TYPE THEORETIC PRELIMINARIES

Abstract. Higher order type compilation needs to discuss several intermediate languages with interesting type structure. Most of those languages are built on top of F^ω, the polymorphic lambda calculus with type functions. The first lecture is devoted to understanding F^ω so that we may avoid discussing the same type theory over and over again later in the course.

1. System F

Before moving to our discussion of F^ω it's well worth reviewing plain System F, the polymorphic lambda calculus. System F is composed of two syntactic categories: types and terms.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \to \tau \mid \forall \alpha.\tau \\ e & ::= & x \mid \lambda x : \tau.e \mid e\ e \mid \Lambda \alpha.e \mid e[\tau] \end{array}$$

The defining feature of System F is the ability to abstract over types in the term language with $\Lambda\alpha.e$. This let's us write polymorphic code. For the classic example we have the identity function which works for all types

$$id \triangleq \Lambda\alpha.\lambda x : \alpha.x$$

Now we discuss the typing rules for System F. The first thing to note is that the typing rules discuss the context of known type and term variables. We call contexts Γ. The syntax for Γ is

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha : \mathtt{type}$$

There are two key judgments, first is that we have a well-formed type, $\Gamma \vdash \tau : \mathtt{type}$ and that a term has a type $\Gamma \vdash e : \tau$. The rules for System F are

$$\text{1A}\ \frac{\alpha : \mathtt{type} \in \Gamma}{\Gamma \vdash \alpha : \mathtt{type}} \qquad \text{1B}\ \frac{\Gamma \vdash \tau_1 : \mathtt{type} \qquad \Gamma \vdash \tau_1 : \mathtt{type}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathtt{type}} \qquad \text{1C}\ \frac{\Gamma, \alpha : \mathtt{type} \vdash \tau : \mathtt{type}}{\Gamma \vdash \forall \alpha.\tau : \mathtt{type}}$$

$$\text{1D}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash \tau : \mathtt{type}}{\Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2} \qquad \text{1E}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{1F}\ \frac{\Gamma, \alpha : \mathtt{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall \alpha.\tau} \qquad \text{1G}\ \frac{\Gamma \vdash e : \forall \alpha.\tau_2 \qquad \Gamma \vdash \tau_1 : \mathtt{type}}{\Gamma \vdash e[\tau_1] : [\tau_1/\alpha]\tau_2}$$

The thing to notice here is the symmetry between Λ and λ. Λ is a construct to abstract types at the term level, and it introduces polymorphic types which are instantiated via type application. λ is a construct to abstract terms at the term level, and it introduces arrow types which are eliminated via term application.

Neither, however, introduces the notion of abstraction into the *types* themselves. While System F let's us abstract over types at the term level, at the type level there's still no way to describe something like `'a list` in ML. In this case, `list` isn't even a "type" on its own; it doesn't make sense to talk about $e : list$. We want to add more structure to our types so that `list` is expressible, but we need to be sure to classify our types properly so to distinguish between things which can and can't be inhabited by terms.

2. On to F^ω

To work our way towards F^ω we start by making a small change to the syntax of System F, we add a notion of kinds. These classify types just as types classify terms. We'll only add one kind though: `type`. We replace $\Gamma \vdash e : \texttt{type}$ with the ostensibly more general form $\Gamma \vdash e : k$ where k is a kind. Of course, this is only cosmetic until we enrich our kind structure.

$$\begin{array}{lcl} k & ::= & \texttt{type} \\ c & ::= & \alpha \mid c \to c \mid \forall \alpha : k.c \\ e & ::= & x \mid \lambda x : c.e \mid e\ e \mid \Lambda \alpha : k.e \mid e[c] \\ \Gamma & ::= & \cdot \mid \Gamma, x : c \mid \Gamma, \alpha : k \end{array}$$

We have made two changes to our syntax, first of all we replaced `type` with a k and added annotations $\tau : k$ to the syntax of things quantifying over types. Next, we changed τ to c. c denotes the syntactic class of **constructors**, of which types τ are a part. The reason for this distinction is that types τ are understood to be classifiers of terms, but shortly we'll add constructs to c that are *uninhabited by terms*, so it's misleading to use τ. For now it's just alpha conversion. The reader is left to fix the typing rules to match the new syntax.

Now to get F^ω we make two critical changes

$$\begin{array}{lcl} k & ::= & \texttt{type} \mid k \to k \\ c & ::= & \alpha \mid c \to c \mid \forall \alpha : k.c \mid \lambda \alpha : k.c \mid c\ c \\ e & ::= & x \mid \lambda x : c.e \mid e\ e \mid \Lambda \alpha : k.e \mid e[c] \\ \Gamma & ::= & \cdot \mid \Gamma, x : c \mid \Gamma, \alpha : k \end{array}$$

We've now added a new kind, $k \to k$ and added two new types we intend to use as the introduction and elimination forms for this kind: $\lambda \alpha : k.c$ and $c\ c$. If you cover e and look at this syntax you'll notice that we've just replicated the simply typed lambda calculus, but now in the type system of another language.

The typing rules now become more subtle because $\Gamma \vdash c : k$ is interesting. Before it was basically just a check that all variables were bound, now it's slightly more subtle.

We begin by writing the rules for the judgment $\Gamma \vdash e : c$, these are largely unchanged.

$$\text{2A}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{2B}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad \Gamma \vdash \tau : \texttt{type}}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2} \qquad \text{2C}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{2D}\ \frac{\Gamma, \alpha : k \vdash e : \tau}{\Gamma \vdash \Lambda \alpha : k . e : \forall \alpha : k . \tau} \qquad \text{2E}\ \frac{\Gamma \vdash e : \forall \alpha : k . \tau_2 \qquad \Gamma \vdash \tau_1 : k}{\Gamma \vdash e[\tau_1] : [\tau_1/\alpha]\tau_2}$$

As a philosophical note, we have a question to ask, should it be possible to show

$$x : \lambda \alpha : \texttt{type} . \alpha \vdash x : \lambda \alpha : \texttt{type} . \alpha$$

Clearly this is gibberish, since we only care about $\Gamma \vdash e : c$ when for all $x : c \in \Gamma$, $\Gamma \vdash c : \texttt{type}$ (more generally whenever Γ is well formed) but there are two ways to deal with this in our type theory. We can enforce this as a presupposition of the judgment, we only care about the judgment when the context is well-formed so that all term variable have types with the kind `type`. We could also structure our judgment so that it's impossible to demonstrate $\Gamma \vdash e : c$ without making Γ *ok* evident. For our purposes the former is sufficient, but for many developments the latter is more convenient.

Next we present the typing (kinding?) rules for types, which are basically those of the simply typed lambda calculus.

$$\text{2F}\ \frac{x : k \in \Gamma}{\Gamma \vdash x : k} \qquad \text{2G}\ \frac{\Gamma, x : k_1 \vdash c : k_2}{\Gamma \vdash \lambda x : k_1 . c : k_1 \to k_2} \qquad \text{2H}\ \frac{\Gamma \vdash c_1 : k_1 \to k_2 \qquad \Gamma \vdash c_2 : k_2}{\Gamma \vdash c_1\ c_2 : k_2}$$

$$\text{2I}\ \frac{\Gamma, \alpha : k_1 \vdash c : \texttt{type}}{\Gamma \vdash \forall \alpha : k . c : \texttt{type}} \qquad \text{2J}\ \frac{\Gamma \vdash c_1 : \texttt{type} \qquad \Gamma \vdash c_2 : \texttt{type}}{\Gamma \vdash c_1 \to c_2 : \texttt{type}}$$

And thus we have the typing rules for F^ω. Note that we can omit the well-formedness checks on kinds because all kinds are trivially well-formed.

A possibile point of confusion is the difference between $\forall \alpha : k.\ c$ and $\lambda \alpha : k . c$. The difference is as follows. $\forall \alpha : k.\ c$ has kind `type`; it is the type of polymorphic terms, such as the polymorphic identity. $\lambda \alpha : k.\ c$ has kind $\texttt{type} \to \texttt{type}$; it is the kind of types with a single type parameter, such as `list`. In particular, $\lambda \alpha : k.\ c$ is not inhabited by any terms.

There is an oversight in the system as presented thus far. Suppose we have

$$f : \forall \gamma : \texttt{type} \to \texttt{type} .\ \gamma\ \mathit{int} \to \tau$$

What's the type of

$$f\ [\lambda \alpha : \texttt{type} .\ \alpha]\ 12$$

Answer: it's ill-typed! $f\ [...] : (\lambda \alpha . \alpha)\ \mathit{int} \to \tau$, and we don't know that the domain is the same as *int*. We need to introduce a new rule of the form

$$\text{2K}\ \frac{\Gamma \vdash c_1 \equiv c_2 : \texttt{type} \qquad \Gamma \vdash e : c_1}{\Gamma \vdash e : c_2}$$

so that we can freely "reduce" types in order to perform typechecking. This means we have to specify an equivalence $\equiv$ on types.

3. Definitional equality

So the task is to define $\equiv$, called definitional equality. That is, we are introducting a new judgement form $\Gamma \vdash c_1 \equiv c_2 : k$, which is read as "$\Gamma$ derives that c_1 and c_2 are equivalent at kind k." We specify first that it is an equivalence relation:

$$\text{3A}\ \frac{}{\Gamma \vdash c \equiv c : k} \qquad \text{3B}\ \frac{\Gamma \vdash c' \equiv c : k}{\Gamma \vdash c \equiv c' : k} \qquad \text{3C}\ \frac{\Gamma \vdash c_1 \equiv c_2 : k \quad \Gamma \vdash c_2 \equiv c_3 : k}{\Gamma \vdash c_1 \equiv c_3 : k}$$

Brief aside: in F^ω, kinds classify types into disjoint sets, so given constructors can only possibly be "equivalent at" the kind they both belong to. Hence at this point the notation $\Gamma \vdash c_1 \equiv c_2 : k$ is somewhat superfluous, but we introduce it to maintain extensibility with later material. In particular, we will eventually study calculi in which given constructors may be equivalent at one kind and unequivalent at another. This will also turn out to be useful shortly when we describe an algorithm for deciding constructor equivalence in F^ω.

Returning to our specification of definitional equality.

$$\text{3D}\ \frac{\Gamma \vdash c_1 \equiv c_1' : k_1 \to k_2 \quad \Gamma \vdash c_2 \equiv c_2' : k_2}{\Gamma \vdash c_1 c_2 \equiv c_1' c_2' : k_2} \qquad \text{3E}\ \frac{\Gamma, \alpha : k_1 \vdash c \equiv c' : k_2}{\Gamma \vdash (\lambda \alpha : k_1.\ c \equiv \lambda \alpha : k_1.\ c') : k_1 \to k_2}$$

$$\text{3F}\ \frac{\Gamma \vdash \tau_1 \equiv \tau_2 : \mathtt{type} \qquad \Gamma \vdash \tau_1' \equiv \tau_2' : \mathtt{type}}{\Gamma \vdash (\tau_1 \to \tau_2) \equiv (\tau_1' \to \tau_2') : \mathtt{type}} \qquad \text{3G}\ \frac{\Gamma, \alpha : k \vdash \tau \equiv \tau' : \mathtt{type}}{\Gamma \vdash (\forall \alpha : k.\ \tau \equiv \forall \alpha : k.\ \tau') : \mathtt{type}}$$

$$\text{3H}\ \frac{\Gamma \vdash c_2 : k \qquad \Gamma, \alpha : k \vdash c_1 : k'}{\Gamma \vdash (\lambda \alpha : k.\ c_1) c_2 \equiv [c_2/\alpha] c_1 : k'}$$

$$\text{3I}\ \frac{\Gamma \vdash c : k_1 \to k_2 \qquad \Gamma \vdash c' : k_1 \to k_2 \qquad \Gamma, \alpha : k_1 \vdash c\,\alpha \equiv c'\,\alpha : k_2}{\Gamma \vdash c \equiv c' : k_1 \to k_2}$$

Our example from before works as we should hope (given some notion of primitive type *int*), with $\cdot \vdash f\ [\lambda \alpha : \mathtt{type}.\ \alpha]\ 12 : \tau$ now being derivable.

4. Souped-up F^ω

Now we will introduce a couple of modest extensions that are not part of F^ω as typically presented. The reason we want to do this is because when we develop the elaboration of modules into F^ω it turns out we will need **product kinds** to explain them.

$$\begin{array}{rcl} k & ::= & \cdots \mid k * k \\ c & ::= & \cdots \mid \langle c, c \rangle \mid \pi_1\ c \mid \pi_2\ c \end{array}$$

These work as pairs of kinds. The kinding rules are as follows.

$$\text{4A}\ \frac{\Gamma \vdash c_1 : k_1 \quad \Gamma \vdash c_2 : k_2}{\Gamma \vdash \langle c_1, c_2 \rangle : k_1 * k_2} \qquad \text{4B}\ \frac{\Gamma \vdash c : k_1 * k_2}{\Gamma \vdash \pi_1\ c : k_1} \qquad \text{4C}\ \frac{\Gamma \vdash c : k_1 * k_2}{\Gamma \vdash \pi_2\ c : k_2}$$

We also need to extend definitional equivalence to account for the new constructs.

$$\text{4D}\ \frac{\Gamma \vdash c_1 \equiv c_1' : k_1 \quad \Gamma \vdash c_2 \equiv c_2' : k_2}{\Gamma \vdash \langle c_1, c_2 \rangle \equiv \langle c_1', c_2' \rangle : k_1 * k_2} \qquad \text{4E}\ \frac{\Gamma \vdash c \equiv c' : k_1 * k_2}{\Gamma \vdash \pi_1\ c \equiv \pi_1\ c' : k_1}$$

$$\text{4F}\ \frac{\Gamma \vdash c \equiv c' : k_1 * k_2}{\Gamma \vdash \pi_2\ c \equiv \pi_2\ c' : k_2} \qquad \text{4G}\ \frac{\Gamma \vdash c_1 : k_1 \quad \Gamma \vdash c_2 : k_2}{\Gamma \vdash \pi_1 \langle c_1, c_2 \rangle \equiv c_1} \qquad \text{4H}\ \frac{\Gamma \vdash c_1 : k_1 \quad \Gamma \vdash c_2 : k_2}{\Gamma \vdash \pi_2 \langle c_1, c_2 \rangle \equiv c_2}$$

$$\text{4I}\ \frac{\Gamma \vdash \pi_1\ c \equiv \pi_1 c' : k_1 \quad \Gamma \vdash \pi_2\ c \equiv \pi_2\ c' : k_2}{\Gamma \vdash c \equiv c' : k_1 * k_2}$$