CS 4110 – Programming Languages and Logics
Lecture #31: Higher-Kinded Types

In this lecture, we'll explore a kind of generalization of the polymorphic λ-calculus that makes the type system even more powerful. While that type system seemed to *roughly* add λ-calculus-like features at the type level, this extension will complete the job: we'll have a complete "copy" of the term-level language in the type system, including arbitrary functions, variables, and applications.

1 Functions on Types

In the lecture on System F, we saw how to encode sums and products using polymorphism. The nifty conclusion is that, while we originally added these data types to our language as one-off extensions, that wasn't really necessary. You can write terms for the constructors and destructors that work for *any* underlying type. For example, you saw a constructor pair and destructors pi_1 and pi_2 that each worked for any two types you might want to pack together into a pair. The expression pair $[\mathbf{int}]$ $[\mathbf{bool}]$, for example, produces a function that can take a number and a Boolean and construct a pair from them, as in pair $[\mathbf{int}]$ $[\mathbf{bool}]$ 4 true.

One way of seeing this is that you used polymorphism to write your own *functions from types to terms*. The "function" pair, when applied to two *types*, produced a *term* (i.e., a program) that worked as a specialized pair constructor. This is nice because we get to think of pair itself, before giving it specific types, as a self-contained, first-class entity.

However, we did not get the same level of parameterization for the *type* of pairs. The pair of an **int** and a **bool** in our encoding was written $\forall\gamma.(\mathbf{int} \to \mathbf{bool} \to \gamma) \to \gamma$, but there's no succinct way to summarize the *generic* type for any pair. What would it take to write self-contained, first-class entity called, say, capital Pair such that invoking it as Pair **int bool** expanded out to the type above? If we had something like this, we could stop relying on human intuition to see something like **int**×**bool** and replace it with our encoding for that type; the rules for producing complex types out of simpler ones would be formalized as *part of the language*.

That's the motivation for augmenting our language with *functions from types to types*. We'll be able define Pair as a self-contained type operator. This way, you will be able to write the complete type of the pair constructor as

$$\forall\alpha.\forall\beta.\alpha \to \beta \to (\mathsf{Pair}\ \alpha\ \beta)$$

which you have to admit is nicer to read than

$$\forall\alpha.\forall\beta.\alpha \to \beta \to \forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma$$

We'll introduce a language where we can write the type operator Pair as a type-level function using familiar λ-calculus constructs:

$$\mathsf{Pair} \triangleq \lambda\alpha : \mathsf{type}.\ \lambda\beta : \mathsf{type}.\ \forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma$$

where we have a new λ construct for binding type variables.

2 Adding Kinds in λ_ω

We'll define an extension to the simply-typed λ-calculus, $\lambda^{\rightarrow}$, that adds the ability to write type operators. The language will be called λ_ω. (If you combine type operators and polymorphism, you get a language called System F_ω.)

In λ_ω, the grammar for expressions is the same as in the basic $\lambda^{\rightarrow}$. We'll make the language of types more complicated. Instead of just base types and abstractions, we'll add type variables, type abstractions, and type applications:

$$\tau ::= b \mid \tau_1 \to \tau_2 \mid \alpha \mid \lambda\alpha : \kappa.\, \tau \mid \tau_1\ \tau_2$$

Notice that our type abstractions, just like ordinary term-level abstractions, need annotations. These indicate the "type of the type" for the argument to the type operator. The usual word for "types of types" is *kind*, which we'll denote with a new metavariable κ:

$$\kappa ::= \mathsf{type} \mid \kappa_1 \Rightarrow \kappa_2$$

The $\Rightarrow$ symbol denotes the kind of a type-level function.

Now is a good time to pause and think deeply about the fact that this grammar for kinds looks like our *old* grammar for types in $\lambda^{\rightarrow}$: there is a base kind, type, and functions on those kinds. We have essentially described our language of types using a copy of $\lambda^{\rightarrow}$. It's healthy to wonder, then, why this shouldn't continue on forever: why not write types for types of types, and then types for those, and so on into infinity? It's possible to do that, and it leads to a realm of research called *pure type systems*, but going beyond two levels of types does not seem to be very useful for actual programming.

For what it's worth, higher-order type operators, which are type-level functions take other type-level functions as arguments, are also not nearly as useful as more basic type operators. So while it is easy to imagine types of the kind $\mathsf{type} \Rightarrow \mathsf{type}$ and even $\mathsf{type} \Rightarrow \mathsf{type} \Rightarrow \mathsf{type}$, it's harder to imagine useful examples with the kind $(\mathsf{type} \Rightarrow \mathsf{type}) \Rightarrow \mathsf{type}$.

2.1 Typing Rules

Let's define the typing rules for λ_ω. The rules are going to look similar to our type system for System F, where we added a second context Δ to keep track of type variables. In that language, however, Δ was a set, not a mapping. In retrospect, this worked because all type variables referred to plain old types, so there was no need to distinguish between kinds of types. Now, Δ will be a partial function from type variables to kinds κ. The basic rules for variables, abstractions, and application are:

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

Aside from adding Δ to the judgment, these rules look mostly the same as the rules for the simply-typed λ-calculus. As in System F, the variable and function rules need an extra premise that ensures that the types are well-formed—and, here, we make it explicit that they must have the kind type. Term-level variables are not allowed to have types whose kind is $\mathsf{type} \Rightarrow \mathsf{type}$, for example.

We will also add one more rule to capture *type equivalence*. The idea is that we want to let type expressions "evaluate" to produce concrete types: for example, the type expression Pair **int bool** should be able to "expand out" to its full polymorphic type so we can use it in computations. We will define a new relation, $\tau_1 \equiv \tau_2$, to determine equivalent types. Then, we add a typing rule that lets expressions take on any equivalent type:

$$\frac{\Delta; \Gamma \vdash e : \tau \qquad \tau \equiv \tau' \qquad \Delta \vdash \tau' : \mathsf{type}}{\Delta; \Gamma \vdash e : \tau'}$$

This rule says that if you have already derived a type τ for an expression, and you know τ is equivalent to another type τ', you can also say that the expression has type τ'.

2.2 Kinding Rules

Leaving aside type equivalence for a moment, we need a way to determine whether a type τ has kind κ. That is, we need inference rules to define the judgment $\Delta \vdash \tau : \kappa$ that we used in the typing rules above. In System F, we defined a similar judgment that just checked that all the type variables used in a type expression were bound by $\forall$s. In λ_ω, the kinding rules define the meaning of type abstraction and type application:

$$\frac{}{\Delta, \alpha : \kappa \vdash \alpha : \kappa} \qquad \frac{\Delta, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Delta \vdash \lambda\alpha : \kappa_1.\, \tau : \kappa_1 \Rightarrow \kappa_2} \qquad \frac{\Delta \vdash \tau_1 : \kappa_1 \Rightarrow \kappa_2 \qquad \Delta \vdash \tau_2 : \kappa_1}{\Delta \vdash \tau_1\ \tau_2 : \kappa_2}$$

These kinding rules for type variables, type abstraction, and type application are identical to the typing rules for terms in $\lambda^{\rightarrow}$. We also need two more kinding rule for dealing with our only "base kind," i.e., type. We will have one axiom that says that the base types have kind type, and another rule that functions on types are also types:

$$\frac{}{\Delta \vdash b : \mathsf{type}} \qquad \frac{\Delta \vdash \tau_1 : \mathsf{type} \qquad \Delta \vdash \tau_2 : \mathsf{type}}{\Delta \vdash \tau_1 \rightarrow \tau_2 : \mathsf{type}}$$

2.3 Type Equivalence

Finally, we need to define that $\equiv$ relation we alluded to above. First, we will make the operator reflexive, symmetric, and transitive:

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau_1 \equiv \tau_2}{\tau_2 \equiv \tau_1} \qquad \frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}$$

Next, we define equivalences for function types, type abstractions, and type applications that just "recurse through" these structures:

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \rightarrow \tau_2 \equiv \tau_1' \rightarrow \tau_2'} \qquad \frac{\tau \equiv \tau'}{\lambda x : \kappa.\, \tau \equiv \lambda x : \kappa.\, \tau'} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1\ \tau_2 \equiv \tau_1'\ \tau_2'}$$

Finally, we need one more rule that is the equivalent of β-reduction on types:

$$\frac{}{(\lambda\alpha : \kappa.\, \tau_1)\ \tau_2 \equiv \tau_1\{\tau_2/\alpha\}}$$

With these rules, we have, for example, that $(\lambda\alpha : \mathsf{type}.\, \mathbf{int})\ \mathbf{bool} \equiv \mathbf{int}$.