Understanding Object-Oriented Languages: Semantics and Types

Kim B. Bruce
Princeton University & Williams College

November 24, 1998

1 Introduction

In this document we hope to explore the foundations of object-oriented programming languages by providing a way of understanding these languages via a translation to an extended typed lambda calculus. We will start with a quite simple encoding of objects and classes, and then see that support for such features as subclasses and parametric polymorphism will push us into more complex encodings. Moreover these encodings will suggest the addition of new extensions (e.g., `MyType`) to object-oriented programming languages as well as help us understand the type-checking rules of these languages.

2 An Extended Lambda Calculus

We need a richer lambda 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 first an extended typed lambda calculus with pairs, records, and references, and then the polymorphic lambda calculus, an extension of the explicitly-typed lambda calculus that includes both parametric polymorphism and existential types.

2.1 Review of typed lambda calculus with pairs, records and references

In your midterm exam you defined the type-checking rules for a typed version of PCF, an extension of the explicitly-typed lambda calculus. Here we define a slightly different extension of this calculus, Λ_{rr}, which includes pairs, 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.*

$$\tau ::= N \mid void \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \tau_2 \mid \{l_1 {:} \tau_1; \ldots; l_n {:} \tau_n\} \mid \mathtt{Ref}\ \tau$$

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, pair 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:

$$\begin{aligned} e ::= {} & x \mid c \mid () \mid \lambda(x\colon\tau).\ e \mid e\ e' \mid \langle e, e'\rangle \mid fst(e) \mid snd(e) \mid \\ & \{l_1 = e_1\colon\tau_1, \ldots, l_n = e_n\colon\tau_n\} \mid e.l_i \mid \texttt{ref}\ e \mid val\ x \mid x := \ e \mid e; e' \end{aligned}$$

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. Next come terms representing the pairing of terms and the extraction of the first and second components from pairs. Following these are terms for 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 $x\colon\tau$, where each x is unique in E and τ is a type. If the relation $x\colon\tau \in E$, then we sometimes write $E(x) = \tau$. The type-checking rules for Λ_{rr} are given in Figure 1.

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.

The α and β conversion rules of the untyped lambda calculus have equivalents in the typed lambda calculus. These and similar rules for the other types can be found in Figure 2.

Currently our calculus is quite restricted. It can be shown that if we use the given computation 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:* $\tau =$ *e' in e*, which will be treated as an abbreviation for $(\lambda(x\colon\tau).e)\ e'$.

2.2 Polymorphic lambda calculus

The polymorphic lambda calculus, Λ^P_{rr}, 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

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

$$E \vdash c : N$$

$$E \vdash () : void$$

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

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

$$\frac{E \vdash e : \tau, \quad E \vdash e' : \tau'}{E \vdash \langle e, e' \rangle : \tau * \tau'}$$

$$\frac{E \vdash e : \tau * \tau'}{E \vdash fst(e) : \tau}$$

$$\frac{E \vdash e : \tau * \tau'}{E \vdash snd(e) : \tau'}$$

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

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

$$\frac{E \vdash e : \tau}{E \vdash \texttt{ref}\ e : \texttt{Ref}\ \tau}$$

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

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

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

Figure 1: Type-checking rules for Λ_{rr}

α $$\lambda x.e \Rightarrow \lambda y.e[y/x], \text{ if } y \text{ does not occur free in } e.$$

β $$(\lambda x.e)e' \Rightarrow e[e'/x], \text{ if no free variable of } e' \text{ is bound in } e.$$

π_1 $$fst(\langle e, e' \rangle) \Rightarrow e$$

π_2 $$snd(\langle e, e' \rangle) \Rightarrow e'$$

ξ $$\{l_1 = e_1{:}\,\tau_1, \ldots, l_n = e_n{:}\,\tau_n\}.l_i \Rightarrow e_i$$

ρ $$(val\ (\texttt{ref}\ e)) \Rightarrow e$$

ρ $$(x{:}= e; val\ x) \Rightarrow e$$

Figure 2: Computation rules for Λ_{rr}

begin with parametric polymorphism. Later in this section we add the existential types and terms.

2.2.1 Parametric polymorphism and universal types

We add two new forms of type expressions:

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

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

We add three new pre-terms to our language:

$$e ::= \ \ldots \mid \Lambda t.\ e \mid e[\sigma] \mid \mathbf{Y}$$

The first term is a function taking a type parameter, while the second represents the application of a polymorphic function to a type parameter. $\mathbf{Y}$ is a polymorphic version of the fixed point combinator of the untyped lambda calculus. We discuss it in more depth below.

The type-checking rules for these new terms are given in Figure 3. 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 $\mathbf{Y}$ is supposed to suggest its use as a fixed point operator. We also add the following computation rule. If G has type $\tau \rightarrow \tau$, then:

$$\mathbf{Y}[\tau](G) \Rightarrow G(\mathbf{Y}[\tau]G)$$

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

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

$$\vdash \mathbf{Y}\colon \forall t.(t \to t) \to t$$

Figure 3: Type-checking rules for polymorphic terms in Λ_{rr}^{P}

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 $\mathbf{Y}[\tau]$ applied to a function of type $\tau \to \tau$ returns the fixed point of that function.

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

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

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 type 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:

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

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

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

(using some obvious extensions to Λ_{rr}^{P}), then

$$factorial = \mathbf{Y}[int \to int](factGen)\colon\ int \to int$$

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

Later we will find it convenient to use functions from types to types in order to define polymorphic programs. For instance, we may have a polymorphic program which performs operations on a stack. It will be convenient to be able to write a definition *Stack(T)* which can be instantiated as *Stack(Integer)*, *Stack(Real)*, *Stack(Point)*, etc. (Again we are freely extending the lambda calculus to use types available in existing languages.)

2.2.2 Information hiding and existential types

Sometimes we wish to use abstraction to hide information outside of a particular context. This was one of the main reasons for defining modules, but information hiding is also of great importance in the use of object-oriented languages. For example, we generally want to hide instance variables from being visible from outside of an object.

We can actually go back to our environment-based interpreter for a good example of where this information hiding is useful. We evaluated functions to closures, packages including the function as well as the environment it was evaluated in. An alternative way to handle closures is to add the environment as a parameter to the function before packaging up the pair. Thus if f is a function of type $\sigma \rightarrow \tau$ and the environment, ρ is a record of type $\gamma = \{x_1\colon \gamma_1; \ldots; x_n\colon \gamma_n\}$, then we can define $f' = \lambda(e\colon \gamma) f$ of type $\gamma \rightarrow \sigma \rightarrow \tau$. We now create the record *clos* $= \{$*func* $=$ *f'*,*env* $= \rho\}$ of type {func: $\gamma \rightarrow \sigma \rightarrow \tau$; env: γ}.

We can apply a closure o an actual parameter by grabbing the first component of the closure, applying it to the second component, and then applying it to the parameter. If *clos* defined above is our closure and *arg* is the actual parameter, then we compute *clos.func (clos.env) (arg)*, giving an answer of type τ, as desired. Unfortunately the type of the closure itself is quite complicated, involving the type γ of the environment.

One of the important recent innovations in compiler design and implementation is the recognition of the advantages of using typed intermediate languages during the compilation process. When we discussed the operation of compilers earlier in the course, we mentioned that after the program was parsed and type-checked, the compiler generated code for a very simple intermediate language before performing optimizations, assigning registers, etc., and finally generating object code (typically assembly or machine language targeted for the particular architecture).

Traditionally, the type information has been discarded when the intermediate code is generated. After all, the source level type-checking has allowed (some) incorrect code to be flagged, disambiguated overloaded operations, and provided information on how much memory must be allocated for variables and other constructs. However researchers have recently discovered that there are several advantages to keeping type information around in the intermediate language:

1. Type-checking can help catch compiler errors. Unfortunately it is not currently possible to prove that a compiler is correct, i.e., that it correctly translates source code to the equivalent intermediate code. Just as type checkers at the source level are good at catching (some) errors made by the programmer in source code, type checkers on the intermediate code can catch errors made by the compiler.

2. More optimizations can be recognized and performed if type information is available. These include many of the data flow analyses designed to determine where definitions are used, dead code analyses, etc.

For example, the current SML of New Jersey compiler has a back end (FLINT) which uses a typed intermediate language.

An intermediate language will normally represent functions as closures of the form described above. If we are compiling a functional language (and even many imperative languages) we may be passing functions as parameters. As we have seen, at run-time we will actually be passing closures as parameters. Of course what really matters about these closures is that they represent functions of type $\sigma \rightarrow \tau$, NOT the particular type of the environment, γ. In fact, different closures representing functions of type $\sigma \rightarrow \tau$ will have environment components of different types γ. How,

$$\frac{E \vdash e\colon \tau[\sigma/t]}{E \vdash \textit{pack } \langle \sigma, e\rangle \textit{ as } \exists t.\tau\colon \exists t.\tau}$$

$$\frac{E \vdash e\colon \exists t'.\tau, \quad E \cup \{x\colon \tau[t/t']\} \vdash e'\colon \sigma}{E \vdash \textit{open } e \textit{ as } \langle t, x\rangle \textit{ in } e'\colon \sigma} \; \textit{where } t \textit{ does not appear in } \sigma$$

Figure 4: Type-checking rules for existentials in Λ^P_{rr}

then, can we type these closures so that they will all be acceptable for formal parameters of type $\sigma \to \tau$?

The key is that we don't really need to know the type, γ, of the environment in order to apply the closure to the argument. The type γ never escapes to the rest of the program; the terms with types involving γ are used in such a way that its value doesn't matter. As a result, rather than writing the type of the closure as $\{\textit{func: } \gamma \to \sigma \to \tau\textit{; env: } \gamma\}$, it would be more convenient to write it as $\exists Y.\ \{\textit{func: } Y \to \sigma \to \tau\textit{; env: } Y\}$.

The value of this notation is that it gives us just enough information to determine that the value in the *func* field can be applied to the value in the *env* field in order to get back a function from σ to τ. It is important that this type does not reveal too much information. Thus it will be possible to later analyze the closure, perhaps determine that some of the information in the environment is never used, and hence discard it. We might also discover that some of the information in the environment is fixed at compile time, and partially apply the function to those bits of the environment. Both of these optimizations result in a change to the type of the environment and to the parameter type of the function in the argument. However, because we have hidden the type of the environment, replacing a closure by a "thinned" version will not have an impact on the type of the closure visible from the outside.

Now all we need is some notation allowing us to "pack" up and hide certain information in order to get an expression of existential type. We add the new type expression:

$$\tau ::= \ \ldots \mid \exists t.\tau$$

as well as expressions to pack up and open existentials:

$$e ::= \ \ldots \mid \textit{pack } \langle \sigma, e\rangle \textit{ as } \exists t.\tau \mid \textit{open } e \textit{ as } \langle t, x\rangle \textit{ in } e'$$

The first expression takes a value v and creates a packed version of existential type by hiding certain occurrences of σ, and replacing them with the type variable t. The second expression "opens" an existential, revealing that it is a pair of a type and value, and uses that type and value in the expression e'.

The typing rules in Figure 4 are a bit more complex than the earlier ones. The first rule is relatively straightforward. One can simply replace as many occurrences of σ as one would like in the type of e by a type variable, and report the packed version of the element as being of an existential type. The second is a bit more subtle. The term e is of an existential type, so it has been packed as a pair. The *open* term reveals the pair which can be used in e'. However, statically e' doesn't have access to the exact type of the value x in the pair. Thus we need to be able to determine the type of e' without knowing anything about the type variable t, aside from the fact that it represents a type and that the type of x is $\tau[t/t']$ for some t. The side condition on the rule ensures that the type t does not escape from the scope of the "open" term. This is essential both because we want the type to remain hidden, and, because outside the expression, t is a free type variable with no meaning associated with it.

Computation with existentials is relatively straightforward in that if you first pack something and then open it, it is as if you used it directly in a let statement. Thus,

$$open\ (pack\ \langle \sigma, e \rangle\ as\ \exists t.\tau)\ as\ \langle t', x \rangle\ in\ e' \Rightarrow e'[\sigma/t', e/x]$$

There is also a rule similar to α allowing one to change the names of bound variables in open expressions.

Let's see how this would work with closures. Suppose we have the function $f = \lambda(x{:}int).\ x + a$ defined in an environment $\{a = 3\}$. We first rewrite f to take the environment as a parameter: $f' = \lambda(e{:}\{a{:}int\}).\lambda(x{:}int).\ e.x + a$. We form the closure $clos = \{func = f',\ env = \{a{=}3\}\}$ by hiding the type of the environment. We now pack it up as: $packClos$ = pack $\langle\{a{:}int\}, clos\rangle$ as $\exists t.\ \{func{:}\ t \rightarrow int \rightarrow int;\ env{:}\ t\ \}$ by hiding the type of the environment.

We can apply $packClos$ to the argument 7 by opening it and applying the function first to the environment and then to the argument: $apply(packClos,\ 7) = open\ packClos\ as\ \langle \gamma, clos' \rangle\ in\ clos'.func(clos'.env)(7)$. Using the computation rules introduced above, this simplifies to $(\lambda(e{:}\{a{:}int\}).\lambda(x{:}int).\ x + e.a)\ (\{a{=}3\})\ (7) = 7 + 3 = 10$.

In the next section we use this machinery we have built up to explain objects and classes.