1 Introduction

Why Type Theory? If you want to work with programming languages, you want to be sure that the languages you define are useful. The bare minimum for a language being useful is that it works in all the cases it should—what does that mean?

It means that if I have a program that the compiler says is well-typed, then all the errors that happen should be my fault. We should never run into a case when the compiler says "your program is right" but then it says "actually I couldn't figure out what to do here." This would mean, that while we were writing the compiler, we forgot some cases; that's incredibly annoying for users, so to avoid this, we can prove that we didn't miss any cases.

To prove that this won't happen, we need a mathematical representation of our language and its type system. Then we can prove type safety: "Well-typed programs don't get stuck."

The simplest language we can see these ideas in is the simply-typed λ-calculus.

Note that, throughout this document, it is only allowed to have a single binding of any name in a context, so something like $x : \tau_1, x : \tau_2$ is **not** allowed, **even if** $\tau_1 = \tau_2$. This is simply to reduce the need to declare that there is no variable shadowing in any of the systems we discuss.

2 The Simply-Typed λ-Calculus

2.1 Syntax

$$\begin{array}{lcl} \mathrm{V} & ::= & \mathrm{true} \mid \mathrm{false} \mid x \mid \lambda x : \tau . E \\ \mathrm{E} & ::= & V \mid E\ E \\ \tau & ::= & \mathrm{bool} \mid \tau \to \tau \end{array}$$

Examples

1.

2.2 Dynamics

$$\frac{e_1 \to e_1'}{e_1 e_2 \to e_1' e_2}\ \textsc{E-App} \qquad\qquad \frac{}{(\lambda x : \tau . e_1) e_2 \to e_1[e_2/x]}\ \textsc{E-}\beta\textsc{-reduce}$$

2.3 Statics

Type Contexts Syntax

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, x : \tau$$

Typing Rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{}{\Gamma \vdash \mathrm{true} : \mathrm{bool}}\ \textsc{B-True} \qquad \frac{}{\Gamma \vdash \mathrm{false} : \mathrm{bool}}\ \textsc{B-False} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2}\ \textsc{App}$$

2.4 Type Safety

We want to prove that our type system actually works—if a term is well-typed, we don't get stuck. Usually, we break up proofs of type safety into two parts: **Progess**, that a well-typed term is a value or can be evaluated one more step, and **Preservation**, that after taking a step, the resulting term is still well-typed.

Theorem 1 (Progress). *If $\cdot \vdash e : \tau$, then either e is a value or $e \to e'$ for some e'.*

Proof. by induction on the typing derivation of $\cdot \vdash e : \tau$.

Case: Var Contradiction, we assumed that e is closed (but also e is a value).

Case: B-True Then e is a value.

Case: B-False Then e is a value.

Case: Abs Then $e = \lambda x : \tau.e_1$ and e is a value.

Case: App Then $e = e_1 e_2$ and $\cdot \vdash e_1 e_2 : \tau_2$ and $\cdot \vdash e_1 : \tau_1 \to \tau_2$ and $\cdot \vdash e_2 : \tau_1$ by inversion of the typing relation.

By induction, either e_1 is a value or $e_1 \to e_1'$.

If e_1 is a value, then it must be $\lambda x : \tau.b$ because it is a closed term and the only typing rule that applies is Abs. Then $e_1 e_2 = (\lambda x : \tau.b)e_2 \to b[e_2/x]$.

Otherwise, $e_1 \to e_1'$, so $e_1 e_2 \to e_1' e_2$. □

Note that preservation doesn't always require e and e' have the **same** type, but in this case we can guarantee it. We'll need one tool to prove preservation:

Lemma 1 (Substitution Preservation). *If $\Gamma, x : \tau_1 \vdash e : \tau_2$ and $e' : \tau_1$, then $\Gamma \vdash e[e'/x] : \tau_2$.*

Proof. by induction on the typing derivation. □

Theorem 2 (Preservation). *If $\Gamma \vdash e : \tau$, and $e \to e'$, then $\Gamma \vdash e' : \tau$.*

Proof. by induction on the derivation of $\Gamma \vdash e : \tau$.

Case: Var Contradiction, e is a value and so we don't have $e \to e'$ for any e'.

Case: B-True Contradiction, e is a value.

Case: B-False Contradiction, e is a value.

Case: Abs Again, contradiction because $e = \lambda x : \tau.e_1$ is a value.

Case: App Then $e = e_1 e_2$ and $\Gamma \vdash e_1 e_2 : \tau_2$ and $\Gamma \vdash e_1 : \tau_1 \to \tau_2$ and $\Gamma \vdash e_2 : \tau_1$ by inversion of the typing relation. There are two subcases to consider, depending on how we derived $e \to e'$:

E-App In this case, $e_1 \to e_1'$, and we already know that $\Gamma \vdash e_1 : \tau_1 \to \tau_2$, so by the inductive hypothesis, we know $\Gamma \vdash e_1' : \tau_1 \to \tau_2$. Then we can apply the App rule to get $\Gamma \vdash e_1' e_2 : \tau_2$.

E-β-reduce Then $e_1 = \lambda x : \tau_1 . b$, and so we must have $\Gamma \vdash \lambda x : \tau_1 . b : \tau_1 \to \tau_2$, which we must have obtained using Abs. Therefore, by inversion we have $\Gamma, x : \tau_1 \vdash b : \tau_2$.

Also, $e' = b[e_2/x]$ by use of E-β-reduce, so we wish to prove $\Gamma \vdash b[e_2/x] : \tau_2$. This follows from the substitution lemma.

□

3 Girard's System-*F*

The problem with the simply-typed λ-calculus is that we can't write lots of simple functions that we might want: for example, we can't even write an identity function! To allow this, we extend the λ-calculus with *type variables*; but not that this is a pure extension of the λ-calculus—every term that was well-typed in the original λ-calculus is still well-typed.

This is called *parametric polymorphism*—our types with type variables can be parameterized over **any** type. Contrast this with *ad-hoc polymorphism*, which only allows paramterizing over some types, or *bounded polymorphism*, which allows paramterizing over any type satisfying a certain *bound* (e.g., must be a collection of some kind). We will explore these type systems later (probably).

3.1 Syntax

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

Examples

1. $\Lambda \alpha . \lambda x : \alpha . x$
2. $(\Lambda \alpha . \lambda x : \alpha . x)$ [bool] true

3.2 Dynamics

Note the evaluation rules are exactly the same, except we need two new rules to handle the new constructs we've added.

$$\frac{e_1 \to e_1'}{e_1[\tau] \to e_1'[\tau]}\ \textsc{E-TypApp} \qquad\qquad \frac{}{(\Lambda \alpha . e)[\tau] \to e[\tau/\alpha]}\ \textsc{E-T-}\beta\textsc{-reduce}$$

3.3 Statics

Again, note the type rules are mostly the same, except we need two new rules to handle the new constructs we've added. However, we also expand type contexts so that they contain type variables—this is just to be sure we don't introduce a type variable twice:

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha$$

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \textsc{T-Abs} \qquad\qquad \frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]}\ \textsc{T-App}$$

4 Simple Extensions

The languages we've defined so far are technically quite expressive, but in practice nobody would want to use such a poorly-featured language. Real programmers want constructs like let, if-then-else, tuples, records, variants, and so on; note that all of these **can** be encoded with the constructs we already have. However, we will extend our language with them, and the appropriate typing rules to make them work.

4.1 If-Then-Else

4.1.1 Syntax

We extend the expression sort, E, to be:

$$\text{E} \quad ::= \quad \ldots \mid \textbf{if}\ \text{Bool}\ \textbf{then}\ E\ \textbf{else}\ E$$

4.1.2 Dynamics

We now need a couple additional rules to evaluate if expressions:

$$\frac{c \to c'}{\textbf{if}\ c\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to \textbf{if}\ c'\ \textbf{then}\ e_1\ \textbf{else}\ e_2}\ \textsc{If-Cond} \qquad \frac{}{\textbf{if}\ \text{true}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to e_1}\ \textsc{If-True}$$

$$\frac{}{\textbf{if}\ \text{false}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to e_2}\ \textsc{If-False}$$

4.1.3 Statics

$$\frac{\Gamma \vdash c : \text{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{if}\ \text{false}\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau}\ \textsc{T-If}$$

4.2 Products

Most languages have so-called "product-types", though they're generally available under different names. If we think of types as sets (and ineed, many types can be represented by sets), then product types are simply products of sets; for example, the type of pairs of integers and booleans, usually written int×bool (or, in Haskell, (int, bool)), is a product type. More generally, we can form products of any finite number of types, but the resulting type will be isomorphic (but **not** equal) to a pair of pairs of pairs of pairs, and so on. It is possible to define products of an infinite number of types: we can view universal types infinite products (with type application as the projections). However, this view is not tremendously useful, so that's all I'll say about it for now.

4.2.1 Syntax

For pairs, we need to extend our types to contain so-called product types as well as the expression-sort. We also need to extend the values to pairs of values. Finally, we want to actually use our pairs—we need to be able to get values our of them. For this purpose we add "projections" to our syntax.

$$\begin{array}{lll} \mathrm{V} & ::= & \ldots \mid (V,V) \\ \mathrm{E} & ::= & \ldots \mid (E,E) \mid E.\ell \mid E.r \\ \tau & ::= & \ldots \mid \tau \times \tau \end{array}$$

4.2.2 Dynamics

$$\frac{e_\ell \to e_\ell'}{(e_\ell, e_r) \to (e_\ell', e_r)}\ \textsc{Pair-L} \qquad \frac{e_r \to e_r'}{(e_\ell, e_r) \to (e_\ell, e_r')}\ \textsc{Pair-R} \qquad \frac{}{(e_\ell, e_r).\ell \to e_\ell}\ \textsc{Proj-L} \qquad \frac{}{(e_\ell, e_r).r \to e_r}\ \textsc{Proj-R}$$

4.2.3 Statics

$$\frac{\Gamma \vdash e_\ell : \tau_\ell \qquad \Gamma \vdash e_r : \tau_r}{\Gamma \vdash (e_\ell, e_r) : \tau_\ell \times \tau_r}\ \textsc{T-Pair} \qquad \frac{\Gamma \vdash e : \tau_\ell \times \tau_r}{\Gamma \vdash e.\ell : \tau_\ell}\ \textsc{T-Proj-L} \qquad \frac{\Gamma \vdash e : \tau_\ell \times \tau_r}{\Gamma \vdash e.r : \tau_r}\ \textsc{T-Proj-R}$$

4.2.4 Products in System-F

As noted before, it is **possible**, but tedious, to represent products in pure System-F, without extensions. There is one central function, `pair`:

$$\begin{aligned} \texttt{pair} &: \forall\alpha.\forall\beta.\alpha \to \beta \to \forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma \\ \texttt{pair} &= \Lambda\alpha.\Lambda\beta.\lambda x : \alpha.\lambda y : \beta.\Lambda\gamma.\lambda p : \alpha \to \beta \to \gamma.p\ x\ y \end{aligned}$$

Then the term (e_1, e_2) coresponds to `pair` $[\tau_1]\ [\tau_2]\ e_1\ e_2$, where $e_1 : \tau_1$ and $e_2 : \tau_2$, the term $e.\ell$ corresponds to $e\ [\tau_1]\ (\lambda x : \tau_1.\lambda y : \tau_2.x)$ and $e.r$ corresponds to $e\ [\tau_2]\ (\lambda x : \tau_1.\lambda y : \tau_2.y)$ We can check that these terms have the expected types, under the assumption that $e : \tau_1 \times \tau_2$, where $\tau_1 \times \tau_2 := \forall\gamma.(\tau_1 \to \tau_2 \to \gamma) \to \gamma$. Below we can that our translation of $e.\ell$ has the expected type; namely $e.\ell : \tau_1$:

$$\frac{\dfrac{\dfrac{\textsc{Assumption}}{\Gamma \vdash e : \forall\gamma.(\tau_1 \to \tau_2 \to \gamma) \to \gamma}}{\Gamma \vdash e\ [\tau_1] : (\tau_1 \to \tau_2 \to \tau_1) \to \tau_1}\ \textsc{T-App} \qquad \dfrac{\dfrac{\dfrac{x : \tau_1 \in \Gamma, x : \tau_1, y : \tau_2}{\Gamma, x : \tau_1, y : \tau_2 \vdash x : \tau_1}\ \textsc{Var}}{\Gamma, x : \tau_1 \vdash \lambda y : \tau_2.x : \tau_2 \to \tau_1}\ \textsc{Abs}}{\Gamma \vdash \lambda x : \tau_1.\lambda y : \tau_2.x : \tau_1 \to \tau_2 \to \tau_1}\ \textsc{Abs}}{\Gamma \vdash e\ [\tau_1]\ (\lambda x : \tau_1.\lambda y : \tau_2.x) : \tau_1}$$

Having defined these translations, the evaluation of terms simply uses the standard evaluation rules of System-F. The downside is that there are many type annotations that are now necessary that we didn't need before.

4.3 Sums

Sometimes it also makes sense to have a function that may have return one of two types—often this is useful when methods may fail with some error, and return an error message (e.g., a string), or succeed and return the desired result of the computation. This is represented by sum types, which we can encode as follows:

4.3.1 Syntax

$$\begin{array}{lll} \mathrm{V} & ::= & \ldots \mid L_\tau(V) \mid R_\tau(V) \\ \mathrm{E} & ::= & \ldots \mid L_\tau(E) \mid R_\tau(E) \mid \textbf{case}\ E\ \textbf{of}\ L_\tau(x) \mapsto E; R_\tau(y) \mapsto E \\ \tau & ::= & \ldots \mid \tau + \tau \end{array}$$

4.3.2 Dynamics

$$\frac{e \to e'}{L_\tau(e) \to L_\tau(e')}\ \textsc{Congr-L} \qquad \frac{e \to e'}{R_\tau(e) \to R_\tau(e')}\ \textsc{Congr-R}$$

$$\frac{e \to e'}{\textbf{case } e \textbf{ of } L_\tau(x) \mapsto e_1; R_\tau(y) \mapsto e_2 \to \textbf{case } e' \textbf{ of } L_\tau(x) \mapsto e_1; R_\tau(y) \mapsto e_2}\ \textsc{Case-Congr}$$

$$\frac{}{\textbf{case } L_\tau(v) \textbf{ of } L_\tau(x) \mapsto e_1; R_\tau(y) \mapsto e_2 \to e_1[v/x]}\ \textsc{Case-L}$$

$$\frac{}{\textbf{case } R_\tau(v) \textbf{ of } L_\tau(x) \mapsto e_1; R_\tau(y) \mapsto e_2 \to e_2[v/y]}\ \textsc{Case-R}$$

4.3.3 Statics

$$\frac{\Gamma \vdash e : \tau_\ell}{\Gamma \vdash L_{\tau_\ell+\tau_r}(e) : \tau_\ell + \tau_r}\ \text{T-}\textsc{Sum-L} \qquad \frac{\Gamma \vdash e : \tau_r}{\Gamma \vdash R_{\tau_\ell+\tau_r}(e) : \tau_\ell + \tau_r}\ \text{T-}\textsc{Sum-R}$$

$$\frac{\Gamma \vdash e : \tau_\ell + \tau_r \qquad \Gamma, x : \tau_\ell \vdash e_1 : \tau \qquad \Gamma, y : \tau_r \vdash e_2 : \tau}{\textbf{case } e \textbf{ of } L_{\tau_\ell+\tau_r}(x) \mapsto e_1; R_{\tau_\ell+\tau_r}(y) \mapsto e_2 : \tau}\ \text{T-}\textsc{Case}$$

4.3.4 Sums in System-*F*

As noted before, it is also possible to represent sums in pure System-F.

As in products, we convert our "constructors" into System-F terms, which build terms that, given a "destructor" function, return some result type. Because we have two constructors for sums, the left and the right, we need two functions. Both are shown below. Checking that they work as desired is fairly straighforward.

$$\begin{aligned}
&\texttt{left} : \forall\alpha.\forall\beta.\alpha \to \forall\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma \\
&\texttt{left} = \Lambda\alpha.\Lambda\beta.\lambda x : \alpha.\Lambda\gamma.\lambda p_\ell : \alpha \to \gamma.\lambda p_r : \beta \to \gamma.p_\ell\ x
\end{aligned}$$

$$\begin{aligned}
&\texttt{right} : \forall\alpha.\forall\beta.\beta \to \forall\gamma.(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma \\
&\texttt{right} = \Lambda\alpha.\Lambda\beta.\lambda y : \beta.\Lambda\gamma.\lambda p_\ell : \alpha \to \gamma.\lambda p_r : \beta \to \gamma.p_r\ y
\end{aligned}$$

5 Kinds

Up until now, our types have been pretty simple, with only a couple of ways to make new types. But in "real" programming languages, we have many ways to construct types: in Haskell, we have functors, monads, and many other types that we can't really describe yet. In fact, even our description of pairs is informal; we wrote that $\tau_1 \times \tau_2 := \forall\gamma.(\tau_1 \to \tau_2 \to \gamma) \to \gamma$, but this is a somewhat troublesome: the only mechanism for substitution in types we have is the universal quantifier $\forall$, but we didn't use it for τ_1 or τ_2. To have more interesting types, we need to define these concepts in a more general sense.

This is where *kinds* come in—kinds are basically "types for types". They describe which type-level functions (typically called type constructors, in the simple cases, like pairs) we can apply to which types. The simplest kind system, which we'll discuss, is the same one that Haskell uses (modulo extensions).

For our discussion, we'll be using System-F as a base, and we'll obtain the well known System-F_ω. Note: it's called System-F_ω because it's essentially the limiting case of the various System-F_n's, where we allowed only kinds of order n or lower; ω being the first infinite ordinal.

5.1 Syntax

Below is the full syntax for System-F_ω.

$$\begin{array}{lcl}
\mathrm{V} & ::= & x \mid \lambda x:\tau.E \mid \Lambda\alpha::\mathcal{K}.E \\
\mathrm{E} & ::= & V \mid E\ E \mid E\ [\tau] \\
\tau & ::= & \tau \to \tau \mid \forall\alpha::\mathcal{K}.\tau \mid \lambda\alpha::\mathcal{K}.\tau \mid \tau\ \tau \\
\mathcal{K} & ::= & * \mid \mathcal{K} \implies \mathcal{K} \\
\Gamma & ::= & \cdot \mid \Gamma, x:\tau \mid \Gamma, \alpha::\mathcal{K}
\end{array}$$

Here, the kind $* \implies *$ means a type constructor that takes a single (type) argument, and produces a type—for example, $\texttt{List} :: * \implies *$, and $\texttt{List}\alpha :: *$.

5.2 Dynamics

The dynamics are exactly the same as before; kinds are an entirely **type-level** construct.

5.3 Statics

There are now two additional considerations. In previous systems, we were only concerned with type-checking terms; now that we have introduced a limited form of type-level computation, we must ensure that these type-level computations also do not go wrong—we need rules for *kind checking*, and rules to describe how type computations work; in some sense, these are evaluation rules, but they aren't applied during execution, but during typechecking, so we give them here.

First, we give the the kinding rules, which are quite similar to the typing rules in the simply-typed λ-calculus. The main difference is that we now need to give kinding rules for types as well.

$$\frac{\alpha::\mathcal{K}\in\Gamma}{\Gamma\vdash\alpha::\mathcal{K}}\ \textsc{K-Var} \qquad \frac{\Gamma,\alpha::\mathcal{K}_1\vdash\tau::\mathcal{K}_2}{\Gamma\vdash\lambda\alpha::\mathcal{K}_1.\tau::\mathcal{K}_1\implies\mathcal{K}_2}\ \textsc{K-Abs} \qquad \frac{\Gamma\vdash\tau::\mathcal{K}_1\implies\mathcal{K}_2 \qquad \Gamma\vdash\tau'::\mathcal{K}_1}{\Gamma\vdash\tau\tau'::\mathcal{K}_2}\ \textsc{K-App}$$

$$\frac{\Gamma\vdash\tau_1::* \qquad \Gamma\vdash\tau_2::*}{\Gamma\vdash\tau_1\to\tau_2::*}\ \textsc{K-Func} \qquad \frac{\Gamma,\alpha::\mathcal{K}\vdash\tau::*}{\Gamma\vdash\forall\alpha::\mathcal{K}.\tau::*}\ \textsc{K-Forall}$$

Now for the type computation rules:

$$\frac{\sigma_1\Downarrow\tau_1 \qquad \sigma_1\Downarrow\tau_2}{(\sigma_1\to\sigma_2)\Downarrow(\tau_1\to\tau_2)}\ \textsc{TE-Func} \qquad \frac{\sigma\Downarrow\tau}{\forall\alpha::\mathcal{K}.\sigma\Downarrow\forall\alpha::\mathcal{K}.\tau}\ \textsc{TE-Forall}$$

$$\frac{\sigma\Downarrow\tau}{\lambda\alpha::\mathcal{K}.\sigma\Downarrow\lambda\alpha::\mathcal{K}.\tau}\ \textsc{TE-Abs} \qquad \frac{}{(\lambda\alpha::\mathcal{K}.\sigma)\tau\Downarrow\sigma[\tau/\alpha]}\ \textsc{TE-App}$$

$$\frac{\sigma_1\Downarrow\tau_1 \qquad \sigma_2\Downarrow\tau_2}{\sigma_1\sigma_2\Downarrow\tau_1\tau_2}\ \textsc{TE-App-Congr}$$

And finally, the typing rules, which must ensure that every term has the kind of types. We only show rules that are different or new:

$$\frac{\Gamma \vdash \tau_1 :: * \qquad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2}\ \textsc{Abs} \qquad \frac{\Gamma, \alpha :: \mathcal{K} \vdash e : \tau}{\Gamma \vdash \Lambda\alpha :: \mathcal{K}.e : \forall\alpha :: \mathcal{K}.\tau}\ \textsc{T-Abs}$$

$$\frac{\Gamma \vdash f : \forall\alpha :: \mathcal{K}.\tau \qquad \Gamma \vdash \sigma :: \mathcal{K}}{\Gamma \vdash f[\sigma] : \tau[\sigma/\alpha]}\ \textsc{T-App} \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma \Downarrow \tau \qquad \Gamma \vdash \tau :: \mathcal{K}}{\Gamma \vdash e : \tau}\ \textsc{T-Reduce}$$

5.4 Extended Example

Below we develop the necessary definitions to express the Haskell function **sequence** :: **Monad** m =>[m a] –> m [a]. To do this, we must define monads and lists; we will also define functors and pairs along the way for instructional purposes, but also for convenience.

5.5 Pairs

The first new type that we will define is a pair. Recall from our earlier discussion that we can define a function `pair` that builds pairs. There are only a couple slight changes to the syntax we must adhere to, that is, the kind must be specified on type abstractions:

$$\texttt{pair} : \forall\alpha :: *.\forall\beta :: *.\alpha \to \beta \to \forall\gamma :: *.(\alpha \to \beta \to \gamma) \to \gamma$$
$$\texttt{pair} = \Lambda\alpha :: *.\Lambda\beta :: *.\lambda x : \alpha.\lambda y : \beta.\Lambda\gamma :: *.\lambda p : \alpha \to \beta \to \gamma.p\ x\ y$$

This will get quite tedious, so we make the following simplifcations to the syntax:

1. If the kind of a type variable is $*$, it will simply be omitted; otherwise, it will be written as specified above.
2. For repeated abstractions and quantifiers, we will condense them into a single abstraction/quantifier.

There, we will write the above as:

$$\texttt{pair} : \forall\alpha,\beta.\alpha \to \beta \to \forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma$$
$$\texttt{pair} = \Lambda\alpha,\beta.\lambda x : \alpha, y : \beta.\Lambda\gamma.\lambda p : \alpha \to \beta \to \gamma.p\ x\ y$$

We also define the type of pairs using infix notation to be

$$\cdot \times \cdot := \lambda\alpha,\beta.\forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma$$

Note that defining things like this is not strictly allowed in the language's syntax; however, consider the following program:

$$\textbf{let}\ id : \forall\alpha.\alpha \to \alpha := \Lambda\alpha.\lambda x : \alpha.x\ \textbf{in}\ id\ id$$

We can transform this program into one without the definition by changing it to:

$$(\lambda id : \forall\alpha.\alpha \to \alpha.id\ id)(\Lambda\alpha.\lambda x : \alpha.x)$$

So for convenience's sake, as the terms are already ugly enough, we ignore the technicalities of managing these definitions.

So we can write the type of `pair` as:

$$\texttt{pair} : \forall\alpha,\beta.\alpha \to \beta \to \alpha \times \beta$$

Now we define two new functions, the projections of the pair, `fst` and `snd`:

$$
\begin{aligned}
&\texttt{fst} : \forall \alpha, \beta. \alpha \times \beta \to \alpha \\
&\texttt{fst} = \Lambda \alpha, \beta. \lambda pair : \alpha \times \beta. pair\ [\alpha]\ (\lambda x : \alpha, y : \beta. x) \\
&\texttt{snd} : \forall \alpha, \beta. \alpha \times \beta \to \beta \\
&\texttt{snd} = \Lambda \alpha, \beta. \lambda pair : \alpha \times \beta. pair\ [\beta]\ (\lambda x : \alpha, y : \beta. y)
\end{aligned}
$$

It's worth noting at this point that this is already something we couldn't formally write in System F alone, and we obtain the typing rules and evaluation rules for products without having to define them ourselves (and more importantly, prove that everything still works).

5.6 Lists

We now define lists, and two constructors on lists: `nil` and `cons`. Recall that the intuition for pairs is that we store the necessary data "inside" the function (e.g., in a concrete implementation, in a closure); for lists we do something similar, but the way in which lists are consumed in somewhat more complicated. But, recalling our favorite programming language Haskell, we can write everything we want using just `foldl`; our representation of lists will essentialyl be `foldl` with one argument "missing" (in fact, encoded in the "closure", so to speak). First, the declaration of the type:

$$\texttt{List} := \lambda \alpha. \forall \beta. (\alpha \to \beta \to \beta) \to \beta \to \beta$$

Now we define the two constructors:

$$
\begin{aligned}
&\texttt{nil} : \forall \alpha. \texttt{List}\ \alpha \\
&\texttt{nil} = \Lambda \alpha. \Lambda \beta. \lambda red : \alpha \to \beta \to \beta, init : \beta. init \\
&\texttt{cons} : \forall \alpha. \alpha \to \texttt{List}\ \alpha \to \texttt{List}\ \alpha \\
&\texttt{cons} = \Lambda \alpha. \lambda x : \alpha. \lambda xs : \texttt{List}\ \alpha. \Lambda \beta. \lambda red : \alpha \to \beta \to \beta, init : \beta. xs\ [\beta]\ red\ (red\ x\ init)
\end{aligned}
$$

5.6.1 Summing a List

Here is an example program, summing a list of numbers: It assumes we have numbers, but we could define them in pure System F_ω (or even System F) if we chose (e.g., as Church numerals).

$$
\begin{aligned}
&\texttt{sum} : \texttt{List Nat} \to \texttt{Nat} \\
&\texttt{sum} = \lambda xs : \texttt{List Nat}. xs\ [\texttt{Nat}]\ (\lambda x : \texttt{Nat}. \lambda rest : \texttt{Nat}. x + rest)\ 0
\end{aligned}
$$

Then:

$$
\begin{aligned}
&\texttt{sum}\ (\texttt{cons}\ [\texttt{Nat}]\ 1 (\texttt{cons}\ [\texttt{Nat}]\ 2 (\texttt{nil}[\texttt{Nat}]))) \\
&\to \\
&3
\end{aligned}
$$

5.7 Functors

The previous two examples aren't very convincing examples of why we needed kinds—after all, `List` and $\cdot \times \cdot$ are simple abbreviations. Of course, making them formal is nice, but maybe not nice enough to justify all the effort. However, we get much more: we can use so-called "higher-kinded", like Haskell does, and define more complicated structures like functors:

$$\mathtt{Functor} = \lambda f :: * \implies *.\forall\alpha,\beta.(\alpha \to \beta) \to (f\ \alpha \to f\ \beta)$$

Essentially, a functor is a structure that can be mapped over. For example, `List` is a functor; what does it mean to say something is a functor? In this context, "f is a functor" means that we can write a term of type $\mathtt{Functor}\ f$; below is the definition for lists:

$$\begin{aligned}&\mathtt{map} : \mathtt{Functor\ List}\\ &\mathtt{map} = \Lambda\alpha,\beta.\lambda g : \alpha \to \beta.\lambda xs : \mathtt{List}\ \alpha.xs\ [\mathtt{List}\ \beta]\ (\lambda a : \alpha, tail : \mathtt{List}\ \beta.cons\ [\beta]\ (g\ a)\ tail)\ (\mathtt{nil}\ [\beta])\end{aligned}$$

Suppose we have a function $\mathtt{increment} : \mathtt{Nat} \to \mathtt{Nat}$; using map, we can write a functor that increments a whole list of nats: $\mathtt{map}\ [\mathtt{Nat}]\ [\mathtt{Nat}]\ \mathtt{increment}$.

Functors are incredibly useful, because they let us abstract over data structures—they provide an interface of being "mappable". Many common data structures, such as pairs, trees, dictionaries, and many more are all functors. Without kinds, we couldn't generalize this idea in a programming language.

5.8 Monads

Another useful type is the *monad*; we won't talk too much about what these are, but they're essentially a special kind of functor that can be used to compose "effectful" computations. They are essentially a pair of two functions, called `pure` (or `return`) and `bind`. `pure` "wraps" a pure value in the monad, and `bind` performs an effectful computation on a monadic value. We define monads as pair of these functions:

$$\mathtt{Monad} = \lambda m :: * \implies *.(\forall\alpha.\alpha \to m\ \alpha) \times (\forall\alpha,\beta.m\ \alpha \to (\alpha \to m\ \beta) \to m\ \beta)$$

We then define `pure` and `bind` as projections on monads.

$$\begin{aligned}&\mathtt{pure} : \forall m :: * \implies *.\mathtt{Monad}\ m \to \forall\alpha.\alpha \to m\ \alpha\\ &\mathtt{pure} = \Lambda m :: * \implies *.\lambda mon : \mathtt{Monad}\ m.\mathtt{fst}\ [\forall\alpha.\alpha \to m\ \alpha]\ [\forall\alpha,\beta.m\ \alpha \to (\alpha \to m\ \beta) \to m\ \beta]\ mon\\ &\mathtt{bind} : \forall m :: * \implies *.\mathtt{Monad}\ m \to \forall\alpha,\beta.m\ \alpha \to (\alpha \to m\ \beta) \to m\ \beta\\ &\mathtt{bind} = \Lambda m :: * \implies *.\lambda mon : \mathtt{Monad}\ m.\mathtt{snd}\ [\forall\alpha.\alpha \to m\ \alpha]\ [\forall\alpha,\beta.m\ \alpha \to (\alpha \to m\ \beta) \to m\ \beta]\ mon\end{aligned}$$

5.9 Writing `sequence`

We are now ready to implement `sequence`. For brevity we write `pure` and `bind` instead of $\mathtt{pure}\ [m]\ mon$ and $\mathtt{bind}\ [m]\ mon$, respectively.

$$\begin{aligned}&\mathtt{sequence} : \forall m :: * \implies *.\mathtt{Monad}\ m \to \forall\alpha.\mathtt{List}\ (m\ \alpha) \to m\ (\mathtt{List}\ \alpha)\\ &\mathtt{sequence} = \Lambda m :: * \implies *.\lambda mon : \mathtt{Monad}\ m.\Lambda\alpha.\lambda xs : \mathtt{List}\ (m\ \alpha).\\ &\qquad xs\ [m\ (\mathtt{List}\ \alpha)]\\ &\qquad\quad (\lambda mx : m\ \alpha, mxs : m\ (\mathtt{List}\ \alpha).\mathtt{bind}\ [\alpha]\ [\mathtt{List}\ \alpha]\ mx\\ &\qquad\qquad (\lambda x : \alpha.\mathtt{bind}\ [\mathtt{List}\ \alpha]\ [\mathtt{List}\ \alpha]\ mxs\\ &\qquad\qquad\quad (\lambda xs : \mathtt{List}\ \alpha.\mathtt{pure}\ [\mathtt{List}\ \alpha]\ (\mathtt{cons}\ [\alpha]\ x\ xs)))\\ &\qquad\quad (\mathtt{pure}\ [\mathtt{List}\ \alpha]\ (\mathtt{nil}\ [\alpha]))\end{aligned}$$

6 Dependent Types

6.1 Background and Motivation

6.2 Pure First-Order Dependent Types

We define the *pure*, *first-order*, *dependent type* sytem λP. Pure refers to the abscence of Σ-types (dependent sum types), and first-order means that we can't quantify over types, like we can in System-F.

6.2.1 Syntax

Here we think of Type as the sort of "proper" types and Kind as the sort of "kinds." Both essentially are just syntactic constants.

$$\begin{array}{rcl} \mathcal{T} & ::= & \mathcal{S} \mid x \mid \lambda x : \mathcal{T}.\mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \mathbf{\Pi} x : \mathcal{T}.\mathcal{T} \\ \mathcal{S} & ::= & \textsc{Type} \mid \textsc{Kind} \\ \Gamma & ::= & \cdot \mid \Gamma, x : \mathcal{T} \end{array}$$

6.2.2 Dynamics

The only evaluation rules are congruence rules and the standard β-reduction rule; we write this relation as $\rightarrow_\beta$, and we write its reflexive-transitive closure as $\rightsquigarrow_\beta$.

6.2.3 Statics

$$\frac{}{\Gamma \vdash \textsc{Type} : \textsc{Kind}}\ \textsc{Type-Kind} \qquad \frac{x : \tau \in \Gamma \qquad \Gamma \vdash \tau : s}{\Gamma \vdash x : \tau}\ \textsc{Var}$$

$$\frac{\Gamma \vdash \mathbf{\Pi} x : \tau.\sigma : s \qquad \Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x : \tau.e : \mathbf{\Pi} x : \tau.\sigma}\ \textsc{Abs} \qquad \frac{\Gamma \vdash f : \mathbf{\Pi} x : \tau.\sigma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash f\ e : \sigma[e/x]}\ \textsc{App}$$

$$\frac{\Gamma \vdash \tau : r \qquad \Gamma, x : \tau \vdash \sigma : s}{\Gamma \vdash \mathbf{\Pi} x : \tau.\sigma : t}\ \textsc{Pi} \quad \text{where } (r, s, t) \in \{(\textsc{Type}, \textsc{Type}, \textsc{Type}), (\textsc{Type}, \textsc{Kind}, \textsc{Kind})\}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau =_\beta \sigma \qquad \Gamma \vdash \sigma : s}{\Gamma \vdash e : \sigma}\ \textsc{Equiv}$$

We define $\tau =_\beta \sigma$ by the symmetric closure of $\rightsquigarrow_\beta$ (or the reflexive-symmetric-transitive closure of $\rightarrow_\beta$).

6.2.4 Examples

Now that we have this system, how do we actually program in it? The answer is we sort of can't actually **do** anything yet—much like the simply-typed λ-calculus, there actually aren't any well-typed ground terms.

However, using an appropriate context, we can still do some interesting things. For the the next section we'll be using (and expanding) onto the following context:

$$\begin{aligned}
\mathbb{N} &: \textsc{Type}\\
0 &: \mathbb{N}\\
\mathcal{s} &: \Pi n : \mathbb{N}.\mathbb{N}\\
\texttt{Vector} &: \Pi n : \mathbb{N}.\textsc{Type}\\
\texttt{nil} &: \texttt{Vector}\ 0\\
\texttt{cons} &: \Pi n : \mathbb{N}.\Pi x : \mathbb{N}.\Pi xs : \texttt{Vector}\ n.\texttt{Vector}\ (\mathcal{s}\ n)
\end{aligned}$$

Note that we typically write a type like $\Pi x : A.B$ as $A \to B$ if x does not appear in B. With that in mind, we will actually write the type sof $\mathcal{s}$, `Vector`, and cons as:

$$\begin{aligned}
\mathcal{s} &: \mathbb{N} \to \mathbb{N}\\
\texttt{Vector} &: \mathbb{N} \to \textsc{Type}\\
\texttt{cons} &: \Pi n : \mathbb{N}.\mathbb{N} \to \texttt{Vector}\ n \to \texttt{Vector}\ (\mathcal{s}\ n)
\end{aligned}$$

Which looks more like types that we're used to. The main attraction of dependent types is the ability for typechecking to catch bugs that we might make in our code, so we'll focus on typechecking rather than evaluation. Let's look at some well-typed terms:

$$\begin{aligned}
0 &: \mathbb{N}\\
\mathcal{s}\ 0 &: \mathbb{N}\\
\mathcal{s}\ (\mathcal{s}\ 0) &: \mathbb{N}\\
\texttt{nil} &: \texttt{Vector}\ 0\\
\texttt{cons}\ 0\ 0\ \texttt{nil} &: \texttt{Vector}\ (\mathcal{s}\ 0)
\end{aligned}$$

Note that the following term is **not** well-typed — as we would expect.

$$\texttt{cons}\ 0\ 0\ (\texttt{cons}\ 0\ 0\ \texttt{nil})$$

This is because $\texttt{cons}\ 0\ 0\ \texttt{nil} : \texttt{Vector}\ 1$, but it should be a $\texttt{Vector}\ 0$. Note that we can prove this is **not** well-typed by the following lemma:

Lemma 2 (Uniqueness of types). *If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_\beta B$.*

We can see above that both $\texttt{Vector}\ 1$ and $\texttt{Vector}\ 0$ are already fully β-reduced, and they are not the same, so by the lemma, we **cannot** prove $\Gamma \vdash \texttt{cons}\ 0\ 0\ \texttt{nil} : \texttt{Vector}\ 0$. Technically this argument relies on confluence:

Lemma 3 (Confluence). *If $A =_\beta B$, then there is some C so that $A \rightsquigarrow_\beta C$ and $B \rightsquigarrow_\beta C$.*

But hopefully it is clear that this system is confluent.

This shows us that our type system is useful for something—we can't write utter nonsense like this. However, you may note this nonsense (of cons-ing to vectors of improper length) has only been introduced because of dependent types in the first place. We can, however, define functions that can only be called with value arguments:

$$\texttt{head} : \Pi n : \mathbb{N}.\texttt{Vector}\ (\mathcal{s}\ n) \to \mathbb{N}$$

This function can only be called if we supply it a nonempty vector, which lets us actually eliminate the runtime check for this and, more importantly, the possibility of the error. For example, we can type:

$$\texttt{head}\ 0\ (\texttt{cons}\ 0\ 0\ \texttt{nil}) : \mathbb{N}$$

But we **cannot** type `head 0 nil` (or `head` n `nil` for any $n \in \mathbb{N}$); again, this is because we cannot β-reduce `Vector 0` to `Vector` $(s\ n)$ for any n.

So this type system can save us from some errors, but we're missing a lot of things. For example, we can't really build up any actual terms or do any computation because of the limitations of the language. We also can't express things like induction, which is a second-order concept. But, with some simple extensions to the type system, we will be able to do this.

6.3 Pure Type Systems

In fact, the many variants of the λ-calculus (sometimes referred to as the λ-Cube), System F, System F_ω, and so forth, can be formalized as a pure type system. In fact, we can also express the Calculus of Inductive Constructions, the foundations of Coq's type syste, as a Pure Type System.

First let's define a pure type system—in fact, we already almost did this above, so we'll just list the rules that changed.

Definition 1. *A* pure type system *is a triple of sets* $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, *called the* sorts, axioms, *and* rules, *where* $\mathcal{A} \subseteq \mathcal{S}^2$ *and* $\mathcal{R} \subseteq \mathcal{S}^3$, *with the following rules:*

$$\frac{}{\Gamma \vdash r : s}\ \textsc{Axiom} \quad \text{if } (r,s) \in \mathcal{A} \qquad\qquad \frac{\Gamma \vdash \tau : r \qquad \Gamma, x : \tau \vdash \sigma : s}{\Gamma \vdash \mathbf{\Pi} x : \tau.\sigma : t}\ \textsc{Pi} \quad \text{if } (r,s,t) \in \mathcal{R}$$

6.4 The Calculus of Inductive Constructions

7 References

[TODO]

Note that this document roughly follows Benjamin Pierce's *Types and Programming Languages* (as well as Pierce et al.'s *Advanced Types and Programming Languages*) and Robert Harper's *Practical Foundations for Programming Languages*.

Also: `http://www4.di.uminho.pt/~mjf/pub/SFV-CIC-2up.pdf`