PRELIMINARIES FOR COMPILATION

Abstract. HOT 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 unique 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

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

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

$$\frac{\alpha : \texttt{type} \in \Gamma}{\Gamma \vdash \alpha : \texttt{type}} \qquad \frac{\Gamma \vdash \tau_1 : \texttt{type} \qquad \Gamma \vdash \tau_1 : \texttt{type}}{\Gamma \vdash \tau_1 \to \tau_2 : \texttt{type}} \qquad \frac{\Gamma, \alpha : \texttt{type} \vdash \tau : \texttt{type}}{\Gamma \vdash \forall \alpha.\tau : \texttt{type}}$$

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

$$\frac{\Gamma, \alpha : \texttt{type} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall \alpha.\tau} \qquad \frac{\Gamma \vdash e : \forall \alpha.\tau_2 \qquad \Gamma \vdash \tau_1 : \texttt{type}}{\Gamma \vdash e[\tau_1] : [\tau_1/\alpha]\tau_2}$$

The thing to notice here is the symmetry between Λ and λ and what-not. We now begin the process of adding a notion of abstraction into our *types*. 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 SML. In this case, `list` isn't even a "type" on its own, it doesn't make sense to talk about $e : \mathit{list}$. 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. The reason is that types (and therefore things represented by τ) ought to classify terms, shortly however, we'll add occupants to c that do no such thing so it's a misnomer 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.

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

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

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. In HOT compilation the former is sufficient, but for many developments the latter is more convenient.

Question: the former seems more closely related to Martin-Löf's formulation of type theory, we only care about the meanings of judgments when we have assume our presuppositions to be evident. The latter seems closer to CTT where presuppositions become evident in the course of proving a judgment, is this intuition correct?

In any case for now it's just a digression. Next we present the typing (kinding?) rules for types, which are basically those of the simply typed lambda calculus.

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

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

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.

This lecture finished with an oversight in this system. 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

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

so that we can freely "reduce" types in order to typecheck things. This means we have to specify and equivalence on types though.