : * \vdash \alpha : *}}{\beta : *, \alpha : * \vdash \alpha \to \alpha : *}}{\beta : *, \alpha : * \vdash \beta \to \alpha \to \alpha : *} \qquad \dfrac{}{\beta : *, \alpha : * \vdash \alpha : *}$$

$$\dfrac{\dfrac{}{\beta : *, \alpha : * \vdash \alpha : *} \qquad \dfrac{\beta : *, \alpha : * \vdash \beta \to \alpha \to \alpha : * \qquad \beta : *, \alpha : * \vdash \alpha : *}{\beta : *, \alpha : * \vdash (\beta \to \alpha \to \alpha) \to \alpha : *}}{\beta : *, \alpha : * \vdash \alpha \to (\beta \to \alpha \to \alpha) \to \alpha : *}$$

$$\frac{\beta : *, \alpha : * \vdash \alpha \to (\beta \to \alpha \to \alpha) \to \alpha : *}{\beta : * \vdash \forall\alpha : *. \alpha \to (\beta \to \alpha \to \alpha) \to \alpha : *}$$

$$\frac{\beta : * \vdash \forall\alpha : *. \alpha \to (\beta \to \alpha \to \alpha) \to \alpha : *}{\vdash \lambda\beta : *. \forall\alpha : *. \alpha \to (\beta \to \alpha \to \alpha) \to \alpha : * \to *}$$

A common operation on lists is to apply a function on each element of the list. This function is typically called map, and has the following type signature:

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

We can define this function as follows:

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

Let's deconstruct this definition a little bit. From the type signature, we can see that map takes two type parameters ($\alpha : *, \beta : *$) a function parameter ($f : \alpha \to \beta$) and a list parameter ($l : \mathbf{list}\ \alpha$). Here, we think of **list** as standing for its definition above, so we have[1]

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

[1] Remember that in $=_\beta$, the β refers to β-equivalence, not to any variable named β.

Now map has to produce a result of type **list** β, but the CONVERSION rule means that it can produce a result of any type that is β-*convertible* to **list** β. In particular, the result can be of type $\forall\gamma.\,\gamma \to (\beta \to \gamma \to \gamma) \to \gamma$. That is, a function that takes a type $(\gamma : *)$, a handler for the nil case $(n : \gamma)$ and a handler for the cons case $(c : \beta \to \gamma \to \gamma)$ and returns a result of type γ. We can think of defining our result by primitive recursion over the lists:

$$h(\mathsf{nil}) = n$$
$$h(\mathsf{cons}\ x\ l') = c(fx)(hl')$$

That is for nil we return the handler for nil, n; and for cons x l' we call the handler c with fx and the result of mapping over the rest of the list. The Church encoding means that $hl = l\gamma n(\lambda x : \alpha.\, c(fx))$. This then gives us the body of our function.

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 $* \to *$:

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

It is perfectly possible to nest type constructors, for instance: **list** (**list bool**); **tree** (**list nat**). So we can also compose type constructors: $\lambda\alpha : *.\,\mathbf{list}\ (\mathbf{list}\ \alpha)$. Given map-like functions on two type constructors, we can compose them to obtain a map-like function on the composition of the constructors:

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

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

Let us check that this term is well-typed. Let

$$\Gamma_0 = \mathbf{A} : * \to *, \mathsf{mapA} : \mathbf{Map}\ \mathbf{A}, \mathbf{B} : * \to *, \mathsf{mapB} : \mathbf{Map}\ \mathbf{B}$$
$$\Gamma_1 = \Gamma_0, \alpha : *, \beta : *, f : \alpha \to \beta$$

Figure 5.1 shows the key part of the typing derivation. The derivation has been cut off where subderivations follow from weakening and START. Derivations that types have kind $*$, such as the following, have been omitted. Also omitted are premisses that terms are β-equivalent.

$$\dfrac{\dfrac{\Gamma_1, \gamma : * \vdash \mathbf{A} : * \to * \qquad \dfrac{\Gamma_1, \gamma : * \vdash \mathbf{B} : * \to * \qquad \Gamma_1, \gamma : * \vdash \gamma : *}{\Gamma_1, \gamma : * \vdash \mathbf{B}\ \gamma : *}}{\Gamma_1, \gamma : * \vdash \mathbf{A}\ (\mathbf{B}\ \gamma) : *}}{\dfrac{\Gamma_1 \vdash \lambda\gamma : *.\, \mathbf{A}\ (\mathbf{B}\ \gamma) : * \to * \qquad \Gamma_1 \vdash \beta : *}{\Gamma_1 \vdash (\lambda\gamma : *.\, \mathbf{A}\ (\mathbf{B}\ \gamma))\beta : *}}$$

5.2 Logic

Recall that in System F we defined $\phi \vee \psi = \forall\theta.\, (\phi \to \theta) \to (\psi \to \theta) \to \theta$. In F_ω, we can promote $\vee$ to a function at the type level:

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

We can similarly define the other connectives of propositional logic as type-level functions.

When we are using F_ω as a logic, we can think of $*$ as being the "type of propositions". Note, however, that $*$ is not a type, but a kind.

5.3 A Collapsed Presentation

The version of System F_ω that we have seen had three distinct syntactic categories: terms, types and kinds. Some of the syntax of the language is duplicated between these levels (e.g. λ abstraction at both term and type levels, and $\to$ at both type and kind levels). This also leads to duplication in the typing rules.

An alternative presentation is to combine the syntactic categories and use the types to determine whether a term represents as base term, a type or a kind. Our terms are then:

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

Here, x ranges over syntactic variables. (We do not distinguish term variables and type variables, although we may suggestively use x, y, z for term variables and α, β, γ for type variables.) Moreover, $\mathbf{s} \in \{*, \Box\}$ ranges over the set of sorts (constants) which include the familiar *sort of types* $*$ and a new *sort of kinds* $\Box$.

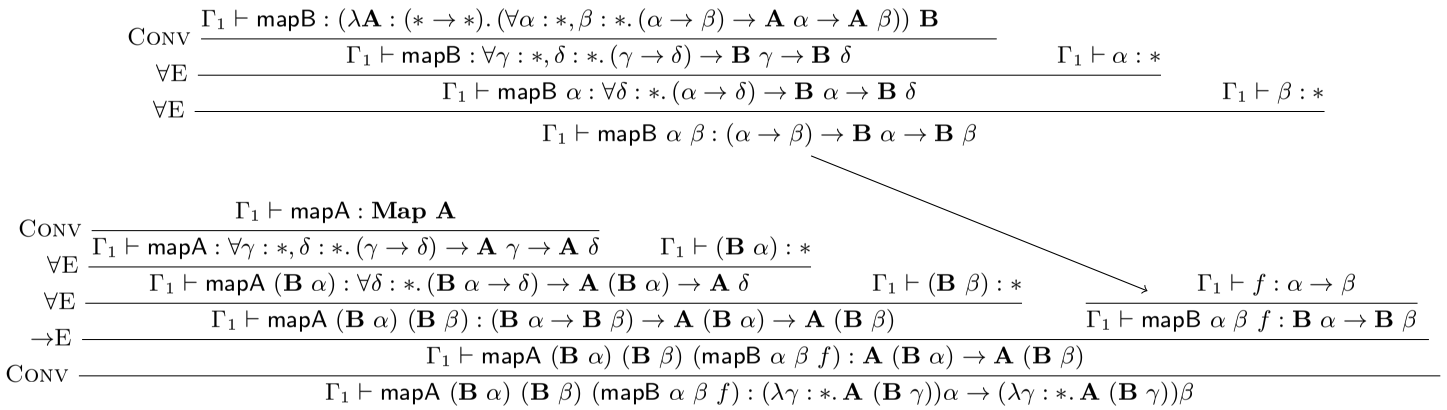

Figure 1: Partial deriviation of mapComp.

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

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

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

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

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

5.4 Exercises

Exercise 18. Give a derivation that the following type has kind $*$:

$$\forall \alpha : *. (\alpha \to \alpha) \to (\lambda F : * \to *. \forall \beta : *. F\beta \to F\beta)(\lambda \gamma : *. \alpha)$$

Give a derivation for the following typing judgement in F_ω:

$$\vdash \lambda \alpha : *, g : \alpha \to \alpha, \beta : *. g \\ : \forall \alpha : *. (\alpha \to \alpha) \to (\lambda F : * \to *. \forall \beta : *. F\beta \to F\beta)(\lambda \gamma : *. \alpha)$$

Compare derivations in the stratified presentation and the collapsed presentation.

Exercise 19. Let $\mathbf{A} : * \to *$. We call $\mathbf{A}$ *pointed* if there is a function $f : \forall \alpha. \alpha \to \mathbf{A}\alpha$. Define $\mathbf{Pointed} := \lambda \mathbf{A} : * \to *. \forall \alpha. \alpha \to \mathbf{A}\alpha$. Find a term of type $\mathbf{Pointed\ list}$. Find a term of type

$$\forall \mathbf{A} : * \to *. \mathbf{Pointed\ A} \to \forall \mathbf{B} : * \to *. \mathbf{Pointed\ B} \to \mathbf{Pointed}\ (\lambda \alpha : *. \mathbf{A}(\mathbf{B}\alpha))$$

Exercise 20. Recalling the definition from System F, $\bot = \forall \alpha : *. \alpha$. We can define negation as $\neg = \lambda \alpha : *. \alpha \to \bot$. Find a term M such that

$$\phi : *, \psi : * \vdash M : (\phi \to \psi) \to (\neg\psi \to \neg\phi)$$

Exercise 21. Following Exercise 17, define a conjunction term $\wedge$ such that $\vdash \wedge : * \to * \to *$, and which satisfies the appropriate axioms. Find a term M such that

$$\phi : * \vdash M : \neg(\phi \wedge \neg\phi)$$

(where $A \wedge B$ here is just syntactic sugar for $\wedge AB$).

6 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.

Following this line of thinking, it makes sense to add types parametrised by terms to our calculus — so-called *dependent types*. In doing so, we obtain the Calculus of Constructions (CoC), introduced by Coquand and Huet in 1985.

To present CoC, we will not syntactically differentiate between terms at the term, type and kind levels, following the collapsed presentation of F_ω. 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 $x \notin \mathrm{FV}(N)$.

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

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

$$\forall\ \textsc{elimination}\ \frac{\Gamma \vdash M : \forall x : \rho.\, \sigma \qquad \Gamma \vdash N : \rho}{\Gamma \vdash MN : \sigma[N/x]} \qquad \textsc{Conversion}\ \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma' : \mathbf{s} \qquad \sigma =_\beta \sigma'}{\Gamma \vdash M : \sigma'}$$

Compared with F_ω, the key point is that the $\forall$ FORMATION rule has been generalised: before, it could only be that $\mathbf{s}_1 = \Box$ and $\mathbf{s}_2 = *$. With this rule, we can construct:

- 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).

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

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

  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.

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

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

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

$$\to\text{F}\ \frac{\Gamma \vdash \kappa_1 : \Box \qquad \Gamma \vdash \kappa_2 : \Box}{\Gamma \vdash \kappa_1 \to \kappa_2 : \Box}$$

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

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

6.1 Some Properties of CoC

The substitution lemma for CoC generalises the equivalent lemma for STLC somewhat. Since terms may occur in types, we have to substitute not only in the term, but also in the type and the context. Recall that contexts for CoC are ordered, and types may depend on variables occurring earlier in the context (but not later).

Lemma 4 (Substitution)**.** *Suppose that* $\Gamma, x : \sigma, \Delta \vdash M : \rho$ *and* $\Gamma \vdash N : \sigma$ *Then* $\Gamma, \Delta[N/x] \vdash M[N/x] : \rho[N/x]$.

Lemma 5. *No well-typed term in CoC contains* $\Box$ *as a subterm.*

Proof. By induction on typing derivations: no rule introduces such a subterm. □

Lemma 6. *If* $\Gamma \vdash M : \sigma$ *then either* $\sigma = \Box$ *or* $\Gamma \vdash \sigma : \mathbf{s}$.

Proof. By induction on the structure of the derivation of $\Gamma \vdash M : \sigma$.

- AXIOM: trivially $\sigma = \Box$.
- START, CONVERSION and $\forall$ INTRODUCTION: immediate from premisses.
- WEAKENING: immediate from the induction hypothesis.
- $\forall$ FORMATION: trivial since $\mathbf{s}_2 \in \{*, \Box\}$.
- $\forall$ ELIMINATION: We must have $M = M'N$ and $\sigma = \sigma'[N/x]$ with $\Gamma \vdash M' : \forall x : \rho.\, \sigma'$ and $\Gamma \vdash N : \rho$. By the inductive hypothesis, we have that $\Gamma \vdash \forall x : \rho.\, \sigma' : \mathbf{s}$. Since this judgement must be derived by $\forall$ FORMATION, it must be that $\Gamma, x : \rho \vdash \sigma' : \mathbf{s}$. By the substitution lemma, since $\Gamma \vdash N : \rho$, it follows that $\Gamma \vdash \sigma'[N/x] : \mathbf{s}$.

□

We state the following two important theorems without proof.

Theorem 3. *The Calculus of Constructions is strongly normalising (with respect to $\rightarrow_\beta$).*

Theorem 4. *Type-checking in the Calculus of Constructions is decidable.*

6.2 Dependent Types

One use of dependent types is to define more constrained types. For instance, we could have a dependent type $\mathbf{vector} : * \rightarrow \mathbf{nat} \rightarrow *$ such that $\mathbf{vector}\ \sigma\ n$ is a list of terms of type σ of length n. There are two constructors:

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

$$\mathbf{vector} = \lambda \sigma : *, n : \mathbf{nat}, v : \mathbf{nat} \rightarrow *.\, v\mathsf{0} \rightarrow (\forall n' : \mathbf{nat}.\, \sigma \rightarrow vn' \rightarrow v(\mathsf{S}n')) \rightarrow vn$$

6.3 Logic

In CoC, under the propositions-as-types interpretation, we can think of $*$ as the "type of propositions". A proposition is valid if it has a proof: i.e. the type has an inhabitant (a term of that type). With dependent types, we can now consider terms of type $\sigma \rightarrow *$, for any type $\sigma : *$. Such terms assign a truth value (i.e. proposition) to each inhabitant of the type σ. We can thus think of such terms as *predicates* over σ.

For example, we could have a predicate $\mathsf{Zero} : \mathbf{nat} \rightarrow *$. We would like $\mathsf{Zero}\ \mathsf{0}$ to be valid, but not $\mathsf{Zero}\ (\mathsf{S}n)$ for any n. We can give a Church-style encoding by considering it to be a parametrised inductive type with a single constructor $\mathsf{z0} : \mathsf{Zero}\ \mathsf{0}$.

$$\mathsf{Zero} = \lambda n : \mathbf{nat}.\, \forall Z : \mathbf{nat} \rightarrow *.\, Z\mathsf{0} \rightarrow Zn$$

One way of interpreting this is that a term of type $\mathsf{Zero}\ n$ transforms a proof that an arbitrary property Z holds for $\mathsf{0}$ to a proof that Z holds for n.

The choice of $\mathsf{0}$ and even of the type **nat** was essentially arbitrary, so we can generalise this to a definition of equality (or identity) at a type:

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

This definition is called Leibniz equality, as it was initially proposed by Gottfried Wilhelm Leibniz. For two terms to be Leibniz equal, there must be no predicate that distinguishes between them. Moreover it embodies an essential elimination principle for equality: if two things are equal then any property of one is a property of the other.[2] Clearly, if two terms are β-equivalent then they will be Leibniz equal.

[2]In the Calculus of Inductive Constructions (the type system of Coq), equality is not defined this way, but as an inductive type. This gives a stronger elimination principle (called Axiom J) that

To be a reasonable notion of equality, Id must be an equivalence relation: that is, it must be reflexive, symmetric and transitive. Leibniz equality satisfies all of these.

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):

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

6.3.1 Functional Extensionality

Functional extensionality is the principle that two functions f and g are equal if, for all x, $f(x) = g(x)$. This is a principle which holds in set theory, but which is not provable with Leibniz equality in CoC. That is, it is not possible to construct a term of the following type:

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

However, it is possible to assume the existence of such a term as an axiom without introducing inconsistency.

6.4 Models of Type Theory

We've stated that functional extensionality is independent of CoC: that is, it is not provable, but not inconsistent. How would we go about proving such a thing? We cannot really do so inside CoC itself, since we do not have a mechanism for describing provability. The answer is to consider models of the theory.

A model of a type theory interprets types, terms and judgements in a different theory, such as set theory or category theory. For instance, a model in set theory might interpret contexts Γ as sets and the judgement $\Gamma \vdash M : \sigma$ as a function from the interpretation of Γ to the interpretation of σ (in the context Γ).

We can show that axiom is independent by showing that there are models where the axiom is valid and models where it is not. For example, we could have a model where functions are represented by set-theoretic functions, and so functional extensionality is valid. On the other hand, we could have a model where there are more functions than those that interpret terms; in such a model there can be a predicate that distinguishes terms that are extensionally equivalent.

We will not talk about models in this course, but it is sometimes useful to think about them to develop intuition.

6.5 Exercises

Exercise 22. For each of the following, determine whether the typing judgement is valid in CoC. If so, determine for which pairs $(\mathbf{s}_1, \mathbf{s}_2)$ the $\forall$ FORMATION

rule is used:

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

1.

$$\vdash \lambda F : * \to *, u : (\forall \beta : *.\, F\ \beta \to \beta), \alpha : *, x : F\ \alpha.\, u \alpha x \\ : \forall F : * \to *.\, (\forall \beta : *.\, F\ \beta \to \beta) \to \forall \alpha : *.\, F\ \alpha \to \alpha$$

2.

$$\alpha : *, F : \alpha \to *, y : \alpha \vdash \lambda g : (\forall x : \alpha.\, Fx).\, gy : (\forall x : \alpha.\, Fx) \to Fy$$

3.

$$\alpha : *, x : \alpha \vdash x : \alpha$$

4.

$$\alpha : *, x : \alpha, F : \alpha \to * \vdash Fx : *$$

5.

$$\alpha : *, F : (* \to *) \to *, G : \alpha \to \alpha \vdash FG : *$$

Exercise 23. Recall the definition of Leibniz equality:

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

Show that equality is transitive. That is, find a term trans with

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

Recall the following definitions:

$$\begin{aligned}
\mathbf{nat} : * &= \forall \alpha : *.\, \alpha \to (\alpha \to \alpha) \to \alpha \\
0 : \mathbf{nat} &= \lambda \alpha : *, o : \alpha, s : \alpha.\, o \\
\mathsf{S} : \mathbf{nat} \to \mathbf{nat} &= \lambda n : \mathbf{nat}.\, \lambda \alpha : *, o : \alpha, s : \alpha \to \alpha.\, s(n \alpha o s) \\
\mathsf{plus} : \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} &= \lambda n : \mathbf{nat}, m : \mathbf{nat}.\, \lambda \alpha : *, o : \alpha, s : \alpha \to \alpha.\, n \alpha (m \alpha o s) s \\
\mathsf{Id} : \forall \alpha : *.\, \alpha \to \alpha \to * &= \lambda \alpha : *, x : \alpha, y : \alpha.\, \forall P : \alpha \to *.\, Px \to Py \\
\mathsf{refl} : \forall \alpha : *, x : \alpha.\, \mathsf{Id}\ \alpha\ x\ x &= \lambda \alpha : *, x : \alpha, P : \alpha \to *, H : Px.\, H
\end{aligned}$$

Exercise 24. Prove that

$$\forall x : \mathbf{nat}, y : \mathbf{nat}.\, \mathsf{Id}\ \mathbf{nat}\ (\mathsf{plus}\ (\mathsf{S}x)\ y)\ (\mathsf{S}\ (\mathsf{plus}\ x\ y))$$

That is, find a term that inhabits the above type.

Exercise 25. The Church encoding of natural numbers is sufficient to define (primitive) recursive functions, but not to prove inductive properties. For that we postulate (i.e. assume) an induction axiom:

$$\mathsf{natInd} : \forall P : \mathbf{nat} \to *.\, P0 \to (\forall n : \mathbf{nat}.\, Pn \to P(\mathsf{S}n)) \to \forall m : \mathbf{nat}.\, Pn$$

This allows us to show that property P holds for all m by showing:

1. P holds for $\mathsf{0}$
2. assuming that P holds for some n (the induction hypothesis) then P holds for $\mathsf{S}n$.

Using natInd, prove that

$$\forall x : \mathbf{nat}, y : \mathbf{nat}.\, \mathsf{Id}\ \mathbf{nat}\ (\mathsf{plus}\ x\ (\mathsf{S}\ y))\ (\mathsf{S}\ (\mathsf{plus}\ x\ y))$$

Hints:

- You do not need to unfold the definition of **nat**. If you use the results of the previous two exercises, then it should also not be necessary to unfold Id, although a more direct proof is possible if you do unfold Id.
- Use natInd with the predicate

$$\lambda n : \mathbf{nat}.\, \mathsf{Id}\ \mathbf{nat}\ (\mathsf{plus}\ n\ (\mathsf{S}y))\ (\mathsf{S}\ (\mathsf{plus}\ n\ y))$$