

# CS256: Programming Languages and Semantics

## Parametric Polymorphism

<http://www.eecs.harvard.edu/~greg/cs256sp2006/>

Greg Morrisett  
greg@eecs.harvard.edu  
327 Maxwell Dworkin

March 17, 2006

## 1 Polymorphism

*Warning: these notes are a little rough so take them with a grain of salt...*

The introduction of types into a programming language eliminates certain classes of errors, but at the price that we reject some programs that are in fact perfectly sensible. In particular, for the untyped lambda calculus, the identity function  $\lambda x.x$  can be used on any argument, but in the simply-typed lambda calculus, we're forced to put a type on the argument (e.g.,  $\lambda x:\tau.x$ ) and then can only apply the function to arguments of that type, in spite of the fact that nothing goes wrong (at least in the model) when we apply it to a value of a different type.

This is an instance of a more general problem: Programmers and programming languages want to make it possible to abstract everything. That follows the general principle that you shouldn't write something twice because then it's harder to maintain the multiple copies that you've written down. On the other hand, it can be much easier to write a context-specific specification and prove properties relative to a specific context. When we start abstracting with respect to the context in which a piece of code must be used, we often need to introduce new forms of abstraction at the *specification-level* to make up for the lack of concrete contextual information.

Polymorphism is a way of abstracting over typing specifications and allows us to re-use a given piece of code in many (typing) contexts. To that end, it's crucial for making abstraction possible.

There are a number of different flavors of polymorphism that you'll hear about in the literature including:

- parametric polymorphism,
- sub-type polymorphism,
- bounded polymorphism,
- ad hoc polymorphism (a.k.a. overloading),
- intensional polymorphism, and
- type-class polymorphism,

We're going to discuss the first couple of these in detail, but if you want to know more about the others, just let me know.

## 2 Parametric Polymorphism

This style of polymorphism is something that is now commonly understood as it arises in languages such as ML, Haskell, and now Java and C#. In fact, all of these languages have slightly different forms of parametric polymorphism (aka uniform polymorphism). We'll start by looking at the pure polymorphic lambda calculus (aka system-F or  $F_2$ ).

We'll augment our abstract syntax as follows:

$$\begin{array}{lcl} \text{(types)} & \tau & ::= \alpha \mid b \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha. \tau \\ \text{(terms)} & e & ::= c \mid x \mid \lambda x : \tau. e \mid e_1 e_2 \mid \Lambda \alpha. e \mid e \tau \\ \text{(values)} & v & ::= c \mid \lambda x : \tau. e \mid \Lambda \alpha. e \end{array}$$

So types have been extended with type variables ( $\alpha$ ) and universal type schemes ( $\forall \alpha. \tau$ ). The term  $\Lambda \alpha. e$  is a type abstraction and should be thought of as a function that takes a type as a parameter (with formal parameter name  $\alpha$ ) and that computes a value whose type is typically dependent upon  $\alpha$ . The term  $e \tau$  is a type application, where we're passing the type  $\tau$  as an argument to a type abstraction.

The big-step evaluation rules are extended with only one other case (for type applications):

$$\frac{e \Downarrow \Lambda \alpha. e' \quad e'[\tau/\alpha] \Downarrow v}{e \tau \Downarrow v}$$

Recall that all values evaluate to themselves, so  $\Lambda \alpha. e \Downarrow \Lambda \alpha. e$ .

It should be fairly easy to see how to extend the small-step operational semantics to handle the new constructs:

$$\frac{e \mapsto e'}{e \tau \mapsto e' \tau}$$

$$(\Lambda \alpha. e) \tau \mapsto e[\tau/\alpha]$$

The typing judgment must be modified to take into account type variables that are in scope. We write  $\Delta; \Gamma \vdash e : \tau$  to mean that  $e$  can be assigned the type  $\tau$  under the assumptions  $\Delta$  and  $\Gamma$ , where  $\Delta$  records the type variables that are in scope, and  $\Gamma$  records the term variables (and their types) that are in scope. For uniformity, we will treat  $\Delta$  as a finite map from type variables to *kinds* (a type for a type) where for now, the only kind that we have is  $\star$  (aka “type”):

$$\begin{array}{lcl} \text{(kinds)} & \kappa & ::= \star \\ \text{(type variable context)} & \Delta & ::= \bullet \mid \Delta, \alpha : \kappa \end{array}$$

Additionally, we shall need a judgment  $\Delta \vdash \tau : \star$  which asserts that the type  $\tau$  is well-formed under the assumptions in  $\Delta$ . That is, the only free type-variables in  $\tau$  should be drawn from  $\Delta$ .

The rules for defining the second judgment are as follows:

$$\begin{array}{c} \Delta \vdash b : \star \\ \Delta \vdash \alpha : \Delta(\alpha) \\ \frac{\Delta \vdash \tau_1 : \star \quad \Delta \vdash \tau_2 : \star}{\Delta \vdash \tau_1 \rightarrow \tau_2 : \star} \\ \frac{\Delta, \alpha : \star \vdash \tau : \star}{\Delta \vdash \forall \alpha. \tau : \star} \quad (\alpha \notin \Delta) \end{array}$$

The rules for defining the main typing judgment are as follows:

$$\Delta; \Gamma \vdash x : \Gamma(x)$$

$$\begin{array}{c}
\Delta; \Gamma \vdash c : b \\
\frac{\Delta \vdash \tau_1 : \star \quad \Delta; \Gamma, x:\tau_1 \vdash e : \tau_2 \quad (x \notin \Gamma)}{\Delta; \Gamma \vdash \lambda x:\tau_1. e : \tau_1 \rightarrow \tau_2} \\
\frac{\Delta; \Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Delta; \Gamma \vdash e_2 : \tau'}{\Delta; \Gamma \vdash e_1 e_2 : \tau} \\
\frac{\Delta, \alpha:\star; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda \alpha. e : \forall \alpha. \tau} \quad (\alpha \notin \Delta) \\
\frac{\Delta; \Gamma \vdash e : \forall \alpha. \tau \quad \Delta \vdash \tau' : \star}{\Delta; \Gamma \vdash e : \tau[\tau'/\alpha]}
\end{array}$$

Note that the inference rules try to preserve the invariant that every type introduced in  $\Gamma$  is well formed with respect to  $\Delta$ . Also note that, as usual, we consider  $\alpha$  to be bound with scope  $\tau$  in the type  $\forall \alpha. \tau$ , and we consider  $\alpha$  to be bound with scope  $e$  in the term  $\Lambda \alpha. e$ . Furthermore, we consider both types and terms to be equivalent up to alpha-conversion of bound type and term variables.

This is particularly important here because implicit in the rules is the fact that we are relying upon type equality which is no longer trivial syntactic equality (since we've introduced bound variables.) In short, if you go to mechanize the polymorphic lambda calculus, then you must worry about alpha-equivalence of types (which can complicate the meta-theory if your environment doesn't provide good support for this.)

As with the simply-typed lambda calculus, it's possible to show a number of properties for this language including:

- Preservation: evaluation preserves types.
- Progress: well-typed expressions will not get stuck.
- Soundness: follows from (1) and (2).
- Termination: all well-typed programs terminate.
- Reduction: equality on terms is decidable by reducing to unique normal forms.

Proving the first three of these is relatively straightforward. Proving the last two is not as we shall see later. The essence of the problem is that a quantified type, such as  $\forall \alpha. \alpha \rightarrow \alpha$ , allows  $\alpha$  to range over *any* type, including itself! With such power, it seems sure that there must be a way to construct a looping term. But in fact, we cannot. The argument requires a different (but related) construction than the one we did for the simply-typed lambda calculus and we'll sketch it later on.

### 3 Encodings

In spite of the fact that all programs in System-F terminate (and have normal forms), it's a surprisingly rich language. For instance, we can encode or simulate a number of other language features including products, sums, natural numbers, and other inductive types such as lists or trees. Furthermore, we can code up operations such as addition on numbers, (polymorphic) maps and folds on lists or trees, and a surprising number of other things.

Let's start with products: How can we encode " $\tau_1 \times \tau_2$ "? The solution is to think about how you use (i.e., eliminate) such a value and to incorporate this into the constructor. The way we use a product is to project its components and then feed them into some computation as in:

```

let (x,y) = p
in
  e
end

```

Thinking about the encoding of `let` we can simplify this to:

$$(\lambda(x, y).e) p$$

At this point, we can think of  $p$  as taking the “let” as an argument instead of the other way around:

$$p(\lambda(x, y).e)$$

and then uncurrying the function we get:

$$p(\lambda x.\lambda y.e)$$

So a pair is something that takes a function  $f$  of two arguments, and then delivers an  $x$  and  $y$  to that function. That is, a pair of a  $\tau_1$  and a  $\tau_2$  should have type:

$$(\tau_1 \rightarrow \tau_2 \rightarrow \tau) \rightarrow \tau$$

for any result type  $\tau$  (since we want to be able to pass in any “let” for the argument.) Abstracting the result type tells us that the encoding of  $\tau_1 \times \tau_2$  should have type:

$$\forall \alpha. (\tau_1 \rightarrow \tau_2 \rightarrow \alpha) \rightarrow \alpha$$

How do we build a value of this type? Well, there’s just about only one way to write a function with the type:

$$\begin{aligned} \tau_1 \rightarrow \tau_2 \rightarrow \text{“}\tau_1 \times \tau_2\text{”} &= \\ \tau_1 \rightarrow \tau_2 \rightarrow (\forall \alpha. (\tau_1 \rightarrow \tau_2 \rightarrow \alpha) \rightarrow \alpha) \end{aligned}$$

and here it is:

$$\lambda x:\tau_1.\lambda y:\tau_2.\Lambda \alpha.\lambda f : (\tau_1 \rightarrow \tau_2 \rightarrow \alpha).f x y$$

Abstracting  $\tau_1$  and  $\tau_2$ , we can define:

$$\begin{aligned} \text{pair} &: \forall \beta, \gamma. \beta \rightarrow \gamma \rightarrow (\forall \alpha. (\beta \rightarrow \gamma \rightarrow \alpha) \rightarrow \alpha) \\ \text{pair} &= \Lambda \beta, \gamma. \lambda x:\beta. \lambda y:\gamma. \Lambda \alpha. \lambda f : (\beta \rightarrow \gamma \rightarrow \alpha). f x y \end{aligned}$$

and then we can define the first projection ( $\pi_1$ ) as follows:

$$\begin{aligned} \pi_1 &: \forall \beta, \gamma. \beta \times \gamma \rightarrow \beta \\ \pi_1 &: \forall \beta, \gamma. \forall \alpha. (\beta \rightarrow \gamma \rightarrow \alpha) \rightarrow \alpha \\ \pi_1 &= \lambda \beta, \gamma. \lambda f. f [\beta] (\lambda x, y. x) \end{aligned}$$

and the second projection similarly.

What about sums (i.e., datatypes) of the form  $\tau_1 + \tau_2$ ? Again, we think of how we use or eliminate the sum as in:

$$\text{case } s \text{ of } x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2$$

So we can think of  $s$  as taking two functions  $\lambda x_1.e_1$  and  $\lambda x_2.e_2$  as arguments, and calling the appropriate function depending upon whether  $s$  is a  $\tau_1$  or a  $\tau_2$ :

$$s(\lambda x_1.e_1)(\lambda x_2.e_2)$$

So that tells us that the type of  $s$  should be something like:

$$\forall \alpha. (\tau_1 \rightarrow \alpha) \rightarrow (\tau_2 \rightarrow \alpha) \rightarrow \alpha$$

Then we can define the first injection as:

$$\begin{aligned} \text{in}_1 &: \forall \beta, \gamma. \beta \rightarrow (\beta + \gamma) \\ \text{in}_1 &: \forall \beta, \gamma. \beta \rightarrow (\forall \alpha. (\tau_1 \rightarrow \alpha) \rightarrow (\tau_2 \rightarrow \alpha) \rightarrow \alpha) \\ \text{in}_1 &= \Lambda \beta, \gamma. \lambda x. (\Lambda \alpha. \lambda f_1, f_2. f_1 x) \end{aligned}$$

and the second injection as:

$$\begin{aligned} \text{in}_2 & : \quad \forall \beta, \gamma. \gamma \rightarrow (\beta + \gamma) \\ \text{in}_2 & : \quad \forall \beta, \gamma. \gamma \rightarrow (\forall \alpha. (\tau_1 \rightarrow \alpha) \rightarrow (\tau_2 \rightarrow \alpha) \rightarrow \alpha) \\ \text{in}_2 & = \quad \Lambda \beta, \gamma. \lambda x. (\Lambda \alpha. \lambda f_1, f_2. f_2 x) \end{aligned}$$

and the case construct as:

$$\begin{aligned} \text{case} & : \quad \forall \beta, \gamma, \alpha. (\beta + \gamma) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\gamma \rightarrow \alpha) \rightarrow \alpha \\ \text{case} & = \quad \Lambda \beta, \gamma, \alpha. \lambda s, f_1, f_2. s [\alpha] f_1 f_2 \end{aligned}$$

That is, case is just an eta-expansion of sorts for the underlying sum representation.

We can encode unit, as  $\forall \alpha. \alpha \rightarrow \alpha$  and void (the empty type) as  $\forall \alpha. \alpha$ . It turns out that there's at most one (semantically-speaking) function that has the unit type, and no closed value that has the empty type, so these encodings are indeed faithful.

And of course, simpler things, like booleans and if can be defined in terms of sums (i.e., unit + unit) and case, and n-ary products and sums in terms of appropriately associated binary sums and products. So at this point, we've effectively encoded the core of ML modulo recursive types and recursive functions.

A limited class of recursive types, corresponding to inductive types (i.e., lists and trees) and a limited notion of recursion (primitive recursion) can be encoded. For instance, we can define list  $\tau$  as the following type:

$$\forall \alpha. (\tau \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$$

which should look familiar — it's the type of “foldr” for lists. Then the empty list (nil) is encoded as:

$$\begin{aligned} \text{nil} & : \quad \forall \beta. \text{list } \beta \\ \text{nil} & = \quad \Lambda \beta. \Lambda \alpha. \lambda f. \lambda a. a \end{aligned}$$

and cons is encoded as:

$$\begin{aligned} \text{cons} & : \quad \forall \beta. \beta \rightarrow \text{list } \beta \rightarrow \text{list } \beta \\ \text{cons} & = \quad \Lambda \beta. \lambda x. \lambda y. \Lambda \alpha. \lambda f. \lambda a. f x a \end{aligned}$$

and it should be pretty obvious that we can code up foldr for lists (it's just an eta expansion in the same way that case was.)

Once you have foldr, you can code up head, tail, map, reverse, append, and all sorts of other list operations. And because we can represent numbers as lists of units, then append can be used to implement addition. From this, you can generate a limited class of “for-loops” that step from n down to 0. You can also implement subtraction and multiplication, an equality test on numbers and a bunch of other operations. But the one thing you cannot do is encode unbounded computation...

### 3.1 Existential Types

Another fun way to come up with encodings of certain data structures is to think of the type constructors logically. For instance, we can read  $\rightarrow$  as a form of implication, unit as true, void as false, products as conjunction, and sums as disjunction, etc. For example, consider a function with type  $(\tau_1 + \tau_2) \rightarrow \tau$ . Treating the arrow as a form of implication, this is similar to  $\neg(\tau_1 + \tau_2) + \tau$ . We can eliminate the negation by an application of DeMorgan's law to get  $(\neg\tau_1 \times \neg\tau_2) + \tau$  and then distribute the sum across the product to get:  $(\neg\tau_1 + \tau) \times (\neg\tau_2 + \tau)$  and finally introduce implication again to arrive at  $(\tau_1 \rightarrow \tau) \times (\tau_2 \rightarrow \tau)$ . This tells us that lurking inside every case-expression on a  $\tau_1 + \tau_2$ , is a pair of functions, one of which takes a  $\tau_1$  and one of which takes a  $\tau_2$ .

A very interesting thing is to think about  $\neg(\forall \alpha. \neg(\tau))$ . In some sense, this is  $\exists \alpha. \tau$ . That suggests that we can encode existential types using a combination of  $\forall$  and arrows. Indeed, an existential  $\exists \alpha. \tau$  can be encoded as  $\forall \beta. (\forall \alpha. \alpha \rightarrow \beta) \rightarrow \tau$ .

For example, consider the type  $\exists \alpha. \{x:\alpha, \text{inc}:\alpha \rightarrow \alpha\}$  where I've taken the liberty of using ML-style records. This is a value that abstracts some type  $\alpha$ , carries with it a value  $x$  of type  $\alpha$ , and an operation  $\text{inc}$  that takes

and returns an  $\alpha$ . Whether  $\alpha$  is int or bool, or a pair of ints, etc. isn't revealed. In this sense, an existential is an infinite sum indexed by type, in the same sense that a universal is an infinite product indexed by type. This is the essence of an object with a private instance variable and a method.

Furthermore, I could have a whole list of such existential values, and each one of them could have a different type. That is, when we have general  $\forall$  quantifiers, we not only get homogeneous data structures, but also heterogeneous data structures. We'll talk more about existentials later, but it's important to note that in some sense, we get the critical notion of a "first-class object" or module as soon as we have  $\forall$ .

## 4 An Attempt at Denotational Semantics

Suppose we now wanted to construct a denotational semantics for System-F, similar to our set-theoretic semantics for the simply-typed lambda calculus. Our first step would be to define an interpretation for types as in:

$$\begin{aligned} \mathcal{T}[\text{int}] &= Z \\ \mathcal{T}[\tau_1 \rightarrow \tau_2] &= \mathcal{T}[\tau_1] \rightarrow \mathcal{T}[\tau_2] \\ \mathcal{T}[\alpha] &= ??? \\ \mathcal{T}[\forall\alpha.\tau] &= ??? \end{aligned}$$

Oops. What do we do about type variables? We need to parameterize the interpretation by an environment that maps type variables to sets (just as we parameterized the term interpretation by an environment that maps term variables to elements of sets.) So formally, we should define the interpretation on well-kinded types (i.e., the judgment  $\Delta \vdash \tau : \star$ ). Here, I'll assume we have an environment  $\delta$  that maps type variables in  $\Delta$  (with kind  $\star$ ) to sets:

$$\begin{aligned} \mathcal{T}[\Delta \vdash \text{int} : \star]\delta &= Z \\ \mathcal{T}[\Delta \vdash \tau_1 \rightarrow \tau_2 : \star]\delta &= (\mathcal{T}[\tau_1]\delta) \rightarrow (\mathcal{T}[\tau_2]\delta) \\ \mathcal{T}[\Delta \vdash \alpha : \star]\delta &= \delta(\alpha) \\ \mathcal{T}[\Delta \vdash \forall\alpha.\tau : \star] &= ??? \end{aligned}$$

That takes care of the first problem. Now our temptation at this point is to interpret quantified types as a giant, infinite product:

$$\mathcal{T}[\Delta \vdash \forall\alpha.\tau : \star]\delta = \prod_S(\mathcal{T}[\tau]\delta[\alpha \mapsto S])$$

The intuition is that a polymorphic function, like the identity with type  $\forall\alpha.\alpha \rightarrow \alpha$  can be thought of as an object that has type  $\text{int} \rightarrow \text{int}$  *and*  $(\text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{int}$  *and* an infinite number of other types where we substitute any type we like for  $\alpha$ . Alternatively, we can think of it as being an infinite tuple of functions, that is indexed by type. We can pull out the the right function for a given type by projecting the tuple at the appropriate index.

The question is, what universe of sets is  $S$  drawn from? We might try to define:

$$\mathcal{U} = \bigcup_{\tau} (\mathcal{T}[\tau])$$

and draw  $S$  from this universe. But note that we can't define  $\mathcal{U}$  until we've defined  $\mathcal{T}[\forall\alpha.\tau]$  and we can't define this until we've defined  $\mathcal{U}$ ! Thus, we can see that there is no (easy) set-theoretic interpretation of our quantifiers.

One way around this problem is to *stratify* the types into two categories: Those without quantifiers (so-called monotypes) and those with (so-called polytypes or type-schemes). If we first define our interpretation of monotypes, and then define our interpretation of polytypes, only allowing type variables to range over monotypes, then we can break the nasty cycle. This approach, termed *predicative* polymorphism, is closest to the spirit of ML and is explored in the next section.

An alternative is to somehow *a priori* define a universe that's big enough to include the interpretation of all of our types (including the quantified types), and then let our quantifiers range over this universe. This is the essence behind the approach taken by Girard in constructing a model for the standard, *impredicative* polymorphic lambda calculus.

## 4.1 Predicative Polymorphism

In this section, we’re going to define a variant of the polymorphic lambda calculus that is *predicative* in the sense that quantifiers will only range over first-order types. That will allow us to define an easy set-theoretic model.

We begin by introducing two kinds,  $\mathbf{M}$  and  $\mathbf{P}$  which are used to classify mono-types and poly-types respectively:

$$\text{(kinds)} \quad \kappa ::= \mathbf{M} \mid \mathbf{P}$$

and a syntactic distinction between mono and poly-types:

$$\begin{aligned} \text{(mono-types)} \quad \tau & ::= b \mid \tau_1 \rightarrow \tau_2 \mid \alpha \\ \text{(poly-types)} \quad \sigma & ::= \uparrow \tau \mid \forall \alpha. \sigma \end{aligned}$$

We’re using an explicit injection “ $\uparrow \tau$ ” to allow us to treat every mono-type as if it were a poly-type. An alternative is to introduce some form of implicit sub-kinding, but this introduces a number of problems when you go to formalize (i.e., mechanize) the meta-theory. So for now, I’m making the injection explicit.

Our kinding contexts,  $\Delta$ , will only allow us to abstract over mono-types:

$$\Delta ::= \bullet \mid \Delta, \alpha : \mathbf{M}$$

We then change the kinding rules for types to draw the appropriate distinctions between mono-types and poly-types. To make the distinctions clearer, I’ll use  $\tau$  to range over mono-types and  $\sigma$  to range over poly-types throughout the rest of the discussion.

$$\begin{aligned} & \Delta \vdash b : \mathbf{M} \\ & \Delta \vdash \alpha : \Delta(\alpha) \\ & \frac{\Delta \vdash \tau_1 : \mathbf{M} \quad \Delta \vdash \tau_2 : \mathbf{M}}{\Delta \vdash \tau_1 \rightarrow \tau_2 : \mathbf{M}} \\ & \frac{\Delta \vdash \tau : \mathbf{M}}{\Delta \vdash \uparrow \tau : \mathbf{P}} \\ & \frac{\Delta, \alpha : \mathbf{M} \vdash \sigma : \mathbf{P}}{\Delta \vdash \forall \alpha. \sigma : \mathbf{P}} \quad (\alpha \notin \Delta) \end{aligned}$$

Notice that since  $\Delta$  only ranges over mono-types, the second rule is guaranteed to produce a mono-type. Notice also that we can inject mono-types into the space of poly-types using the second-to-last rule. Finally, the essence of the change is embodied by the last rule which restricts the quantified variable  $\alpha$  to range over mono-types.

Also notice that these rules prevent us from forming types of the form  $\sigma_1 \rightarrow \sigma_2$ . That is, all poly-types are of the form  $\forall \alpha_1. \dots \forall \alpha_n. \tau$  (i.e., in prenex form.) This is not strictly necessary to remain predicative—we could introduce a distinct rule that allows us to form arrows at the poly-type level:

$$\frac{\Delta \vdash \sigma_1 : \mathbf{P} \quad \Delta \vdash \sigma_2 : \mathbf{P}}{\Delta \vdash \sigma_1 \Rightarrow \sigma_2 : \mathbf{P}}$$

I’ve used a distinct constructor ( $\Rightarrow$ ), but it’s also possible to just overload the normal arrow. I’m going to forgo this here to try to keep things closer to ML (and to simplify the denotational semantics.) But the big point is that it is not necessary to impose the prenex restriction and remain predicative.

Next, we need to modify our typing rules for terms. First, for instantiation, we need to restrict the type to which we’re applying a  $\forall$ -term, so that it is a mono-type as in:

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha. \sigma \quad \Delta \vdash \tau : \mathbf{M}}{\Delta; \Gamma \vdash e : \sigma[\tau/\alpha]}$$

and for type abstraction, we need to restrict the kind of the type-variable to be a mono-type:

$$\frac{\Delta, \alpha:\mathbf{M}; \Gamma \vdash e : \sigma}{\Delta; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\sigma} \quad (\alpha \notin \Delta)$$

But this is not enough. We also need to restrict the type of arguments for  $\lambda$ -expressions so that they are mono-types in order to meet the requirements of the prenex restriction:

$$\frac{\Delta \vdash \tau_1 : \mathbf{M} \quad \Delta; \Gamma, x:\tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x:\tau_1.e : \tau_1 \rightarrow \tau_2} \quad (x \notin \Gamma)$$

Notice also that for the  $\tau_1 \rightarrow \tau_2$  to be well-formed, we must rig our typing judgment to ensure that the type of  $e$  is always a mono-type. Alas, this makes it clear that with the language we have thus far, we can *never* introduce polymorphic functions (because they have poly-types). Even if we could introduce them, we could never bind them to a variable because variables are restricted to mono-types. So we'd be forced to just immediately apply them to types until we got out a mono-type (at which point, there's not much reason to introduce the polymorphism at all!)

The solution to these problems is to introduce distinctions at the term level between monotypes and polytypes:

$$\begin{aligned} \text{(mono terms)} \quad e &::= c \mid x \mid \lambda x:\tau.e \mid e_1 e_2 \mid \cdots \mid \downarrow P \mid \text{let } X = P \text{ in } e \\ \text{(poly terms)} \quad P &::= \uparrow e \mid X \mid \Lambda\alpha.P \mid P\tau \\ \text{(mono values)} \quad v &::= c \mid \lambda x:\tau.e \\ \text{(poly values)} \quad V &::= \uparrow v \mid \Lambda\alpha.P \end{aligned}$$

Here, we use  $X$  to range over a distinct class of variables that will range over terms of polymorphic type.

From an operational standpoint, the let-expression behaves just as expected:

$$\frac{P \Downarrow V \quad e[V/X] \Downarrow v}{\text{let } X = P \text{ in } e \Downarrow v}$$

and type application remains unchanged except that it's operating over poly-terms instead of mono-terms:

$$\frac{P \Downarrow \Lambda\alpha.P' \quad P'[\tau/\alpha] \Downarrow V}{P\tau \Downarrow V}$$

The up and down terms are just technical devices that are used to keep type-checking syntax-directed. The form  $\uparrow e$  coerces a mono-term to a poly term. The form  $\downarrow P$  downcasts a poly-term to a mono-term, but is only defined when  $P$  evaluates to a term of the form  $\uparrow v$ . Thus, we have:

$$\frac{P \Downarrow \uparrow v}{\downarrow(\uparrow v) \Downarrow v}$$

Note that  $\downarrow(\Lambda\alpha.P)$  is stuck—the downcast can only return a mono value. The typing rules should ensure that this downcast never fails.

Now we need to extend our contexts to account for the polymorphic variables in scope:

$$\Gamma ::= \bullet \mid \Gamma, x:\tau \mid \Gamma, X:\sigma$$

with the understanding that a context  $\Gamma$  is well-formed with respect to  $\Delta$  if each  $x$  in  $\Gamma$  maps to a mono-type (w.r.t.  $\Delta$ ) and each  $X$  maps to a poly-type. We can formalize this with a judgment  $\Delta \vdash \Gamma$  as follows:

$$\frac{\Delta \vdash \bullet \quad \Delta \vdash \Gamma \quad \Delta \vdash \tau : \mathbf{M}}{\Delta \vdash \Gamma, x:\tau} \quad (x \notin \Gamma)$$

$$\frac{\Delta \vdash \Gamma \quad \Delta \vdash \sigma : \mathbf{P}}{\Delta \vdash \Gamma, X:\sigma} \quad (X \notin \Gamma)$$

Finally, let us re-cast the typing rules for terms. We will need two judgments: one for mono-terms ( $\Delta; \Gamma \vdash_{\mathbf{M}} e : \tau$ ) and one for poly-terms ( $\Delta; \Gamma \vdash_{\mathbf{P}} e : \sigma$ ). We'll rig the definition of the judgments so that  $\Delta \vdash \Gamma$ ,  $\Delta \vdash \tau : \mathbf{M}$ , and  $\Delta \vdash \sigma : \mathbf{P}$ . The monomorphic judgment is defined as follows:

$$\begin{array}{c} \Delta; \Gamma \vdash_{\mathbf{M}} x : \Gamma(x) \\ \\ \Delta; \Gamma \vdash_{\mathbf{M}} c : b \\ \\ \frac{\Delta \vdash \tau_1 : \mathbf{M} \quad \Delta; \Gamma, x:\tau_1 \vdash_{\mathbf{M}} e : \tau_2}{\Delta; \Gamma \vdash_{\mathbf{M}} \lambda x:\tau_1. e : \tau_1 \rightarrow \tau_2} \quad (x \notin \Gamma) \\ \\ \frac{\Delta; \Gamma \vdash_{\mathbf{M}} e_1 : \tau' \rightarrow \tau \quad \Delta; \Gamma \vdash_{\mathbf{M}} e_2 : \tau'}{\Delta; \Gamma \vdash_{\mathbf{M}} e_1 e_2 : \tau} \\ \\ \frac{\Delta; \Gamma \vdash_{\mathbf{P}} P : \sigma \quad \Delta; \Gamma, X:\sigma \vdash_{\mathbf{M}} e : \tau}{\Delta; \Gamma \vdash_{\mathbf{M}} \text{let } X = P \text{ in } e : \tau} \\ \\ \frac{\Delta; \Gamma \vdash_{\mathbf{P}} P : \uparrow \tau}{\Delta; \Gamma \vdash_{\mathbf{M}} \downarrow P : \tau} \end{array}$$

and the polymorphic judgment is defined as:

$$\begin{array}{c} \Delta; \Gamma \vdash_{\mathbf{P}} X : \Gamma(X) \\ \\ \frac{\Delta; \Gamma \vdash_{\mathbf{M}} e : \tau}{\Delta; \Gamma \vdash_{\mathbf{P}} \uparrow e : \uparrow \tau} \\ \\ \frac{\Delta, \alpha:\mathbf{M}; \Gamma \vdash P : \sigma}{\Delta; \Gamma \vdash \lambda \alpha. P : \forall \alpha. \sigma} \quad (\alpha \notin \Delta) \\ \\ \frac{\Delta; \Gamma \vdash P : \forall \alpha. \sigma \quad \Delta \vdash \tau : \mathbf{M}}{\Delta; \Gamma \vdash e : \sigma[\tau/\alpha]} \end{array}$$

Whew! That looks harder than it really needs to be. I've tried to make the definitions something that could be readily encoded into Twelf which accounts for some of the apparent complexity. In particular, if we took an implicit approach to the sub-kinding, the rules would be greatly simplified, but making it explicit keeps things syntax directed ensuring that there is at most one typing proof for a given term and context.

## 4.2 A model

By stratifying the mono and poly-types, we can now construct a suitable model. I'll just sketch the relevant bits here. The first step is to define an interpretation for *closed* mono types:

$$\begin{array}{lcl} \mathcal{M}[\text{int}] & = & Z \\ \mathcal{M}[\tau_1 \rightarrow \tau_2] & = & \mathcal{M}[\tau_1] \rightarrow \mathcal{M}[\tau_2] \end{array}$$

This allows us to define our first universe  $\mathcal{U}_0$  as follows:

$$\mathcal{U}_0 = \bigcup_{\tau} \mathcal{M}[\tau]$$

Next, we can lift the interpretation to open types by parameterizing the interpretation by a suitable environment  $\delta$  mapping type variables to subsets of  $\mathcal{U}_0$ :

$$\begin{aligned}\mathcal{M}[\Delta \vdash \alpha : \Delta(\alpha)]\delta &= \delta(\alpha) \\ \mathcal{M}[\Delta \vdash \text{int} : \mathbf{M}]\delta &= Z \\ \mathcal{M}[\Delta \vdash \tau_1 \rightarrow \tau_2 : \mathbf{M}]\delta &= (\mathcal{M}[\Delta \vdash \tau_1 : \mathbf{M}]\delta) \rightarrow (\mathcal{M}[\Delta \vdash \tau_2 : \mathbf{M}]\delta)\end{aligned}$$

It's relatively easy to see that this definition coincides with the previous one for closed types, regardless of the environment.

We can then define an interpretation for poly-types as follows:

$$\begin{aligned}\mathcal{P}[\Delta \vdash \uparrow \tau : \mathbf{P}]\delta &= \mathcal{M}[\Delta \vdash \tau : \mathbf{M}]\delta \\ \mathcal{P}[\Delta \vdash \forall \alpha. \sigma : \mathbf{P}]\delta &= \Pi S \subseteq \mathcal{U}_0. \mathcal{P}[\Delta, \alpha : \mathbf{M} \vdash \sigma : \mathbf{P}](\delta[\alpha \mapsto S])\end{aligned}$$

You may wonder why I chose  $S$  to range over any subset of  $\mathcal{U}_0$ . Why not restrict it to one of those sets that corresponds to a particular monotype? The reason is that we can then use the model to reason about the behavior of polymorphic functions when applied not only to the built-in types, but also any type we wish to form. That is, we can start to model and think about user-defined abstract data types.

Consider just for a second the polymorphic type  $\forall \alpha. \alpha$ . Suppose  $x \in \mathcal{P}[\forall \alpha. \alpha]$ . That means that for any subset  $S$  of  $\mathcal{U}_0$ , we have  $x(S) \in S$ . Now pick  $S$  to be the empty-set... Oops. There is no such  $x$ ! So that tells us that  $\forall \alpha. \alpha$  is an empty type.

Now consider an  $x \in \mathcal{P}[\forall \alpha. \alpha \rightarrow \alpha]$ . That means that for any  $S$ ,  $x(S) \in S \rightarrow S$ . Now pick any (well-typed) term  $e$  and consider the singleton set  $S = \{\mathcal{E}[e]\}$ . We are guaranteed that  $x(S)(\mathcal{E}[e]) \in \{\mathcal{E}[e]\}$ . Thus,  $x(S)(\mathcal{E}[e]) = \mathcal{E}[e]$ . Therefore,  $x$  behaves like the identity function on all terms.

## 5 Impredicative Polymorphism

In the last section, we discussed a model for the predicative variant of the polymorphic lambda calculus. By stratifying the types into two levels, and restricting type variables to range over only the first universe, we were able to come up with a simple, set-theoretic model.

However, in practice, predicativity can turn out to be a big constraint. For instance, it's common to model objects or ADTs using existential types (which in turn can be encoded with universal types.) In that setting, type variables are used to abstract hidden components of the object. More often than not, these hidden components are themselves objects, and thus have quantified types. That's not to say that we couldn't find *some* stratification, but rather, we may have to come up with a different one for each application.

So the question is whether we can construct a model for the impredicative version of the polymorphic lambda calculus and let type variables range over any type, including quantified types. The negative result, due to Reynolds, is that there is no set-theoretic model for the impredicative language (i.e., where we interpret arrow types as set-theoretic functions, and universal types as dependent products.) We simply always run into a cardinality problem.

The positive result (really due to Girard, but simplified and perhaps rediscovered by Tait, Statman, and others) is that there is a useful model, just not one where we use set-theoretic interpretations of arrows and products.

The basic idea is to instead use sets of equivalence classes of (undecorated) terms to interpret types. Intuitively, we think of  $\mathcal{T}[\text{int}]$  as the set of all equivalence classes  $[e]$  such that each term  $e' \in [e]$  computes the same integer. And we think of  $\mathcal{T}[\tau_1 \rightarrow \tau_2]$  as the set of  $[e]$  that when applied to equivalent terms in  $\mathcal{T}[\tau_1]$ , produce equivalent terms in  $\mathcal{T}[\tau_2]$ . And then we interpret universal types as indexed *intersections* of certain sets of equivalence classes called "candidates".

The trick is to define a space of "candidates" before we define the interpretation of types (else we'll end up with a cycle in the definition again) and to make sure of two things: (1) the candidates have proper closure properties so that they behave like types, (2) the candidates include all of the sets of equivalence classes that correspond to what will be our interpretation of types.

## 5.1 System F is terminating

To make these ideas a bit more concrete, I'm going to do a simpler, but closely related construction where we argue that every term in impredicative System F terminates. In essence, I'm going to be constructing a unary logical relation on System F terms, but it's possible to extend this construction to a binary logical relation corresponding to a notion of partial equivalence, or to consider reduction instead of evaluation. (Recall that we did this sort of generalization for the simply-typed lambda calculus.)

For the purposes of this discussion, it's easier to work with an implicitly typed, call-by-value version of System F. So, let us define the class of erased terms and values as:

$$\begin{aligned} e &::= e \mid c \mid x \mid \lambda x.e \mid e_1 e_2 \\ v &::= c \mid \lambda x.e \end{aligned}$$

We can relate the explicit and implicit versions with an erasure function:

$$\begin{aligned} \text{erase}(c) &= c \\ \text{erase}(x) &= x \\ \text{erase}(\lambda x:\tau.e) &= \lambda x.\text{erase}(e) \\ \text{erase}(e_1 e_2) &= \text{erase}(e_1) \text{erase}(e_2) \\ \text{erase}(\Lambda\alpha.e) &= \text{erase}(e) \\ \text{erase}(e \tau) &= \text{erase}(e) \end{aligned}$$

though there's a technical hitch that  $\Lambda\alpha.e$  is always a value, but its erasure may not be. However, if we show that  $e$  always evaluates to a value, then this won't turn out to be a real problem, so I'm going to ignore this minor point for now.

Let VAL be the set of all (erased) values. We're going to interpret types as subsets of VAL, or more properly, as sets of expressions which always evaluate to values.

$$\begin{aligned} \mathcal{V} &: \text{Type} \rightarrow (\text{TypeVar} \rightarrow 2^{\text{VAL}}) \rightarrow 2^{\text{VAL}} \\ \mathcal{V}[\delta] &= \{c\} \\ \mathcal{V}[\alpha] &= \delta(\alpha) \\ \mathcal{V}[\tau_1 \rightarrow \tau_2] &= \{v \mid FV(v) = \emptyset \wedge \forall v' \in \mathcal{V}[\tau_1].\delta.(v v') \in \mathcal{C}[\tau_2]\delta\} \\ \mathcal{V}[\forall\alpha.\tau] &= \bigcap_{S \in 2^{\text{VAL}}} \mathcal{V}[\tau]\delta[\alpha \mapsto S] \\ \mathcal{C}[\tau] &= \{e \mid FV(e) = \emptyset \wedge e \Downarrow v \wedge v \in \mathcal{V}[\tau]\} \\ \mathcal{V}[\Gamma] &= \{\gamma \mid \gamma(x) \in \mathcal{C}[\Gamma(x)]\delta\} \end{aligned}$$

The key reason this definition is well-founded is that when we formed the intersection for  $\forall$ -types, we chose from the set of all subsets of values—a space that we had already defined. In other words, our candidates here are sets of values.

With these definitions, it's possible to show the following theorem:

If  $\Delta; \Gamma \vdash e : \tau$ , then for all  $\delta$  and  $\gamma \in \mathcal{V}[\Gamma]\delta$ ,  $\gamma(e) \in \mathcal{C}[\tau]\delta$ .

The proof is by induction on the typing derivation. A key step in making sure that our quantifiers are big enough is to show that for every type,  $\mathcal{V}$  maps that type to a candidate (i.e., a set of closed values) but that's fairly obvious.

Let's examine a couple of interesting cases. For  $\forall$ -elimination, the rule for the erased calculus looks like this:

$$\frac{\Delta; \Gamma \vdash e : \forall\alpha.\tau \quad \Delta \vdash \tau' : \star}{\Delta; \Gamma \vdash e : \tau[\tau'/\alpha]}$$

Pick a  $\delta$  and  $\gamma \in \mathcal{V}[\Gamma]\delta$ . We must show  $\gamma(e) \in \mathcal{C}[\tau[\tau'/\alpha]]\delta$ . By induction, we know that  $\gamma(e) \in \mathcal{C}[\forall\alpha.\tau]\delta$ . So it suffices for us to show that  $\mathcal{C}[\forall\alpha.\tau]\delta \subseteq \mathcal{C}[\tau[\tau'/\alpha]]\delta$ . From the definition of  $\mathcal{C}$  and  $\mathcal{V}$ , this would follow

if we can establish the following substitution property:

$$\mathcal{V}[\tau]\delta[\alpha \mapsto \mathcal{V}[\tau']\delta] = \mathcal{V}[\tau[\tau'/\alpha]]$$

which turns out to be a fairly straightforward induction on  $\tau$ .

For  $\forall$ -introduction, the rule looks like this:

$$\frac{\Delta, \alpha : \star; \Gamma \vdash e : \tau \quad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash e : \forall \alpha. \tau} \quad (\alpha \notin \Delta)$$

The condition  $\Delta \vdash \Gamma$  is meant to ensure that we picked a “fresh”  $\alpha$  which cannot occur free in the environment. That is, since  $\alpha$  is not in  $\Delta$ , and  $\Delta \vdash \Gamma$ , then no type in  $\Gamma$  can mention  $\alpha$ .

Again, pick a  $\delta$  and suitable  $\gamma$ , and we must show that  $e \in \mathcal{C}[\forall \alpha. \tau]\delta$ . The following lemma will come in handy:

If  $\Delta \vdash G$  and  $\gamma \in \mathcal{V}[\Gamma]\delta$ , then for all  $\alpha \notin \Delta$  and  $S \subseteq \text{VAL}$ ,  $\gamma \in \mathcal{V}[\Gamma]\delta[\alpha \mapsto S]$ .

This lemma tells us that if  $\gamma$  only depends upon  $\Delta$ , then if we extend  $\delta$  with  $\alpha \mapsto S$ , it won't change the interpretation. That is, it's saying that we can weaken our contexts without any adverse effects.

Now by induction, we have that for all  $\delta'$  and  $\gamma'$  in  $\mathcal{V}[\Gamma]\delta'$ , that  $\gamma'(e) \in \mathcal{C}[\tau]\delta'$ . By the lemma, for all  $S$ , we know that  $\gamma \in \mathcal{V}[\Gamma]\delta[\alpha \mapsto S]$ . So it follows that  $\gamma(e) \Downarrow v \in \mathcal{V}[\tau]\delta[\alpha \mapsto S]$ . Therefore,  $v \in \mathcal{V}[\forall \alpha. \tau]\delta$  and hence  $\gamma(e) \in \mathcal{C}[\forall \alpha. \tau]\delta$ .

The rest of the cases follow in a straightforward fashion.

As in the predicative model, we can use the interpretation to argue about what terms must be or not be in a given type. For instance:

$$\mathcal{V}[\forall \alpha. \alpha] = \bigcap_S .S$$

where  $S$  is drawn from subsets of VAL. But the empty set is such an  $S$ , so the intersection must be empty!

As another example:

$$\mathcal{V}[\forall \alpha. \alpha \rightarrow \alpha] = \bigcap_S .\{v \mid \forall v_1 \in S. v v_1 \Downarrow v_2 \in S\}$$

Here, when we pick  $S$  to be empty, the set:

$$\{v \mid \forall v_1 \in S. v v_1 \Downarrow v_2 \in S\}$$

is just VAL because there is no  $v_1$  in  $S$ , so every value trivially satisfies the constraints. But we must look at all possible sets of closed values. Notice that for any  $v'$ , picking  $S = \{v'\}$  we have:

$$\{v \mid \forall v_1 \in \{v'\}. v v_1 \Downarrow v_2 \in \{v'\}\} = \{v \mid v v' \Downarrow v'\}$$

That is,  $v$  must behave like the identity function. So we know that if there's anything in  $\forall \alpha. \alpha \rightarrow \alpha$ , it must behave like the identity function. Of course, the intersection could be empty, so we need to exhibit at least one  $v$  that behaves like the identity function. That is, we must show  $\lambda x. x \in \mathcal{V}[\forall \alpha. \alpha \rightarrow \alpha]$ . But this is easy since we can construct a proof that  $\vdash \lambda x. x : \forall \alpha. \alpha \rightarrow \alpha$ .

So to summarize, it is possible to construct a model that avoids the circularity, but it hinges on constructing some suitable universe up front, that over-approximates the interpretation of the types. In this example, we picked sets of closed values. When we interpret for-all types, we are forced to intersect over all of the possible approximates which is actually a much stronger condition than what is required. For instance, our interpretation of  $\forall$  ranges over some types that we can't even write down (e.g., the positive integers or singleton types.) However, this is a feature in that we can use these non-standard “types” to argue about the properties of polymorphic functions.

## 6 Representation Independence

The power of parametric polymorphism starts to become really apparent when we interpret types not as unary relations, but rather as binary relations. Consider, for a moment a simple signature such as:

```
type bool
val true : bool
val false : bool
val if : All 'a.bool -> 'a -> 'a -> 'a
```

We can represent a "structure" that implements this signature using an existential:

$$\exists t.(t \times t \times \forall \alpha.t \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$$

Eliminating the existential, we see that a client of the package must look like this:

$$(\forall t.t \times t \times \forall \alpha.t \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \tau$$

for some  $\tau$ . Now consider two implementations of the signature:

1. we represent booleans using integers with 1 for true, and 0 for false.
2. we represent booleans using integers with 0 for false, and any non-zero value for true.

Can the client tell which implementation we're using? Intuitively, the answer is no—this is the power of type abstraction—one of representation independence. We can prove that indeed, the client will behave the same by setting up a type-indexed *binary* relation:

$$\begin{aligned} \mathcal{V}[[b]]\delta &= \{(c, c)\} \\ \mathcal{V}[[\alpha]]\delta &= \delta(\alpha) \\ \mathcal{V}[[\tau_1 \rightarrow \tau_2]]\delta &= \{(v_1, v_2) \mid \forall (v'_1, v'_2) \in \mathcal{V}[[\tau_1]]\delta. (v_1 v'_1, v_2 v'_2) \in \mathcal{C}[[\tau_2]]\delta\} \\ \mathcal{V}[[\forall \alpha.\tau]]\delta &= \bigcap_{S \subseteq \text{VAL} \times \text{VAL}} \mathcal{V}[[\tau]]\delta[\alpha \mapsto S] \end{aligned}$$

Here, our type variables range over sets of pairs of values (i.e., relations).

Now suppose we want to prove that, for *any* client, they can't observe whether we're using implementation 1 or 2. It suffices to construct a binary logical relation that relates two  $t$  values  $i$  and  $j$  which captures the relation between our two possible implementations:

$$\mathcal{V}[[t]] = \{(i, j) \mid (i = 0 \wedge j = 0) \vee (i = 1 \wedge j \neq 0)\}$$

Because the client has a polymorphic type, it must respect this invariant because we can instantiate  $S$  with our chosen interpretation. If we pass in operations for our abstract type that respect the invariant, then the client will be forced to return only related values. At base type, the only related values are equal values. So in some sense, this construction captures the idea that clients will behave the same regardless of the implementation of an ADT.

This kind of "obliviousness" is particularly useful when it comes to proving other kinds of results, such as non-interference—a particularly strong form of secrecy.

## 7 Implementing Polymorphism

We've discussed the semantics of polymorphic languages such as System-F, but not the implementation issues. There are about 5 ways to handle polymorphism which I'll briefly discuss below:

## 7.1 Statically Monomorphize the code

This is the approach taken by C++ and MLton. In languages such as SML, the type system forces a prenex restriction on quantifiers:

$$\begin{array}{ll} \text{(monotypes)} & \tau ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \dots \\ \text{(polytypes)} & \sigma ::= \tau \mid \forall \alpha. \sigma \end{array}$$

which effectively makes the polymorphic definitions second-class (i.e., note that functions can't take/return a polytype, and that products only range over monotypes, etc.) Another way to see this is that the only polymorphic definitions are introduced by "let" expressions. We can simply inline/duplicate the let declaration everywhere it is used and do the reductions of the type applications at compile time:

$$\begin{array}{l} \text{let } x = e_1 \text{ in } e \longmapsto e[e1/x] \\ (\Lambda \alpha_1, \dots, \alpha_n. e) \tau_1 \dots \tau_n \longmapsto e[\tau_i/\alpha_i] \end{array}$$

Of course, we risk blowing up the code (because we're duplicating it). In practice, this can be avoided by doing CSE on the type applications before doing the inlining/duplication and by specializing the code once for each unique type instantiation, so this is not a real problem in practice. (Indeed, TIL, MLton, and HUGS have all tried this and shown that you don't get a code blow up. I wonder if this is related to the fact that although ML-style type inference is something like EXP-time worst case, in practice, it's linear...)

The key advantage of monomorphizing the code is that we can compile the resulting language like Pascal. In particular, we need not pick a uniform representation for values. For instance, floats can be represented as 64-bit values and passed in fp registers, while ints can be represented as 32-bit values and passed in gp registers. If we pass a float to "the" identity function, then we'll really end up passing it to a specialized version of the identity function which expects its argument in an fp register, and if we pass an int to "the" identity function then we'll really pass it to a specialized version that accepts the argument in a gp register.

The key disadvantage of monomorphizing code is that destroys separate compilation. In particular, it's impossible to compile a polymorphic library (say, the list library) separate from the clients. Another way to say this is that we're forced to expose the implementation in the interfaces (as in C++ and Ada) so that if the implementation changes, we may have to recompile the clients. In practice, this hurts if you change something very deep in the library dependency graph. Note that one option that can minimize these costs is to cache the instantiations of polymorphic code so that we don't necessarily have to generate specialized code for each client. For instance, we might specialize the list library to integers, floats, and (say) pointers eagerly and then a client wouldn't have to worry about paying the cost of generating and compiling these specializations.

Another disadvantage of monomorphization is that we're really forced to have 2nd-class polymorphism which I believe is way too limiting. For instance, it precludes things like existentials, GADTs, and data structures like you'll find in Chris Okasaki's book (e.g., rose trees) that demand support for polymorphic recursion.

## 7.2 Monomorphize at JIT time

This is the approach taken by languages such as C# (today). To avoid the problems mentioned above, an alternative is to use a JIT compiler to generate the specialized code at run-time. Again, caching is important to avoid generating the code over and over again. The key advantages of this approach are that (a) we get all of the wins of static monomorphization in terms of native representations, (b) we are no longer limited to second-class polymorphic functions, and (c) we can support the illusion of separate compilation. The downside is that we don't really get separate compilation, but rather just delay when we do the compilation and that latency may be of concern. Another downside is that it's complicated and only works in relatively "heavyweight" environments where we can afford to have a JIT around at run-time (i.e., not sensors). Nonetheless, this is probably the best of the options in terms of raw performance and clearly where things are heading.

### 7.3 Use a uniform representation

This is the “classical” approach used by O’Caml, Java, and languages such as Modula-3 as well as older versions of SML/NJ (circa 0.93).

Consider the identity function:  $\Lambda\alpha.\lambda x:\alpha.x$ . If we ensure that all values passed to it have the same size and calling convention, then we can use one piece of code. For instance, we may “box” floats (i.e., represent them by a pointer to a cell holding the actual float) to ensure that they can be represented in 32-bits and be passed in a gp register.

There are obvious disadvantages to boxing in that we pay the price of constructing the box, and reading out the real value. (There are subtle disadvantages too, such as the fact that Java does not lift “equality” on boxed scalars in the right way...)

In languages such as Modula-3 (and older versions of Java), there is another disadvantage in that we are forced to make a distinction at the source level between the “uniform” types (i.e., reference types) and other types (e.g., int, float, etc.) Programmers must explicitly coerce non-uniform values to and from a uniform representation if they want to pass such values to a polymorphic function.

In languages such as O’Caml, there are no non-uniform types. Rather, the compiler is responsible for making sure that every type is uniform. Now at first blush, you might think that we only need to insert coercions around polymorphic functions, and indeed this is true, but it’s a bit more subtle than you think and doesn’t always work (see below.) So most compilers just made *every* value uniform (e.g., a 32-bit gp value.) That meant that floats were always boxed for instance. The advantage of this approach is that polymorphic code is fast and programmers don’t have to deal with anything. The disadvantage is that monomorphic code pays the price...

### 7.4 Leroy-Style Coercions

To see the subtlety with coercions, consider the following attempted translation from a source language where types are meant to be uniform, to a target language where they aren’t.

$$\begin{array}{l}
 \text{Source types } \tau ::= \alpha \mid \text{int} \mid \text{float} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \\
 \sigma ::= \forall\alpha_1, \dots, \alpha_n. \tau \\
 \\
 \text{Target types } \mu ::= \alpha \mid \text{int} \mid \mu_1 \rightarrow \mu_2 \mid \mu_1 \times \mu_2 \mid \text{Float} \\
 \tau ::= \mu \mid \text{float} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \\
 \sigma ::= \forall\alpha_1, \dots, \alpha_n. \tau
 \end{array}$$

The intention is that in the target language, Float is a boxed float, whereas float is an unboxed float. Furthermore, the type variables in the language range only over uniform types ( $\mu$ ). That allows the compiler to know, for instance,  $\text{sizeof}(\alpha)$  and  $\text{regtype}(\alpha)$  when emitting code.

Let us start our compiler by defining a type translation mapping source types to target types. The goal of the translation is to use floats by default and only use Float when we have to. We’ll assume the target language has primitives:

$$\begin{array}{l}
 \text{box} \quad : \text{float} \rightarrow \text{Float} \\
 \text{unbox} \quad : \text{Float} \rightarrow \text{float}
 \end{array}$$

to mediate the mismatch. The naive translation might look like this:

$$\begin{array}{l}
 T[\alpha] = \alpha \\
 T[\text{int}] = \text{int} \\
 T[\text{float}] = \text{float} \\
 T[\tau_1 \rightarrow \tau_2] = T[\tau_1] \rightarrow T[\tau_2] \\
 T[\tau_1 \times \tau_2] = T[\tau_1] \times T[\tau_2] \\
 T[\forall\alpha_1, \dots, \alpha_n. \tau] = \forall\alpha_1, \dots, \alpha_n. T[\tau]
 \end{array}$$

The problem with this is that when we get to translating polymorphic instantiation at the term level:

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha. \tau \quad \Delta \vdash \tau_1}{\Delta; \Gamma \vdash e \tau : \tau[\tau_1/\alpha]}$$

something will break. In particular, consider the identity function. At source level it has type  $\forall \alpha. \alpha \rightarrow \alpha$ . So we can conclude:

$$\frac{\Delta; \Gamma \vdash \text{id} : \forall \alpha. \alpha \rightarrow \alpha \quad \Delta \vdash \text{float}}{\Delta; \Gamma \vdash \text{id float} : \text{float} \rightarrow \text{float}}$$

Now we see what the problem is: at the target level, we can't instantiate a with float because it's not a uniform type. So that suggests that we need another type translation which produces a uniform type out of a source-level type:

$$\begin{aligned} U[\alpha] &= \alpha \\ U[\text{int}] &= \text{int} \\ U[\text{float}] &= \text{Float} \\ U[\tau_1 \rightarrow \tau_2] &= U[\tau_1] \rightarrow U[\tau_2] \\ U[\tau_1 \times \tau_2] &= U[\tau_1] \times U[\tau_2] \\ U[\forall \alpha_1, \dots, \alpha_n. \tau] &= \forall \alpha_1, \dots, \alpha_n. U[\tau] \end{aligned}$$

Then we could define the translation for instantiation thusly:

$$E[e \tau] = E[e] U[\tau]$$

For example,  $E[\text{id float}] = E[\text{id}] U[\text{float}] = E[\text{id}] \text{Float}$ . Now presumably,  $E[\text{id}] : T[\forall \alpha. \alpha \rightarrow \alpha] = \forall \alpha. T[\alpha \rightarrow \alpha]$ , so we can conclude that  $E[\text{id}] \text{Float} : T[\alpha \rightarrow \alpha][\text{Float}/\alpha] = \text{Float} \rightarrow \text{Float}$ .

More generally, if  $\Delta; \Gamma \vdash e : \forall \alpha_1, \dots, \alpha_n. \tau$  and  $\Delta \vdash \tau_1$ , then  $E[e] U[\tau_1] : T[\tau]([U[\tau_1]/\alpha_1])$ .

That seems good except that we want our translation to have the property that if  $\Delta; \Gamma \vdash e : \tau$ , then  $\Delta; T[\Gamma] \vdash E[e] : T[\tau]$ , and it is *not* the case that  $T[\tau]([U[\tau_1]/\alpha]) = T[\tau[\tau_1/\alpha]]$ .

For instance, at source level,  $\text{id float} : \text{float} \rightarrow \text{float}$  so at the target level, it should have type  $T[\text{float} \rightarrow \text{float}] = \text{float} \rightarrow \text{float}$ . But the translation we've used yields  $\text{Float} \rightarrow \text{Float}$ . Oops!

Notice that we could fix this problem by demanding that  $\Delta; U[\Gamma] \vdash E[e] : U[\tau]$  because  $U[\tau]([U[\tau_1]/\alpha]) = U[\tau[\tau_1/\alpha]]$ . That is, the  $U$  translation commutes with substitution. But also note that this would force:

$$E[3.14] = \text{box } 3.14$$

That is, every float expression would have to be boxed!

The fix, which Leroy noted, is that at polymorphic instantiation, what we need to do is construct a coercion  $S$  of type:

$$S : T[\tau]([U[\tau_1]/\alpha]) \rightarrow T[\tau_1/\alpha]$$

Then we can use  $S$  to coerce the polymorphic object to the right thing. For instance, to fix up the identity function we want  $E[\text{id float}]$  to yield something like:

$$\lambda x:\text{float}. (\text{unbox}(\text{id Float} (\text{box } x)))$$

This function has type  $\text{float} \rightarrow \text{float}$  (i.e.,  $T[\text{float} \rightarrow \text{float}]$ ). It boxes the argument and then unboxes the result. We can rewrite this as:

$$(\lambda f:\text{Float} \rightarrow \text{Float}. \text{unbox} \circ f \circ \text{box}) (\text{id Float})$$

and see that  $S$  for this case should be  $(\lambda f:\text{Float} \rightarrow \text{Float}. \text{unbox} \circ f \circ \text{box})$ .

More generally, we can define  $S$  indexed by the substitution we're trying to perform as follows:

$$\begin{aligned}
S[\alpha][\text{Float}/\alpha] &= \text{unbox} \\
S[\alpha_1][\mu/\alpha_2] &= \lambda x:\alpha_1.x && (\mu \neq \text{Float} \vee \alpha_1 \neq \alpha_2) \\
S[\text{int}][\mu/\alpha] &= \lambda x:\text{int}.x \\
S[\text{float}][\mu/\alpha] &= \lambda x:\text{float}.x \\
S[\tau_1 \times \tau_2][\mu/\alpha] &= \lambda p:T[\tau_1 \times \tau_2].(S[\tau_1][\mu/\alpha](\pi_1 p), S[\tau_2][\mu/\alpha](\pi_2 p)) \\
S[\tau_1 \rightarrow \tau_2][\mu/\alpha] &= \lambda f:T[\tau_1 \rightarrow \tau_2].S[\tau_2][\mu/\alpha] \circ f \circ G[\tau_1][\mu/\alpha]
\end{aligned}$$

$$\begin{aligned}
G[\alpha][\text{Float}/\alpha] &= \text{box} \\
G[\alpha_1][\mu/\alpha_2] &= \lambda x:\alpha_1.x && (\mu \neq \text{Float} \vee \alpha_1 \neq \alpha_2) \\
G[\text{int}][\mu/\alpha] &= \lambda x:\text{int}.x \\
G[\text{float}][\mu/\alpha] &= \lambda x:\text{float}.x \\
G[\tau_1 \times \tau_2][\mu/\alpha] &= \lambda p:T[\tau_1 \times \tau_2].(G[\tau_1][\mu/\alpha](\pi_1 p), G[\tau_2][\mu/\alpha](\pi_2 p)) \\
G[\tau_1 \rightarrow \tau_2][\mu/\alpha] &= \lambda f:T[\tau_1 \rightarrow \tau_2].G[\tau_2][\mu/\alpha] \circ f \circ S[\tau_1][\mu/\alpha]
\end{aligned}$$

The "S" stands for specialization and the "G" stands for generalization. Note that for functions, we need to be able to box arguments (i.e., generalize them) and unbox results. More generally, for positive occurrences of the type variable we're specializing, we need to use  $S$  to unbox, and for negative occurrences, we need to use  $G$  to box. You can check that these two coercions have the properties:

$$\begin{aligned}
S[T[\tau]][U[\tau_1]/\alpha] &: (T[\tau][U[\tau_1]/\alpha]) \rightarrow T[\tau_1/\alpha] \\
G[T[\tau]][U[\tau_1]/\alpha] &: T[\tau_1/\alpha] \rightarrow (T[\tau][U[\tau_1]/\alpha])
\end{aligned}$$

and that furthermore,  $S \circ G = \text{id}$  and  $G \circ S = \text{id}$  (i.e.,  $S$  and  $G$  form an isomorphism.) Given this, we can define the translation on terms in a straightforward fashion. Technically, we need to do this in a type-directed fashion (i.e., on the judgements) but I'll assume we've decorated terms with types as needed:

$$\begin{aligned}
E[i] &= i \\
E[f] &= f \\
E[x] &= x \\
E[\lambda x:\tau.e] &= \lambda x:T[\tau].E[e] \\
E[e_1 e_2] &= E[e_1] E[e_2] \\
E[(e : \forall \alpha.\tau)\tau_1] &= (S[T[\tau]][U[\tau_1]/\alpha](E[e]U[\tau_1])) \\
E[\Lambda \alpha.e] &= \Lambda \alpha.E[e]
\end{aligned}$$

And you can verify that if  $\Delta; \Gamma \vdash e : \tau$ , then  $\Delta; T[\Gamma] \vdash E[e] : T[\tau]$ .

The beauty of this translation is that if you don't use polymorphic values, then you don't pay any boxing/unboxing overhead. Of course, where we do use a polymorphic function, we'll be paying some overhead and in all likelihood, we'll want to inline and beta-reduce the hell out of those coercions.

Unfortunately, there's a problem with this translation when it comes to refs, inductive, or recursive types. For instance, consider what needs to happen to a call to map, when we specializing it to operate on a list of floating-point values. We end up constructing a coercion  $S$  which maps  $\text{list}(\text{float})$  to  $\text{list}(\text{Float})$ . Hopefully, a really good compiler could do the deforestation and push the  $S/G$  coercions into the map itself. But that's hard to do with separate compilation, and without very aggressive optimization. To avoid this overhead, Leroy's Gallium compiler forced values in inductive data structures to be boxed.

For references, the problem is that the  $S/G$  coercions want to make *copies* of the values. But you can't make a copy of the reference cell because you'll end up destroying the sharing. (Technically, you could represent the ref by a pair of functions to get/set the unboxed contents and then coerce those functions, but doing so kills all of the wins of the unboxed representations.)

A third subtle problem that arose with coercions when implemented in SML/NJ is that they can pile up. In particular, it's possible with certain kinds of loops to end up with a  $G \circ S$  wrapped around a value each time you go around the loop.

Nonetheless, I find this one of the most interesting and somehow satisfying translations that really demonstrates why type-directed translation can be such a powerful tool for understanding compilers.

## 7.5 Intensional Polymorphism

A fourth approach to dealing with polymorphism is called "intensional type analysis" or "runtime-type dispatch". The basic idea was laid out in a POPL'95 paper by Harper and Morrisett and was implemented in the TIL, TIL(T), and SML/NJ compilers. (Actually, SML/NJ uses a combination of coercions and intensional type analysis.) The observation is that we could translate a polymorphic function, such as the identity so that it actually took a representation of the type as a value at run-time. Then it could look at this type-representation to decide what to do. For instance, the identity function might get compiled to something like this:

$$\Lambda\alpha.\lambda t : R(\alpha).\text{if } (t = R(\text{float})) \text{ then } \lambda x:\text{float}.x \text{ else } \lambda x:\alpha.x$$

Here,  $R(\alpha)$  is a type corresponding to the *representation* of the unknown type  $\alpha$ . It turns out that you can compile ML-like languages (really, full, predicative System-F) in this fashion and get all of the benefits of the coercion-based approach without some of the drawbacks. In particular, this technique is compatible with refs, inductive data structures, recursive types, etc.

Of course, it has its own costs in terms of constructing, passing, and testing representations of types at run-time. But those representations can be used for other things (e.g., GC information, ad hoc polymorphic operations such as ML's polymorphic equality, etc.) I'm not sure this is really a win compared to say, the JIT approach. It does have the advantage that like the coercion-based approach, you only pay for polymorphism if you use it. Still, I think that it's probably not worth the trouble.

## 7.6 Finite Products

Another approach, pioneered by the Church project across the river, is to really think of a polymorphic value as a product. For a language without polymorphic recursion, the set of types at which we can apply a polymorphic value can be (conservatively) computed at compile time. Then we could build the specialized versions and put them all in a big tuple. Then polymorphic instantiation becomes a projection off of that tuple. (Dually, for existentials, we can use a datatype, and unpacking corresponds to doing a pattern-match.) I suspect that this technique could be used to make MLton support 1st class polymorphism (though not polymorphic recursion.)

In summary, there are a number of approaches for implementing polymorphism all with different tradeoffs and limitations. The issues start to multiply when you consider languages such as C# and the latest versions of Java where you combine parametric polymorphism with subtyping (and run-time downcasts, reflection, etc.)