Polymorphic lambda calculus

Kim B. Bruce
Princeton University & Williams College

November 19, 1998

1 introduction

We need a richer lmabda calculus in order to talk about object-oriented languages. In fact, this richer calculus is needed even to talk about languages like ML and Ada that support parametric polymorphism. In the next few sections we will define and provide the type-checking and computation rules for the polymorphic lambda calculus, an extension of the explicitly-typed lambda calculus that includes both parametric polymorphism and existential types.

2 Review of typed lambda calculus

In your midterm exam you defined the type-checking rules for a typed version of PCF, and extension of the explicitly-typed lambda calculus. Here we define a slightly different extension of this calculus, Λ_{rr}, which includes records and references (variables).

We being by defining the set of types in Λ_{rr}.

Definition 2.1 *Let $\mathcal{L}$ be an infinite collection of labels. The type expressions of Λ_{rr} are given by the following grammar.*

$$\tau ::= N \mid void \mid \tau_1 \rightarrow \tau_2 \mid \textbf{\textit{ref}}\ \tau \ \mid \{l_1 : \tau_1; \ldots; l_n : \tau_n\}$$

N is intended to be a type constant (like *Integer*, *Double*, etc.). It is included so that we have some type to start with. *Void* is also a constant type, designed to be used as the return type of commands and as the parameter type for parameterless functions. It will have only one trivial value. The other types represent function types, reference types (the types of variables), and record types.

We next define the collections of *pre-terms* of Λ_{rr}. They are called pre-terms because the only valid terms are those which type-check properly:

$$e ::= x \mid c \mid () \mid \lambda(x:\tau).\ e \mid e\ e' \mid \{l_1 = e_1 : \tau_1, \ldots, l_n = e_n : \tau_n\} \mid e.l_i \mid val\ x \mid x\ := \ e \mid e; e'$$

In the above definition, x denotes a variable, c is a constant of type N, while the term () is a constant with intended type *void* which will be used as an argument for parameterless functions. The terms involving function definition and application are familiar from the lambda calculus. The next two terms are record formation and record field extraction. If x is a reference (variable), then it will denote its location, while *val* x will denote the value stored at that location. The last two terms are really statement formation rules, but for simplicity we treat them as expressions in the same way that C and C++ do. If x is a reference, $x := e$ denotes the assignment of e to x. Finally $e; e'$ indicates the sequencing of the two statements. Its intended meaning is similar to that of ML: do the first statement for the side effect and then return the value of the second. Obviously we can sequence as many expressions as we like.

We could have added many other terms to the language like *if-then-else* expressions, loop constructs, etc. However it will be most convenient to work with a small language. However we will add these extra features without comment when we do examples.

The type checking rules for these terms require information about the types of free variables in a term.. A variable type assignment, E, is a finite set of associations between variables and type expressions of the form $x : \tau$, where each x is unique in E and τ is a type. If the relation $x : \tau \in E$, then we sometimes write $E(x) = \tau$.

$$E \cup \{x : \tau\} \vdash x : \tau$$

$$E \vdash c : N$$

$$E \vdash () : void$$

$$\frac{E \cup \{x : \sigma\} \vdash e : \tau}{E \vdash \lambda(x : \sigma).e : \sigma \to \tau}$$

$$\frac{E \vdash e : \sigma \to \tau, \quad E \vdash e' : \sigma}{E \vdash e\ e' : \tau}$$

$$\frac{E \vdash e_i : \tau_i, \quad for\ 1 \leq i \leq n}{E \vdash \{l_1 = e_1 : \tau_1, \ldots, l_n = e_n : \tau_n\} : \{l_1 : \tau_1; \ldots; l_n : \tau_n\}}$$

$$\frac{E \vdash e : \{l_1 : \tau_1; \ldots; l_n : \tau_n\}}{E \vdash e.l_i : \tau_i} \text{ for } 1 \leq i \leq n$$

$$\frac{E \vdash x : \texttt{ref}\ \tau}{E \vdash val\ x : \tau}$$

$$\frac{E \vdash x : \texttt{ref}\ \tau, \quad E \vdash e : \tau}{E \vdash x := e : void}$$

$$\frac{E \vdash e : \tau, \quad E \vdash e' : \tau'}{E \vdash e; e' : \tau'}$$

As usual, these rules can be read algorithmically by starting with the bottom left side and proceeding clockwise around the rule. Thus to find the type of a pre-term of the form $e\ e'$ with respect to a variable type assignment E, first find the type of e and the type of e'. If the type of e is of the form $\sigma \to \tau$ and the type of e' is σ, then the type of the application is τ. Otherwise no rule applies and the type-checking algorithm fails.

The collection of terms of Λ_{rr} with respect to E is the set of pre-terms which can be assigned types with respect to the type-assignment axioms and rules.

Currently our calculus is quite restricted. It is easy to show that if we use the typed versions of the α and β conversion rules then all terms of the typed lambda calculus converge to a normal form (i.e., one in which no further reductions – aside from simple α-conversions – can be made). In particular, we cannot define a fixed point operator in Λ_{rr}. It is clear that $Y = \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$ cannot be assigned a type (what type could x have so that $(x\ x)$ is well-typed?), but the result that all terms converges shows there cannot be a recursive operator, since we can easily define non-converging recursive functions. If desired, we can get around this by postulating the existence of a type fixed-point operator Y_σ for each type σ. We won't bother to do this now, but instead introduce a polymorphic fixed-point operator in the next section.

As with the untyped lambda calculus, we will sometimes find it convenient to write *let* $x = e'$: τ *in* e, which will be treated as an abbreviation for $(\lambda(x : \tau).e)\ e'$.

3 Polymorphic lambda calculus

The polymorphic lambda calculus, Λ_{rr}^P, extends our Λ_{rr} in the previous section by adding functions which are parameterized over types and by adding a mechanism to hide information in a term. We begin with parametric polymorphism. Later in this section we add the existential types and terms.

We add two new forms of type expressions:

$$\tau \ ::= \ t \mid \forall t.\tau$$

In each of these, t represents a type variable. The expression $\forall t.\tau$ represents the type of a function which takes a type variable t and returns a value of type τ (where t may occur in τ).

We add three new pre-terms to our language:

$$e ::= \ Y \mid \Lambda t.\ e \mid e[\sigma]$$

Y is a polymorphic version of the fixed point combinator of the untyped lanbda calculus. We discuss it in more depth below. The second term is a function taking a type parameter, while the third represents the application of a polymorphic function to a type parameter.

The type-checking rules for these new terms are given below:

$$\frac{E \vdash e : \tau}{E \vdash \Lambda t.e : \forall t.\tau}$$

$$\frac{E \vdash e : \forall t.\tau}{E \vdash e[\sigma] : \tau[\sigma/t]}$$

$$\vdash Y : \forall t.(t \to t) \to t$$

The first two rules are similar to those for regular functions. The main twist is that after the application of a polymorphic function, e, to a type expression, σ, the type of the result is obtained by replacing all occurrences of the type variable t in the polymorphic type by σ. Rewriting rules analogous to α and β can be added to the calculus.

The typing rule for Y is supposed to suggest its use aas a fixed point operator. We also add the following computation rule. If G has type $\tau \to \tau$, then:

$$Y[\tau]G = G(Y[\tau]G)$$

In other words, rather than defining an explicit term which turns out to give a fixed point, we simply add a rule which guarantees that for any type τ, the term $Y[\tau]$ always returns the fixed point of a function from $\tau \to \tau$.

Here are a few simple examples of terms of this language:

$$PID = \Lambda t.\ \lambda(x : t).x : \forall t.t \to t$$

is the polymorphic identity function. In particular, *PID[N]* has type $N \to N$ and denotes the identity function on elements of type *N*. Interestingly, it can be applied to its own type $PID[\forall t.t \to t]$ to get the identity function on $\forall t.t \to t$. Once we have this, we can also apply that specialized identity function to PID. That is,$PID[\forall t.t \to t]PID$ is a well-typed term of the language. Just as self-application in the untyped lambda calculus seemed to contradict classical mathematics, the application of a term to its function and then itself is also inconsistent with classical mathematics. (By contrast the calculus Λ_{rr} presented earlier has simple classical set-based models that live entirely within classical mathematics.)

Natural numbers are terms of the form:

$$\underline{n} = \Lambda t.\lambda(f : t \to t).\lambda(x : t).f^n x$$

where f^n is an abbreviation for f composed with itself n times. Other operations like addition and multiplication, as well as boolean values and operations that we defined for the untyped lambda calculus also have corresponding forms in the polymorphic lambda calculus.

If

$$factgen = \lambda(f : int \rightarrow int).\lambda(x : int).if\ x = 0\ then\ 1\ else\ x * f(x - 1)$$

(using some obvious extensions to Λ^P_{rr}), then

$$factorial = Y[int \rightarrow int]factGen : int \rightarrow int$$

is the usual factorial function in the obvious extension to the calculus with ints and if-then-else expressions.