Lecture Notes on
Parametric Polymorphism

15-814: Types and Programming Languages
Frank Pfenning, lecture by David Kahn

Lecture 6
Thursday, September 16, 2021

1 Introduction

Polymorphism refers to the possibility of an expression to have multiple types. In that sense, the simply-typed λ-calculus is *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 *int* → *int* → *int* but also *float* → *float* → *float* with different implementations at these two types. Similarly, a function *show* : $\forall \alpha.\, \alpha \to string$ 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 until the beautiful analysis by John Reynolds [Rey83], which we will sketch in a later lecture on parametricity. In this lecture we will present typing rules and some examples.

Slightly different systems for parametric polymorphism were discovered independently by Jean-Yves Girard [Gir71] and John Reynolds [Rey74]. Girard worked in the context of *logic* and developed *System F*, while Reynolds worked directly on *type systems* for programming language and designed the polymorphic λ-calculus. With minor syntactic changes, we will follow Reynolds's presentation.

2 Universally Quantified Types

We would like to add types of the form $\forall \alpha. \tau$ to express parametric polymorphism. The fundamental idea is that an expression of type $\forall \alpha. \tau$ is 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 expressions 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, our contexts carry two different forms of declarations: $x : \tau$ (as we had so far) and now also $\alpha \, type$, expressing that α is a type variable. The typing judgment then is still $\Gamma \vdash e : \tau$, without repeated variables or type variables in Γ. There will be some further presuppositions mentioned later. For type abstractions, we have the rule

$$\frac{\Gamma, \alpha \, type \vdash e : \tau}{\Gamma \vdash \Lambda \alpha. e : \forall \alpha. \tau} \; \text{tp/tplam}$$

Here, α is a bound variable in $\Lambda\alpha.\,e$ and $\forall\alpha.\,\tau$ so we allow it to be silently renamed if it conflicts with any variable already declared in Γ.

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

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

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 $e\,[\tau]$.

Let's return to Church's representation of natural numbers. With the quantifier, we now have

$$nat = \forall\alpha.\,(\alpha\to\alpha)\to\alpha\to\alpha$$

Then we can verify with typing derivations as above:

$$
\begin{array}{rcl}
zero & : & nat \\
zero & = & \Lambda\alpha.\,\lambda s.\,\lambda z.\,z
\end{array}
$$

We also expect the successor function to have type $nat \to nat$, but there is one slightly tricky spot. We start:

$$
\begin{array}{rcl}
succ & : & nat \to nat \\
succ & = & \lambda n.\,\Lambda\alpha.\,\lambda s.\,\lambda z.\,s\,(n\,\boxed{})
\end{array}
$$

Before, we just applied n to s and z, but now $n:nat$, which means that it expects a *type* as its first argument! At this point (in a hypothetical typing derivation we did not write out) we have the context

$$n:nat, \alpha\;type, s:\alpha\to\alpha, z:\alpha$$

so we need to instantiate the quantifier with α, which next requires arguments of type $\alpha\to\alpha$ and α (which we have at hand with s and z).

$$
\begin{array}{rcl}
succ & : & nat \to nat \\
succ & = & \lambda n.\,\Lambda\alpha.\,\lambda s.\,\lambda z.\,s\,(n\,[\alpha]\,s\,z)
\end{array}
$$

It becomes more interesting with the addition function. Recall that in the untyped setting we had

$$plus = \lambda n.\, \lambda k.\, n \; succ \; k$$

iterating the successor function n times on argument k. The start of the typed version is again relatively straightforward: the only difference is that we need to apply n first to a type.

$$
\begin{aligned}
plus &: \quad nat \to nat \to nat \\
plus &= \quad \lambda n.\, \lambda k.\, n\; [\;\boxed{}\;]\; succ \; k
\end{aligned}
$$

But what type do we need? We have that the next argument has type $nat \to nat$ and the following one nat, so that we need to instantiate α with nat!

$$
\begin{aligned}
plus &: \quad nat \to nat \to nat \\
plus &= \quad \lambda n.\, \lambda k.\, n\; [nat]\; succ \; k
\end{aligned}
$$

So we need that

$$n : \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$$

and then

$$n\; [nat] : (nat \to nat) \to nat \to nat$$

We should point out that this definition of addition cannot be typed in the simply-typed λ-calculus. In that setting, n can only be applied to functions s of type $\alpha \to \alpha$ to iterate starting from $z : \alpha$. This means that very few functions are actually definable—essentially only functions like successor and addition, but not exponentiation, or predecessor (see Exercise 2).

A significant aspect of this is that we instantiate the quantifier in $nat = \forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$ with nat itself.

These considerations lead us to a rule where we *substitute into the type*:

$$\frac{\Gamma \vdash e : \forall \alpha.\, \tau \quad \Gamma \vdash \sigma \; type}{\Gamma \vdash e\, [\sigma] : [\sigma/\alpha]\tau} \; \text{tp/tpapp}$$

The second premise is there to check that the type σ which is part of the expression $e\,[\sigma]$ is *valid*. At this point in the course, this just means that all the type variables occurring in σ are declared in Γ (just like all the expression variables in e must be declared in Γ).

Here is a small sample derivation, assuming we have defined

$$
\begin{aligned}
id &: \quad \forall \alpha.\, \alpha \to \alpha \\
id &= \quad \Lambda \alpha.\, \lambda x.\, x
\end{aligned}
$$

Then we can typecheck:

$$
\cfrac{
\cfrac{\vdots \qquad\qquad \vdots}{
\cfrac{\cdot \vdash id : \forall \alpha.\,\alpha \to \alpha \quad \cdot \vdash nat\ type}{\cdot \vdash id\,[nat] : nat \to nat}\ \text{tpapp}
\qquad
\cfrac{\vdots}{\cdot \vdash \overline{3} : nat}
}
}{\cdot \vdash id\,[nat]\,\overline{3} : nat}\ \text{app}
$$

where we need some rules 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.

$$
\cfrac{\Gamma \vdash \tau_1\ type \quad \Gamma \vdash \tau_2\ type}{\Gamma \vdash \tau_1 \to \tau_2\ type}\ \text{tp/arrow}
\qquad
\cfrac{\alpha\ type \in \Gamma}{\Gamma \vdash \alpha\ type}\ \text{tp/tpvar}
$$

$$
\cfrac{\Gamma, \alpha\ type \vdash \tau\ type}{\Gamma \vdash \forall \alpha.\,\tau\ type}\ \text{tp/forall}
$$

3 Summary: Typing Rules

For the "official" typing rules it is convenient to assume that λ-abstractions are annotated with the type of the bound variable. In practice, we use a more flexible set of rules where λ-abstractions do not necessarily have to be annotated.

Here is the summary of the language of the polymorphic λ-calculus:

Types	τ ::=	$\alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha.\,\tau$
Expressions	e ::=	$x \mid \lambda x{:}\tau.\,e \mid e_1\,e_2 \mid \Lambda \alpha.\,e \mid e\,[\tau]$
Contexts	Γ ::=	$\cdot \mid \Gamma, x : \tau \mid \Gamma, \alpha\ type$

We assume that all variables and type variables in a context are distinct, and rename bound variable or type variables to maintain this invariant.

In order to avoid any "loopholes" in typing derivations we would like to *presuppose* that the context is well-formed, which comes down to ensuring that all the types occurring in them are valid. We did not discuss this somewhat technical point in lecture, but for completness' sake we provide the rules. The judgment Γ *ctx* means that Γ is a valid (or well-formed) context.

$$\frac{}{(\cdot)\ ctx}\ \text{ctx/emp} \qquad \frac{\Gamma\ ctx}{(\Gamma, \alpha\ type)\ ctx}\ \text{ctx/tpvar} \qquad \frac{\Gamma\ ctx \quad \Gamma \vdash \tau\ type}{(\Gamma, x : \tau)\ ctx}\ \text{ctx/var}$$

We now assume that whenever we write $\Gamma \vdash e : \tau$ or $\Gamma \vdash \tau\ type$ then $\Gamma\ ctx$. In type theory we call this a *presupposition* and we are always careful to maintain this presupposition.

$$\frac{\Gamma \vdash \tau_1\ type \quad \Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1{:}\tau_1.\, e_2 : \tau_1 \to \tau_2}\ \text{tp/lam} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{tp/var}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \text{tp/app}$$

$$\frac{\Gamma, \alpha\ type \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\, \tau}\ \text{tp/tplam} \qquad \frac{\Gamma \vdash e : \forall\alpha.\, \tau \quad \Gamma \vdash \sigma\ type}{\Gamma \vdash e\,[\sigma] : [\sigma/\alpha]\tau}\ \text{tp/tpapp}$$

We then have the property (called *regularity* in the textbook) that if $\Gamma \vdash e : \tau$ under the presupposition that $\Gamma\ ctx$ then $\Gamma \vdash \tau\ type$. This is easy to verify by rule induction (see Exercise 3).

4 Typing Self-Application Polymorphically

As an exercise in building a typing derivation, we provide a polymorphic type for self-application $\lambda x.\, x\, x$. We accomplish this by allowing x to have a polymorphic types $\forall\alpha.\, \alpha \to \alpha$. We call this type u because there is exactly one normal term of this type: the polymorphic identity function. Applying the identity to itself seems plausible in any case. So we claim:

$$\begin{aligned} u &= \forall\alpha.\, \alpha \to \alpha \\ \omega &: u \to u \\ \omega &= \lambda x.\, x\,[u]\, x \end{aligned}$$

This is established by the following typing derivation. When you want to build such a derivation yourself, you should always built it "bottom-up", starting with the final conclusion. The fact that the rules are syntax-directed

means you have no choice which rule to choose, but some parts of the type may be unknown and may need to be filled in later.

$$\dfrac{\dfrac{x : u \vdash x\,[u] : \boxed{} \qquad x : u \vdash x : \boxed{}}{x : u \vdash x\,[u]\;x : u}\ \text{tp/app}}{\cdot \vdash \lambda x.\,x\,[u]\;x : u \to u}\ \text{tp/lam}$$

As a rule of thumb, it seems to work best to first fill in the first premise of an application (rule tp/app) and then the second. Continuing in the left branch of the derivation (and remembering that $u = \forall \alpha.\,\alpha \to \alpha$:

$$\dfrac{\dfrac{\dfrac{\overline{x : u \vdash x : u}\ \text{tp/var} \qquad \dfrac{\vdots \quad \dfrac{x : u, \alpha\ type \vdash \alpha \to \alpha\ type}{x : u \vdash u\ type}\ \text{tp/forall}}{}}{x : u \vdash x\,[u] : {\color{red}u \to u}}\ \text{tp/tpapp} \qquad x : u \vdash x : \boxed{}}{x : u \vdash x\,[u]\;x : u}\ \text{tp/app}}{\cdot \vdash \lambda x.\,x\,[u]\;x : u \to u}\ \text{tp/lam}$$

The type emphasized in red arises as $[u/\alpha](\alpha \to \alpha) = u \to u$. The second premise of the application is immediate by the typing rule for variables and we obtain

$$\dfrac{\dfrac{\dfrac{\overline{x : u \vdash x : u}\ \text{tp/var} \qquad \dfrac{\vdots \quad \dfrac{x : u, \alpha\ type \vdash \alpha \to \alpha\ type}{x : u \vdash u\ type}\ \text{tp/forall}}{}}{x : u \vdash x\,[u] : u \to u}\ \text{tp/tpapp} \qquad \dfrac{}{x : u \vdash x : u}\ \text{tp/var}}{x : u \vdash x\,[u]\;x : u}\ \text{tp/app}}{\cdot \vdash \lambda x.\,x\,[u]\;x : u \to u}\ \text{tp/lam}$$

The fact that $\alpha \to \alpha$ is a valid type follows quickly by the tp/arrow and tp/tpvar rules. There are more types that work for self-application (see Exercise 4).

Crucial in this example is that we can instantiate the quantifier in $u = \forall \alpha.\,\alpha \to \alpha$ with u itself. This "self-referential" nature of the type quantifier is called *impredicativity* because it quantifies not only over types already defined, but also itself. Some systems of type theory reject impredicative quantification because the meaning of the quantified type is not constructed from the meaning of types we previously understand. Impredicativity was

also seen as a source of paradoxes, although Girard did give a syntactic argument for the consistency of System F [Gir71] with impredicative quantification.

5 Church Numerals Revisited

We can now revisit the representation of Church numerals and express them and functions on them in the polymorphic λ-calculus. We present the definitions in the language LAMBDA, which uses polymorphic types when files have extension .poly or the command line argument -l poly. We use !a as concrete syntax for $\forall \alpha$, and /\a for $\Lambda \alpha$. Type definitions are preceded by the keyword type, and type declarations for variable definitions are preceded by the keyword decl.

```
1  type nat = !a. (a -> a) -> a -> a
2
3  decl zero : nat
4  decl succ : nat -> nat
5
6  defn zero = /\a. \s. \z. z
7  defn succ = \n. /\a. \s. \z. s (n [a] s z)
8
9  decl plus : nat -> nat -> nat
10 defn plus = \n. \k. n [nat] succ k
11
12 decl times : nat -> nat -> nat
13 defn times = \n. \k. n [nat] (plus k) zero
14
15 norm _0 = zero
16 norm _1 = succ _0
17 norm _2 = succ _1
18 norm _3 = succ _2
19
20 norm _6 = times _2 _3
```

Listing 1: Polymorphic natural numbers in LAMBDA

So far, this straightforwardly follows the structure of the motivating examples. In order to represent the predecessor function, we require pairs of natural numbers. But what are their types? Recall:

$$pair = \lambda x.\, \lambda y.\, \lambda k.\, k\, x\, y$$

from which conjecture something like

$$pair : nat \rightarrow nat \rightarrow (nat \rightarrow nat \rightarrow \tau) \rightarrow \tau$$

where τ is arbitrary. So we realize that this function is *polymorphic* and we abstract over the result type of the continuation. We call the type of pairs of natural numbers *nat2*. In the type of the *pair* function it is then convenient to place the type abstraction *after* the two natural numbers have been received.

$$nat2 \quad = \quad \forall \alpha. (nat \rightarrow nat \rightarrow \alpha) \rightarrow \alpha$$

$$
\begin{aligned}
pair \quad &: \quad nat \rightarrow nat \rightarrow nat2 \\
pair \quad &= \quad \lambda x. \lambda y. \Lambda \alpha. \lambda k. k \, x \, y
\end{aligned}
$$

Now we can define the *pred2*, with the specification that $pred2 \; \overline{n} = pair \; \overline{n} \; \overline{n \div 1}$. We leave open the two places we have to provide a type.

$$
\begin{aligned}
pred2 \quad &: \quad nat \rightarrow nat \rightarrow nat \\
pred2 \quad &= \quad \lambda n. n \, [\;\boxed{}\;] \, (\lambda p. p \, [\;\boxed{}\;] \, (\lambda x. \lambda y. pair \; (succ \; x) \; x)) \, (pair \; zero \; zero)
\end{aligned}
$$

In the first box, we need to fill in the result type of the iteration (which is the type argument to $\overline{n}$), and this is *nat2*. In the second box we need to fill in the result type for the decomposition into a pair, and that is also *nat2*. Then, for the final definition of *pred* we only extract the second component of the pair, so the continuation only returns a natural number rather than a pair.

$$
\begin{aligned}
pred \quad &: \quad nat \rightarrow nat \\
pred \quad &= \quad \lambda n. n \, [nat] \, (\lambda x. \lambda y. y)
\end{aligned}
$$

Below is a summary of this code in LAMBDA.

```
1  type nat2 = !a. (nat -> nat -> a) -> a
2
3  decl pair : nat -> nat -> nat2
4  defn pair = \x. \y. /\a. \k. k x y
5
6  decl pred2 : nat -> nat2
7  defn pred2 = \n. n [nat2] (\p. p [nat2] (\x. \y. pair (succ x) x))
8                           (pair zero zero)
9
10 decl pred : nat -> nat
11 defn pred = \n. pred2 n [nat] (\x. \y. y)
12
13 norm _6_5 = pred2 _6
14 norm _5 = pred _6
```

Listing 2: Predecessor on natural numbers in LAMBDA

6   Theory

We did not discuss this in lectures, but of course we should expect the properties for the simply-typed λ-calculus to carry over, once suitable reduction rules have been defined. We will talk about these at the beginning of the next lecture.

   One remarkable fact about the polymorphic λ-calculus (which is quite difficult to prove) is that every expression still has a normal form.

Exercises

Exercise 1 Fill in the blanks in the following judgments so that it holds, or indicate there is no way to do so. You do not need to justify your answer or supply a typing derivation, and the types do not need to be "most general" in any sense. As always, feel free to use LAMBDA to check your answers.

(i) $\boxed{} \vdash \forall \alpha.\, \alpha \to \beta\ type$

(ii) $\boxed{} \vdash \Lambda\alpha.\, x\,[\alpha \to \alpha]\, y\,[\beta] : \forall \alpha.\, \beta$

(iii) $\cdot \vdash \lambda x.\, x\,[\boxed{}]\, x\, x : \boxed{}$

(iv) $\alpha\ type \vdash \boxed{} : \forall \beta.\, \alpha \to \beta$

(v) $x : \forall \alpha.\, (\forall \beta.\, \beta \to \beta) \to \alpha, \gamma\ type \vdash \boxed{} : (\gamma \to \gamma) \to \gamma$

Exercise 2

(i) Find a definition of *plus* : *nat* $\to$ *nat* $\to$ *nat* that works in the simply-typed λ-calculus in the sense that we need to instantiate the type $\forall \alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$ only with a type variable.

(ii) Give a simply-typed definition (in the sense of part (i)) for *times* or conjecture that none exists.

Exercise 3 Prove that if $\Gamma \vdash e : \tau$ under the presupposition that $\Gamma\ ctx$ then $\Gamma \vdash \tau\ type$.

Exercise 4 We write F for a (mathematical) function from types to types (which is not expressible in the polymorphic λ-calculus but requires system

F^ω). A more general family of types (one for each F) for self-application is given by

$$u_F \quad = \quad \forall \alpha.\, \alpha \to F(\alpha)$$

$$\omega_F \quad : \quad u_F \to F(u_F)$$
$$\omega_F \quad = \quad \lambda x.\, x\,[u_F]\,x$$

We recover the type from this lecture with $F = \Lambda\alpha.\,\alpha$. You may want to verify the general typing derivation in preparation for the following questions, but you do not need to show it.

(i) Consider $F = \Lambda\alpha.\,\alpha \to \alpha$. In this case $u_F = bool$. Calculate the type and characterize the behavior of ω_F as a function on Booleans.

(ii) Consider $F = \Lambda\alpha.\,(\alpha \to \alpha) \to \alpha$. Calculate u_F, the type of ω_F, and characterize the the behavior of ω_F. Can you relate u_F and ω_F to the types and functions we have considered in the course so far?

References

[Gir71] Jean-Yves Girard. Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, Amsterdam, 1971.

[Rey74] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[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.