Lecture Notes on Parametric Polymorphism

15-814: Types and Programming Languages
Frank Pfenning

Lecture 11
October 9, 2018

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, the judgment

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

expresses all the types above, but now in a single form. This means we can 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* $\tau, \ldots$".

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 12 on *Parametricity*. In this lecture we will present typing rules and some examples.

2 Extrinsic Polymorphic Typing

We now return to the pure simply-typed λ-calculus.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \\ e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \end{array}$$

We would like the judgment $e : \forall\alpha.\, \tau$ to express that e has *all* types $[\sigma/\alpha]\tau$ for arbitrary σ. This will close an important gap in our earlier development, where the fixed type variables seemed to be inflexible. The construct $\forall\alpha.\, \tau$ binds the type variable α with scope τ. As usual, we identify types that differ only in the names of their bound type variables.

Now we would like to allow the following:

$$\begin{array}{rcl} \mathit{bool} & = & \forall\alpha.\, \alpha \to \alpha \\ \mathit{true} & : & \mathit{bool} \\ \mathit{true} & = & \lambda x.\, \lambda y.\, x \\ \mathit{false} & : & \mathit{bool} \\ \mathit{false} & = & \lambda x.\, \lambda y.\, y \\ \\ \mathit{nat} & = & \forall\alpha.\, \alpha \to (\alpha \to \alpha) \to \alpha \\ \mathit{zero} & : & \mathit{nat} \\ \mathit{zero} & = & \lambda z.\, \lambda s.\, z \\ \mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\ \mathit{succ} & : & \lambda n.\, \lambda z.\, \lambda s.\, s\,(n\, z\, s) \end{array}$$

This form of typing is called *extrinsic* because polymorphic types describe a properties of expression, but the expressions themselves remain unchanged.

In an *intrinsic* formulation the expressions themselves carry types and express polymorphism. There are good arguments for both forms of presentation. For the sake of simplicity we use the extrinsic form. This means we depart from our approach so far where each new type constructor was accompanied by corresponding expression constructors and destructors for the new type.

In slightly different forms these calculi were designed independently by Jean-Yves Girard [Gir71] and John Reynolds [Rey74]. Girard started from higher-order logic while Reynolds from a programming where types could be passed as arguments to functions.

Given that $\lambda x.\, x : \alpha \to \alpha$ we might propose the following simple rule:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \forall \alpha.\, \tau}\ \forall I\ ?$$

We can then derive, for example,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \beta \vdash x : \alpha}\ \mathsf{var}}{x : \alpha \vdash \lambda y.\, x : \beta \to \alpha}\ {\to}I}{x : \alpha \vdash \lambda y.\, x : \forall \beta.\, \beta \to \alpha}\ \forall I\ ?}{\cdot \vdash \lambda x.\, \lambda y.\, x : \alpha \to \forall \beta.\, \beta \to \alpha}\ {\to}I}{\cdot \vdash \lambda x.\, \lambda y.\, x : \forall \alpha.\, \alpha \to \forall \beta.\, \beta \to \alpha}\ \forall I\ ?$$

This seems certainly correct. $\lambda x.\, \lambda y.\, x$ should **not** have type $\forall \alpha.\, \alpha \to \forall \beta.\, \beta \to \beta$. But:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \alpha \vdash x : \alpha}\ \mathsf{var}}{x : \alpha \vdash \lambda y.\, x : \alpha \to \alpha}\ {\to}I}{x : \alpha \vdash \lambda y.\, x : \forall \alpha.\, \alpha \to \alpha}\ \forall I\ ?}{\cdot \vdash \lambda x.\, \lambda y.\, x : \alpha \to \forall \alpha.\, \alpha \to \alpha}\ {\to}I}{\cdot \vdash \lambda x.\, \lambda y.\, x : \forall \alpha.\, \alpha \to \forall \alpha.\, \alpha \to \alpha}\ \forall I\ ?$$

is clearly incorrect, because by variable renaming we would obtain

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

and the function does not have this type. For example, instantiating α with *bool* and β with *nat* we would conclude the result is of type *nat* when it actually returns a Boolean.

The problem here lies in the instance of $\forall I$ in the third line. We say that $\lambda y.\, x$ has type $\forall \alpha.\, \alpha \to \alpha$ when it manifestly does not have this type. The problem is that α appears as the type of $x : \alpha$ in the context, so we should be not allowed to quantify over α at this point in the deduction. One way to prohibit this is the have a side condition on the rule

$$\frac{\Gamma \vdash e : \tau \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash e : \forall \alpha.\, \tau}\ \forall I\ ?$$

This would work, but in the similar situation when we wanted to avoid confusion between expression variables, we postulated that the variable was not already declared. We adopt a similar restriction here by adding a new form of context declaring type variables.

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

Here, all the α_i must be distinct. The typing judgment is then generalized to

$$\Delta\ ;\ \Gamma \vdash e : \tau$$

where all the free type variables in Γ and τ are declared in Δ, and (as before) all free expression variables in e are declared in Γ. We express that a type is well-formed in the judgment

$$\Delta \vdash \tau\ \mathit{type}$$

For now, this is just defined compositionally—we show only two rules by way of example. We refer to these as *type formation rules*.

$$\frac{\Delta \vdash \tau_1\ \mathit{type} \quad \Delta \vdash \tau_2\ \mathit{type}}{\Delta \vdash \tau_1 \otimes \tau_2\ \mathit{type}}\ \otimes F \qquad \frac{\Delta, \alpha\ \mathit{type} \vdash \tau\ \mathit{type}}{\Delta \vdash \forall \alpha.\, \tau\ \mathit{type}}\ \forall F$$

Now we can formulate the correct rule for introducing the universal quantifier in the type.

$$\frac{\Delta, \alpha\ \mathit{type}\ ;\ \Gamma \vdash e : \tau}{\Delta\ ;\ \Gamma \vdash e : \forall \alpha.\, \tau}\ \forall I^{\alpha}$$

In order to keep the context $\Delta, \alpha\ \mathit{type}$ well-formed, we imply that α is not already declared in Δ and therefore does not occur in Γ. In future, when we might allow types in expressions, α would not be allowed to occur there as well: it must be globally fresh. Sometimes we add the superscript on the rule to remind ourselves of the freshness condition.

When we instantiate the quantifer to get a more specific types we need to make sure the type we substitute is well-formed.

$$\frac{\Delta\,;\,\Gamma \vdash e : \forall\alpha.\,\tau \quad \Delta \vdash \sigma\ \mathit{type}}{\Delta\,;\,\Gamma \vdash e : [\sigma/\alpha]\tau}\ \forall E$$

Now we can easily derive that the Booleans *true* and *false* have the expected type $\forall\alpha.\,\alpha \to \alpha \to \alpha$. How about the conditional? Based on the usual conditional, we might expect

$$\mathit{if} : \mathit{bool} \to \tau \to \tau \to \tau$$

for any type τ, where the first occurrence is the 'then' branch, the second the 'else' branch and the final one the result of the conditional. But we can capture this without having to resort to metalevel quantification:

$$\mathit{if} : \mathit{bool} \to \forall\beta.\,\beta \to \beta \to \beta$$

But this is exactly the same as

$$\mathit{if} : \mathit{bool} \to \mathit{bool}$$

which makes sense since we saw in the lecture on the untyped λ-calculus that

$$\mathit{if} = \lambda b.\,b$$

3 Encoding Pairs

Now that we have the rules in place, we can consider if we can type some of the other constructions of generic data types in the pure λ-calculus. Recall:

$$\begin{array}{lcl} \mathit{pair} & = & \lambda x.\,\lambda y.\,\lambda f.\,f\,x\,y \\ \mathit{fst} & = & \lambda p.\,p\,(\lambda x.\,\lambda y.\,x) \\ \mathit{snd} & = & \lambda p.\,p\,(\lambda x.\,\lambda y.\,y) \end{array}$$

With these definitions we can easily verify

$$\begin{array}{lcl} \mathit{fst}\ (\mathit{pair}\,x\,y) & = & \mathit{fst}\ (\lambda f.\,f\,x\,y) \\ & \mapsto & (\lambda f.\,f\,x\,y)\,(\lambda x.\,\lambda y.\,x) \\ & \mapsto & x \\ \mathit{snd}\ (\mathit{pair}\,x\,y) & \mapsto^* & y \end{array}$$

Can we type these constructors and destructors in the polymorphic λ-calculus? Let's consider defining a type $\mathit{prod}\,\tau\,\sigma$ to form the product of τ and σ.

$$\begin{array}{lcl} \mathit{prod}\,\tau\,\sigma & = & ?? \\ \mathit{pair} & : & \forall\alpha.\,\forall\beta.\,\alpha \to \beta \to \mathit{prod}\,\alpha\,\beta \\ \mathit{pair} & = & \lambda x.\,\lambda y.\;\underbrace{\lambda f.\,f\,x\,y}_{:\,\mathit{prod}\,\alpha\,\beta} \end{array}$$

Since $x : \alpha$ and $y : \beta$ we see that $f : \alpha \to \beta \to ?$. But what should be the type ?? When we apply this function to the first projection (in the function *fst*), then it should be α, when we apply it to the second projection it should be β. Therefore we conjecture it should be an arbitrary type γ. So $f : \alpha \to \beta \to \gamma$ and $\mathit{prod}\,\alpha\,\beta = \forall\gamma.\,(\alpha \to \beta \to \gamma) \to \gamma$.

$$\begin{array}{lcl} \mathit{prod}\,\tau\,\sigma & = & \forall\gamma.\,(\tau \to \sigma \to \gamma) \to \gamma \\ \mathit{pair} & : & \forall\alpha.\,\forall\beta.\,\alpha \to \beta \to \mathit{prod}\,\alpha\,\beta \\ \mathit{pair} & = & \lambda x.\,\lambda y.\,\lambda f.\,f\,x\,y \\ \mathit{fst} & : & \forall\alpha.\,\forall\beta.\,\mathit{prod}\,\alpha\,\beta \to \alpha \\ \mathit{fst} & = & \lambda p.\,p\,(\lambda x.\,\lambda y.\,x) \\ \mathit{snd} & : & \forall\alpha.\,\forall\beta.\,\mathit{prod}\,\alpha\,\beta \to \beta \\ \mathit{snd} & = & \lambda p.\,p\,(\lambda x.\,\lambda y.\,x) \end{array}$$

As an example, in the definition of *fst*, the argument p will be of type $\forall\gamma.\,(\alpha \to \beta \to \gamma) \to \gamma$. We instantiate this quantifier with α to get $p : (\alpha \to \beta \to \alpha) \to \alpha$. Now we apply p to the first projection function to obtain the result α.

The observation that it may be difficult to see whether a given expression has a given type is not accidental. In fact, the question whether an expression is typable is undecidable [Wel94], even if significant information is added to the expressions [Boe85, Pfe93].

4 Encoding Sums

Now that we have represented products in the polymorphic λ-calculus, let's try sums. But it is useful to analyze a bit more how we ended up encoding products. The destructor for eager products is

$$\frac{\Gamma \vdash e : \tau \otimes \sigma \quad \Gamma, x : \tau, y : \sigma \vdash e' : \tau'}{\Gamma \vdash \mathbf{case}\; e\; \{\langle x, y\rangle \Rightarrow e'\} : \tau'}\otimes E$$

If we try to reformulate the second premise as a function, it would be $(\lambda x.\, \lambda y.\, e') : \tau \to \sigma \to \tau'$. If we think of this version of **case** as a function, it would have type $\tau \otimes \sigma \to (\tau \to \sigma \to \tau') \to \tau'$. We can now abstract over τ' to obtain $\tau \otimes \sigma \to \forall\gamma.\, (\tau \to \sigma \to \gamma) \to \gamma$. The conjecture about the representation of pairs then arises from replacing the function type an isomorphism

$$\tau \otimes \sigma \cong \forall\gamma.\, (\tau \to \sigma \to \gamma) \to \gamma$$

Our calculations in the previous section lend support to this, although we didn't actually prove such an isomorphism, just that the functions *pair*, *fst*, and *snd* satisfy the given typing and also compute correctly.

Perhaps the elimination rule for sums is subject to a similar interpretation?

$$\frac{\Gamma \vdash e : \tau + \sigma \quad \Gamma, x : \tau \vdash e_1 : \tau' \quad \Gamma, y : \sigma \vdash e_2 : \tau'}{\Gamma \vdash \mathbf{case}\ e\ \{l \cdot x \Rightarrow e_1 \mid r \cdot y \Rightarrow e_2\} : \tau'}\ +E$$

The second premise would have type $\tau \to \tau'$, the third $\sigma \to \tau'$ and the conclusion has type τ'. Therefore we conjecture

$$\tau + \sigma \cong \forall\gamma.\, (\tau \to \gamma) \to (\sigma \to \gamma) \to \gamma$$

As a preliminary study, we can define

$$\begin{array}{lcl}
\mathit{sum}\,\tau\,\sigma & = & \forall\gamma.\, (\tau \to \gamma) \to (\sigma \to \gamma) \to \gamma \\
\mathit{inl} & : & \tau \to \mathit{sum}\,\tau\,\sigma \\
\mathit{inl} & = & \lambda x.\, \lambda l.\, \lambda r.\, l\, x \\
\mathit{inr} & : & \sigma \to \mathit{sum}\,\tau\,\sigma \\
\mathit{inr} & = & \lambda y.\, \lambda l.\, \lambda r.\, r\, y \\
\mathit{case_sum} & : & \mathit{sum}\,\tau\,\sigma \to \forall\gamma.\, (\tau \to \gamma) \to (\sigma \to \gamma) \to \gamma \\
\mathit{case_sum} & = & \lambda s.\, s
\end{array}$$

Then we verify the expected reductions

$$\begin{array}{lcl}
\mathit{case_sum}\ (\mathit{inl}\ x)\ z_1\ z_2 & \mapsto & (\mathit{inl}\ x)\ z_1\ z_2 \\
& \mapsto & (\lambda l.\, \lambda r.\, l\, x)\ z_1\ z_2 \\
& \mapsto^2 & z_1\ x \\
\mathit{case_sum}\ (\mathit{inr}\ y)\ z_1\ z_2 & \mapsto^* & z_2\ y
\end{array}$$

5 Predicativity

First, we summarize the language and rules of the polymorphic λ-calculus, sometimes referred to as System F, in its extrinsic formulation.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha.\, \tau \\ e & ::= & x \mid \lambda x.\, e \mid e_1\, e_2 \end{array}$$

$$\frac{\Delta, \alpha\ \mathit{type}\ ;\ \Gamma \vdash e : \tau}{\Delta\ ;\ \Gamma \vdash e : \forall \alpha.\, \tau}\ \forall I^{\alpha} \qquad \frac{\Delta\ ;\ \Gamma \vdash e : \forall \alpha.\, \tau \quad \Delta \vdash \sigma\ \mathit{type}}{\Delta\ ;\ \Gamma \vdash e : [\sigma/\alpha]\tau}\ \forall E$$

Several objections may be made to this system. A practical objection is the aforementioned undecidablity of the typing judgment. A philosophical objection is that the system is *impredicative*, that is, the domain of quantification includes the quantifier itself. The latter can be addressed by stratifying the language of types into simple types and type schemas.

$$\begin{array}{lrcl} \text{Simple types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \ldots \\ \text{Type schemas} & \sigma & ::= & \forall \alpha.\, \sigma \mid \tau \end{array}$$

This simple stratification allows type inference using an algorithm due to Robin Milner [Mil78], which adopts a previous algorithm by Roger Hindley for combinatory logic [Hin69].

The decomposition into simple types and type schemas is the core of the solution adopted in functional languages such as OCaml, Standard ML, Haskell and even object-oriented languages such as Java where polymorphic functions are implemented in so-called *generic methods* and classes.

The system of type schemes can be further extended (while remaining *predicative*) by considering a hierarchy of *universes* where the quantifier ranges over types at a lower universe. Systems with dependent types such as NuPrl or Agda employ universes for the added generality and sound type-theoretic foundation.

References

[Boe85] Hans Boehm. Partial polymorphic type inference is undecidable. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 339–345. IEEE, October 1985.

[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.

[Hin69] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal Of Computer And System Sciences*, 17:348–375, August 1978.

[Pfe93] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993. Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.

[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.

[Wel94] J. B. Wells. Typability and type checking in the second-order lambda-calculus are equivalent and undecidable. In *Proceedings of the 9th Symposium on Logic in Computer Science (LICS'94)*, pages 176–185, 1994.