Lecture Notes on Parametric Polymorphism

15-814: Types and Programming Languages
Frank Pfenning

Lecture 15
Thursday, October 24, 2019

1 Introduction

Polymorphism refers to the possibility of an expression to have multiple types. In that sense, all the languages we have discussed so far are polymorphic. For example, we have

$$\lambda x.\, x : \tau \to \tau$$

for any type τ. More specifically, then, we are interested in reflecting this property in a type itself. For example, we might want to state

$$\lambda x.\, x : \forall \alpha.\, \alpha \to \alpha$$

to express all the types above, but now in a single form. This means we could now reason within the type system about polymorphic functions rather than having to reason only at the metalevel with statements such as "*for all types τ, . . .*". Our system will be slightly different from this, for reasons that will become apparent later.

Christopher Strachey [Str00] distinguished two forms of polymorphism: *ad hoc polymorphism* and *parametric polymorphism*. Ad hoc polymorphism refers to multiple types possessed by a given expression or function which has different implementations for different types. For example, *plus* might have type $\mathit{int} \to \mathit{int} \to \mathit{int}$ but als $\mathit{float} \to \mathit{float} \to \mathit{float}$ with different implementations at these two types. Similarly, a function $\mathit{show} : \forall \alpha.\, \alpha \to \mathit{string}$ might convert an argument of any type into a string, but the conversion function itself will of course have to depend on the type of the argument: printing Booleans, integers, floating point numbers, pairs, etc. are all very different operations. Even though it is an important concept in programming languages, in this lecture we will not be concerned with ad hoc polymorphism.

In contrast, *parametric polymorphism* refers to a function that behaves the same at all possible types. The identity function, for example, is parametrically polymorphic because it just returns its argument, regardless of its type. The essence of "parametricity" wasn't rigorously captured the beautiful analysis by John Reynolds [Rey83], which we will sketch in Lecture 16 on *Parametricity*. In this lecture we will present typing rules and some examples.

2 Universally Quantified Types

We would like to add types of the form $\forall\alpha.\,\tau$ to our menagerie of types to express parametric polymorphism. All of our types so far had constructors and destructors, where computation arises when a destructor meets its corresponding constructor. So our language design principles suggest that we should have a constructor for elements of type $\forall\alpha.\,\tau$ and a corresponding destructor. The at first surprising approach is to think of an expression of type $\forall\alpha.\,\tau$ as a *function* that takes a *type* as an argument.

This is a rather radical change of attitude. So far, our expressions contained no types at all, and suddenly types become embedded in expressions and are actually passed to functions! Let's see where it leads us. Now we *could* write

$$\lambda\alpha.\,\lambda x.\,x : \forall\alpha.\,\alpha \to \alpha$$

but abstraction over a type seems so different from abstraction over a value that we make up a new notation and instead write

$$\Lambda\alpha.\,\lambda x.\,x : \forall\alpha.\,\alpha \to \alpha$$

using a capital lambda (Λ). In order to express the typing rules, we carry an additional context Δ which has the form

$$\Delta ::= \alpha_1\,\mathit{tp}, \ldots, \alpha_n\,\mathit{tp}$$

where each of the α_i must be distinct. Our judgment is $\Delta\,;\,\Gamma \vdash e : \tau$ where

1. all value variables x in e are declared in Γ, and
2. all type variables α in Γ, e, and τ are declared in Δ.

Then we have the rule

$$\dfrac{\Delta, \alpha\ \mathit{tp}\ ; \Gamma \vdash e : \tau}{\Delta\ ; \Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, \tau}\ \textsf{tplam}$$

We haven't yet seen how α can actually appear in e, but we can already verify:

$$\dfrac{\dfrac{\dfrac{}{\alpha\ \mathit{tp}\ ; x : \alpha \vdash x : \alpha}\ \textsf{var}}{\alpha\ \mathit{tp}\ ; \cdot \vdash \lambda x.\, x : \alpha \to \alpha}\ \textsf{lam}}{\cdot\ ; \cdot \vdash \Lambda\alpha.\, \lambda x.\, x : \forall\alpha.\, \alpha \to \alpha}\ \textsf{tplam}$$

The next question is how do we *apply* such a polymorphic function to a type? Again, we *could* just write $e\ \tau$ for the application of a polymorphic function e to a type τ, but we would like it to be more syntactically apparent so we write $e\,[\tau]$. We would expect, for example, that

$$(\Lambda\alpha.\, \lambda x.\, x)\,[\mathit{nat}]\,\overline{3} : \mathit{nat}$$

and (store this away for later) that this expression evaluates to $\overline{3}$. In order for this to work out as expected, we require that

$$(\Lambda\alpha.\, \lambda x.\, x)\,[\mathit{nat}] : \mathit{nat} \to \mathit{nat}$$

which leads us to the following typing rule.

$$\dfrac{\Delta\ ; \Gamma \vdash e : \forall\alpha.\, \tau \quad \Delta \vdash \sigma\ \mathit{tp}}{\Delta\ ; \Gamma \vdash e\,[\sigma] : [\sigma/\alpha]\tau}\ \textsf{tpapp}$$

The second premise is there to check that all type variables in σ are among the variables in Δ. We only pass in Δ because value variables can not appear in σ, only type variables. Now assume we have

$$\begin{array}{rcl} \mathit{id} & : & \forall\alpha.\, \alpha \to \alpha \\ \mathit{id} & = & \Lambda\alpha.\, \lambda x.\, x \end{array}$$

Then we can typecheck:

$$\dfrac{\dfrac{\overset{\vdots}{\cdot\ ; \cdot \vdash \mathit{id} : \forall\alpha.\, \alpha \to \alpha} \quad \cdot \vdash \mathit{nat}\ \mathit{tp}}{\cdot\ ; \cdot \vdash \mathit{id}\,[\mathit{nat}] : \mathit{nat} \to \mathit{nat}}\ \textsf{tpapp} \quad \dfrac{}{\cdot\ ; \cdot \vdash \overline{3} : \mathit{nat}}}{\cdot\ ; \cdot \vdash \mathit{id}\,[\mathit{nat}]\,\overline{3} : \mathit{nat}}\ \textsf{app}$$

where it only remains to be seen how to verify that *nat* is a closed type (that is, has no free type variables). Fortunately, that's easy: we just check all the components of a type. We only give the rules for type variables, function types, recursive types, and universal types since the others are analogous and straightforward.

$$\frac{\Delta \vdash \tau_1\ \mathit{tp} \quad \Delta \vdash \tau_2\ \mathit{tp}}{\Delta \vdash \tau_1 \to \tau_2\ \mathit{tp}}\ \mathsf{tp/arrow} \qquad \frac{\alpha\ \mathit{tp} \in \Delta}{\Delta \vdash \alpha\ \mathit{tp}}\ \mathsf{tp/var}$$

$$\frac{\Delta, \alpha\ \mathit{tp} \vdash \tau\ \mathit{tp}}{\Delta \vdash \rho\alpha.\, \tau\ \mathit{tp}}\ \mathsf{tp/rho} \qquad \frac{\Delta, \alpha\ \mathit{tp} \vdash \tau\ \mathit{tp}}{\Delta \vdash \forall\alpha.\, \tau\ \mathit{tp}}\ \mathsf{tp/forall}$$

In summary, what we have is the following:

$$\begin{array}{lccl} \text{Types} & \tau & ::= & \ldots \mid \forall\alpha.\, \tau \\ \text{Expressions} & e & ::= & \ldots \mid \Lambda\alpha.\, e \mid e\,[\tau] \end{array}$$

3 Dynamics of Polymorphism

Now that we have settled the statics, we should decide on the dynamics: how do polymorphic functions execute. We begin with a decision on what a value is and specify, in analogy with the usual λ-abstraction:

$$\frac{}{\Lambda\alpha.\, e\ \mathit{val}}\ \mathsf{val/tplam}$$

So a type abstraction is always a value. We then have two rules for applying functions to types.

$$\frac{e \mapsto e'}{e\,[\tau] \mapsto e'\,[\tau]}\ \mathsf{step/tpapp}_0 \qquad \frac{}{(\Lambda\alpha.\, e)[\tau] \mapsto [\tau/\alpha]e}\ \mathsf{step/tpapp}$$

In the second rule, we do need to substitute τ for α in e, because α may actually occur in e. For example, in the first reduction below we need to substitute *nat* for β.

$$(\Lambda\beta.\, \mathsf{id}\,[\beta])\,[\mathit{nat}]\,\overline{3} \mapsto \mathsf{id}\,[\mathit{nat}]\,\overline{3} = (\Lambda\alpha.\, \lambda x.\, x)\,[\mathit{nat}]\,\overline{3} \mapsto (\lambda x.\, x)\,\overline{3} \mapsto \overline{3}$$

We also substitute in the second step, but since α does not occur in $\lambda x.\, x$ it has no effect.

We have already used type substitution (for example, for recursive types as $[\rho.\,\tau/\alpha]\tau$), and it is the usual capture-avoiding substitution. The only thing new here is that we substitute into an *expression* e, where we need to avoid capture by the type abstractions present in e.

Other choices regarding the dynamics are possible. For example, we could declare that only $\Lambda\alpha.\,v$ is a value. But this does not seem to fit as well with our structure, and pattern matching against a Λ-abstraction seems like a strange construct.

4 Some Examples

We can draw on examples from the pure λ-calculus encodings to write some polymorphic functions. For example, what do we expect the functions of type

$$\forall\alpha.\,\alpha \to \alpha \to \alpha$$

to be? Traversing this type and postulating the right form of abstraction, we arrive at

$$\Lambda\alpha.\,\lambda x.\,\lambda y.\,\boxed{}$$

where

$$x : \alpha, y : \alpha \vdash \boxed{} : \alpha$$

that is, the expression to fill the hole has type α and we know that x and y both have type α. We might suspect that the only *normal forms* we can put are x or y (we proved a theorem of this kind in Lecture 4 for the simply-typed λ-calculus). So we get two functions

$$\begin{array}{lcl} f & : & \forall\alpha.\,\alpha \to \alpha \to \alpha \\[4pt] f_1 & = & \Lambda\alpha.\,\lambda x.\,\lambda y.\,x \\ f_2 & = & \Lambda\alpha.\,\lambda x.\,\lambda y.\,y \end{array}$$

and we might suspect in some way these are the *only* functions of this type. Or at least all other functions should be equal to one of these two. Our earlier proof does not apply here: at that time we *only* had functions and nothing else—now we have products, sums, recursive types, and also universal types. Also, our λ-expressions are now no longer observable, so it seems wrong to argue about normal forms rather than with evaluation.

But if there are morally only two functions inhabiting this type, maybe we could prove:

$$\forall\alpha.\,\alpha \to \alpha \to \alpha \overset{?}{\cong} 1 + 1$$

As a preliminary exercise, and also an exercise in writing polymorphic functions, let's try it. One really has to write these kind of functions in a type-directed way, building them up based on the type.

$$\begin{array}{lcl} \mathit{Forth} & : & (\forall\alpha.\, \alpha \to \alpha \to \alpha) \to (1+1) \\ \mathit{Forth} & = & \lambda f.\, \boxed{} \end{array}$$

This gap is typed with

$$f : (\forall\alpha.\, \alpha \to \alpha \to \alpha) \vdash \boxed{} : 1+1$$

Looking at the type of f, we eventually want something of type $1+1$, which we can achieve if we apply f to the type $1+1$.

$$\mathit{Forth} = \lambda f.\; \underbrace{f\,[1+1]}_{:\,(1+1)\to(1+1)\to(1+1)}\; \boxed{}$$

We obtained the type of $f\,[1+1]$ from the type of f, instantiated at type $1+1$. Recall how f_1 and f_2 worked: they are the two projection functions we used to call *true* and *false* in the pure λ-calculus. So if we apply $f\,[1+1]$ to $\ell \cdot \langle\,\rangle$ and $r \cdot \langle\,\rangle$, we should receive one of these two back, depending on whether $f = f_1$ or $f = f_2$.

$$\mathit{Forth} = \lambda f.\, f\,[1+1]\,(\ell \cdot \langle\,\rangle)\,(r \cdot \langle\,\rangle)$$

Moving on to the *Back* function

$$\begin{array}{lcl} \mathit{Back} & : & (1+1) \to (\forall\alpha.\, \alpha \to \alpha \to \alpha) \\ \mathit{Back} & = & \lambda s.\, \underbrace{\boxed{}}_{:\,(\forall\alpha.\, \alpha \to \alpha \to \alpha)} \end{array}$$

At this point we want to return f_1 or f_2, depending on whether s is $\ell \cdot \langle\,\rangle$ or $r \cdot \langle\,\rangle$.

$$\mathit{Back} = \lambda s.\, \mathsf{case}\; s\; (\ell \cdot \_ \Rightarrow f_1 \mid r \cdot \_ \Rightarrow f_2)$$

We would like to check that these two functions form an isomorphism, but we'll postpone this question until the next lecture.

In lecture we also considered a simpler potential isomorphism:

$$\forall\alpha.\, \forall\beta.\, \alpha \to \beta \to \alpha \overset{?}{\cong} \forall\alpha.\, \alpha \to \forall\beta.\, \alpha \to \beta \to \alpha$$

We constructed, following the structure of the type

$$\begin{array}{lcl} \mathit{Forth} & : & (\forall\alpha.\,\forall\beta.\,\alpha \to \beta \to \alpha) \to \forall\alpha.\,\alpha \to \forall\beta.\,\beta \to \alpha \\ \mathit{Forth} & = & \lambda f.\,\Lambda\alpha.\,\lambda x.\,\Lambda\beta.\,\lambda y.\,f\,[\alpha]\,[\beta]\,x\,y \\[1em] \mathit{Back} & : & (\forall\alpha.\,\alpha \to \forall\beta.\,\beta \to \alpha) \to \forall\alpha.\,\forall\beta.\,\alpha \to \beta \to \alpha \\ \mathit{Back} & = & \lambda g.\,\Lambda\alpha.\,\Lambda\beta.\,\lambda x.\,\lambda y.\,g\,[\alpha]\,x\,[\beta]\,y \end{array}$$

We didn't check, but it was a good bet that these two would compose to the identity in both directions.

5 Example: Natural Numbers

Recall from Lecture 4 that in the pure λ-calculus we found that normal forms of type $(\alpha \to \alpha) \to (\alpha \to \alpha)$ corresponded to the natural numbers. So we may conjecture:

$$\forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha) \overset{?}{\cong} \mathit{nat}$$

where

$$\mathit{nat} = \rho\alpha.\,(\texttt{zero} : 1) + (\texttt{succ} : \alpha) \cong (\texttt{zero} : 1) + (\texttt{succ} : \mathit{nat})$$

Again, let's write the two potential witnesses of the isomorphism.

$$\begin{array}{lcl} \mathit{Forth} & : & (\forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)) \to \mathit{nat} \\ \mathit{Forth} & = & \lambda f.\quad \underbrace{f\,[\mathit{nat}]}_{(\mathit{nat} \to \mathit{nat}) \to (\mathit{nat} \to \mathit{nat})} \quad \boxed{} \end{array}$$

How do we fill the box? Because the representation of a number is an iterator, it seems clear we should apply it to the successor function and zero.

$$\begin{array}{lcl} \mathit{zero} & : & \mathit{nat} \\ \mathit{zero} & = & \mathsf{fold}\,(\texttt{zero} \cdot \langle\,\rangle) \\[0.5em] \mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\ \mathit{succ} & = & \lambda n.\,\mathsf{fold}\,(\texttt{succ} \cdot n) \\[0.5em] \mathit{Forth} & : & (\forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)) \to \mathit{nat} \\ \mathit{Forth} & = & \lambda f.\,f\,[\mathit{nat}]\,\mathit{succ}\,\mathit{zero} \end{array}$$

The other direction is trickier. We want to build the right functional representation of the number from it representation as a member of a recursive type. We didn't do this in lecture, but to understand this code better, let's recall the *λ-calculus representation* of zero and successor.

$$\begin{array}{lcl} \mathit{lzero} & : & \forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha) \\ \mathit{lzero} & = & \Lambda\alpha.\,\lambda s.\,\lambda z.\,z \\ \mathit{lsucc} & : & (\forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)) \to (\forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha)) \\ \mathit{lsucc} & = & \lambda k.\,\Lambda\alpha.\,\lambda s.\,\lambda z.\,s\,(k\,[\alpha]\,s\,z) \end{array}$$

Now we can implement *Back* as a recursive function

$$\begin{array}{l} \mathit{Back} : \mathit{nat} \to \forall\alpha.\,(\alpha \to \alpha) \to (\alpha \to \alpha) \\ \mathit{Back} = \mathsf{fix}\ B.\,\lambda n.\,\mathbf{case}\ (\mathsf{unfold}\,n)\ (\,\mathtt{zero}\cdot\_ \Rightarrow \mathit{lzero} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathtt{succ}\cdot m \Rightarrow \mathit{lsucc}\,(B\,m)\,) \end{array}$$

You should convince yourself that this is type-correct and represents the intuitively correct function. We don't have the tools yet to prove that these two really constitute an isomorphism, though.

6 Theory

We did not discuss this in lectures, but of course we should expect preservation and progress, as well as some substitution properties and canonical form theorems. We will talk about these at the beginning of the next lecture.

Bidirectional type checking continues to work well, but it requires the programmer to supply a lot of types. For the fully general system, many problems (such as type inference, carefully defined) will be undecidable. Some languages such as ML have adopted a restricted form of parametric polymorphism where the quantifiers can occur only on the outside, and can only be instantiated with quantifier-free types. In that case, type inference can remain more or less what it is for the language without parametric polymorphism: we construct the skeleton of a typing derivation, solve all the equations that arise from when we fill in the holes. The most general solution will have some free variables that we then explicit quantify over. We may talk about this aspect of elaboration in a future lecture.

Exercises

Exercise 1 Find closed types τ and σ such that

$$\cdot\,;\,\cdot \vdash \lambda x.\,x\,[\tau]\,x : \sigma$$

Exercise 2 Extend the K Machine with additional continuations and transitions to implement polymorphism.

Exercise 3 For each of the following potential isomorphisms, fill in the missing entry and write down properly typed candidate functions *Forth* and *Back* to witness an isomorphism. You do not need to prove the isomorphism property. On the left side of each candidate isomorphism, we have a type with only universal quantification and function types. On the right side we have a type using any of the type constructors from this course (functions, eager products, lazy products, unit, sum, recursive types, lazy products) but *not* universally quantified types. We have filled in the first line for you, and you can find the *Forth* and *Back* functions in Section 4 (no need to repeat them).

$$\begin{array}{lllll}
 & \forall\alpha.\, \alpha \to \alpha \to \alpha & \cong & 1 + 1 \\
(1) & \forall\alpha.\, \alpha \to \alpha & \cong & \boxed{} \\
(2) & \forall\alpha.\, \alpha & \cong & \boxed{} \\
(3) & \boxed{} & \cong & \rho\alpha.\, (\texttt{e} : 1) + (\texttt{b0} : \alpha) + (\texttt{b1} : \alpha) \\
(4) & \boxed{} & \cong & \mathit{nat} \times \mathit{nat}
\end{array}$$

The type $\mathit{nat} = \rho\alpha.\, (\texttt{zero} : 1) + (\texttt{succ} : \alpha)$. You may use the functions from Section 5 in your solution to Part 4.

Exercise 4 In the presented formulation of polymorphism, we have to carry type information at runtime. But we also said that polymorphism should be *parametric*, that is, the behavior of the function should not depend on its type. We want to take advantage of that by "erasing" the types before a (closed, well-typed) program is executed. For this exercise, we remain in the fragment with only functions, unit, and parametric polymorphism.

1. Consider the following replacements in the types and expressions, written as $\#(\tau)$ and $\#(e)$:

   (a) Replace $\forall\alpha.\, \tau$ by τ everywhere

   (b) Replace $\Lambda\alpha.\, e$ by e everywhere

   (c) Replace $e\,[\tau]$ by e everywhere

If $\cdot\,;\cdot \vdash e : \tau$ then $\cdot\,;\cdot \vdash \#(e) : \#(\tau)$, but e is not bisimilar to $\#(e)$. Give a counterexample to bisimilarity.

2. Find alternative translations $\$(\tau)$ and $\$(e)$ from the language with parametric polymorphism to the language without such that e is bisimilar to $\$(e)$ under the usual small-step dynamics $e \mapsto e'$ for both source and target. You do not need to prove this result.

References

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, September 1983.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 2000. Notes for lecture course given at the International Summer School in Computer Programming at Copenhagen, Denmark, August 1967.