Understanding Object-Oriented Languages: Semantics and Types

Kim B. Bruce
Princeton University & Williams College

December 14, 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.*

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

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:

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

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 $x{:}\,\tau$, where each x is unique in E and τ is a type. If the relation $x{:}\,\tau \in E$, then we sometimes write $E(x) = \tau$. 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 $\sigma \rightarrow \tau$ 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:* τ = *e' in e*, which will be treated as an abbreviation for $(\lambda(x{:}\,\tau).e)\ e'$.

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

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

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

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

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

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

ρ $(x\colon= e; val\ x) \Rightarrow e$

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. **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 **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 $\gamma = \{x_1{:}\,\gamma_1; \ldots; x_n{:}\,\gamma_n\}$, then we can define $f' = \lambda(e{:}\,\gamma)f$ 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 $\sigma \rightarrow \tau$?

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 $\{$*func:* $\gamma \rightarrow \sigma \rightarrow \tau$*; env:* $\gamma\}$, it would be more convenient to write it as $\exists Y.$ $\{$*func:* $Y \rightarrow \sigma \rightarrow \tau$*; 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 $\exists t.\ \{func{:}\ t \to int \to int;\ env{:}\ t\ \}$ 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

```
(λ(e:{a:int}).λ (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.

3 A simple model of objects and classes

In this section we provide a first approximation of a translation of objects and classes into the language Λ^P_{rr}. Later we will modify this translation in order to take into account more of the complexities of object-oriented programming.

In this and the following sections we will use concrete examples of classes in order to illustrate the ideas. Rather than committing to a particular existing programming language, we use a simple notation that is a cross between Pascal and Java.

With this notation, we will presume that all instance variables are hidden from clients (*protected* in Java and C++ notation) and all methods are visible (*public*). Later we will refine this notation so that we can allow the hiding of methods as well.

Our first example is a class which represents a cell holding an integer.

```
class CellClass{
   x: int = 0;

   get(): int { return self.x }
   set(nuVal:int): void { self.x := nuVal }
   bump(): void { self.set(self.get()+1) }
}
```

As in Java and C++, we can distinguish between instance variables and methods in the declarations because methods have a (possibly empty) parameter list. The keyword `self` is used within methods to refer to the object executing the method at runtime. Accesses to local instance variables and other local methods within a method body are prefixed with `self`. While the prefix could be omitted, we will find it handy to include it when translating to Λ^P_{rr}. We do not discuss constructors at this time. Instead we presume that all instance variables are initialized in their declarations and that each class has a parameterless constructor.

3.1 Modeling objects

Earlier, when we discussed the implementation of object-oriented languages in real systems, we noted that a convenient representation for objects was as a record of its instance variables along with a pointer to the method suite of the object. When a message was sent to an object, it found the appropriate method and then applied it using the object's instance variables.

We can abstract this into a more formal encoding of the structure of objects. Let `IV` be the type of the record of instance variables of the object and let `M` be the type of the record of methods of a class. For example, if we were modeling object generated by the class `CellClass` above, the type of the record of instance variables is `CellIV = {x: int ref }` while the type of the record of methods is `CellM = {get: void → int; set: int → void; bump: void → void}`. The types of instance variables are always `ref` types, because they are updatable in methods.

Using this notation, we denote the type of an object with instance variables of type `IV` and methods of type `M` as

```
VOT(IV,M) = IV × (IV → M)
```

Note the similarity to the actual implementation of objects in computers. If `o` is an object with type `VOT(IV,M)`, and message name `m` is a label in `M`, we can send message `m` with parameters `params` to `o` as follows:

```
o.m(params) = (snd(o) fst(o)).m(params)
```

That is, just as in the real implementation, we take the method suite, apply it to the instance variables, extract the appropriate method, and finally apply it to the actual parameters.[1]

Similarly, it is easy to extract the instance variable from an object of type `VOT(IV,M)` where `x` is a label in `IV`:

```
o.x = (fst(o)).x
```

This looks promising, but we have not correctly modeled the fact that instance variables should be hidden outside of the object. If the type of the object displays the type of the instance variables, then clients can get access to the instance variables of the object. However, we introduced existential types exactly so that we could hide this type of information.

The only change we need to make to our representation of objects is to change the type of objects so that the type of the instance variables is hidden from outside the object. We let

```
OT(M) = ∃Y. Y × (Y → M)
```

Notice that `IV` is no longer a parameter of the object type constructor since it has now been hidden by the use of the existential type.

[1]Clearly we could interchange the order of applying to instance variables and extracting the method field with very little change in our model. We chose the order above mainly for notational simplicity, though one could also make the argument that if several messages are being sent to the same object, it is simpler to apply the entire method suite to the instance variables once and then select whatever methods one wants without needing to reapply the methods to the suite of instance variables. (Note that because the instance variables are references, any changes in the values stored there will be visible in methods.)

We read `OT(M)` as the object type with method suite of type `M`, while `VOT(IV,M)` is the "visible" object type with instance variables of type `IV` and methods of type `M`. We can transform objects from the visible object type into the preferred object type using the following function, `close: VOT(IV,M)` $\rightarrow$ `OT(M)`:

```
close[IV,M] = λ(vo:VOT(IV,M)). pack ⟨IV, vo⟩ as OT(M)
```

Note that `close` is a polymorphic function of type

```
close: ∀IV. ∀M. VOT(IV,M) → OT(M)
```

Summarizing, we can construct an object by building the pair of the record of instance variables and the record of methods (where the methods are parameterized by the instance variables), and then closing them to form an existential which protects the instance variables from being visible from outside of the object.

Sending a message now involves one more step, in that the object must be opened before the method suite can be applied to the instance variables:

```
o.m(params) = open o as ⟨IV, vo⟩ in (snd(vo) fst(vo)).m(params)
```

As long as the type `IV` of the record of instance variables does not occur in either the parameter or return types of methods, then the above expression will be well typed.

Going back to our `CellClass` example, we see that the type of objects generated by `CellClass` is

```
CellType = OT(CellM) = ∃Y. Y × (Y → CellM).
```

In particular, if `o` has type `OT(CellM)`, then `o.set(7)` will have type `void` and `o.get()` will have type `int` as expected.

3.2 Modeling classes

While we now know how to represent objects, we have not yet talked about how classes can be used to create objects. An important part of this modeling is to figure out how to use the initial values of instance variables and methods provided in the class in order to create the "visible" objects, which can then be packed up to hide the instance variables.

Using the initial values of instance variables is pretty straightforward, but getting the methods in proper form will be a bit more difficult.

The bodies of methods in classes may refer to the keyword `self`, which represents the receiver of the message. This keyword may be understood as an extra parameter of these messages. We expect that when we instantiate new objects using the class, `self` will be resolved as a fixed point representing the object being defined. As we have seen in our `CellClass` example, `self` refers both to the instance variables and methods of the object. Thus it seems plausible that the type of `self` should be a visible object type rather than `OT(M)`. Thus the type of `self` will be assumed to be `VOT(IV,M)`.[2]

Let `inst` of type `IV` be the record of initial values of instance variables, and let `meth` of type `M` be the record of method bodies (which may include the keyword `self`). In order to determine the type of the methods, we use the assumption stated above that the type of `self` is `VOT(IV,M)`.

[2]Later on, we will see that a somewhat more general type for `self` is useful, but this will work fine for now.

It is important to note that the type `IV` differs from that used with objects in that here `IV` is the type of a record of initial values, while for objects `IV` stood for a record of variables. In an object generated from the class, the type of the record of instance variables is obtained by adding a `ref` to the type of each field of `IV`.

To simplify notation we define the type $\mathtt{IV}^{\mathtt{ref}}$ as follows. If $\mathtt{IV} = \{\mathtt{l}_1{:}\,\tau_1; \ldots; \mathtt{l}_n{:}\,\tau_n\}$, let $\mathtt{IV}^{\mathtt{ref}} = \{\mathtt{l}_1{:}\mathtt{Ref}\tau_1; \ldots; \mathtt{l}_n{:}\mathtt{Ref}\tau_n\}$. Similarly, if $\mathtt{inst} = \{\mathtt{l}_1 = \mathtt{i}_1; \ldots; \mathtt{l}_n = \mathtt{i}_n\}$, let $\mathtt{inst}^{\mathtt{ref}} = \{\mathtt{l}_1 = \mathtt{ref}\ \mathtt{i}_1; \ldots; \mathtt{l}_n = \mathtt{ref}\ \mathtt{i}_n\}$.

With this notation we can represent the class as a pair:

$$\mathtt{class(inst,meth)} = \langle \mathtt{inst},\ \lambda(\mathtt{self{:}VOT(IV^{ref},M)}).\mathtt{meth}\rangle$$

The type of a class is of the form:

$$\mathtt{ClassType(IV,M)} = \mathtt{IV} \times (\mathtt{VOT(IV^{ref},M)} \rightarrow \mathtt{M})$$

Notice the initial values of the instance variables have type `IV`, while the visual object type involves type $\mathtt{IV}^{\mathtt{ref}}$ because it includes variables rather than just values.

In order to create an object from the class, it appears that all we have to do is to create references for the instance variables, initialize those references with the values in the class, and create a pair of this record of instance variables and the method suite. However, we have one problem. The method suite for objects takes only the record of instance variables (of type $\mathtt{IV}^{\mathtt{ref}}$) as a parameter, while the method suite for classes takes an entire object of type $\mathtt{VOT(IV^{ref},M)}$ as a parameter.

This differences makes sense because subclasses might override the definition of a method, so we would not like to have the meanings of methods frozen to be their definitions in the class. We need to leave open the possibility that the meaning of `self.m` will be different in a subclass than in the class.

Nevertheless we must resolve this inconsistency for objects, where the meanings of the methods are fixed. Luckily this is not difficult to do using fixed points. Given the suite of methods `meth` of a class with type $\mathtt{VOT(IV^{ref},M)} \rightarrow \mathtt{M}$, we create `meth'` of type $\mathtt{IV^{ref}} \rightarrow \mathtt{M}$ more appropriate for an object as follows:

$$\mathtt{meth'(inst'{:}IV^{ref})} = \mathtt{meth}(\langle \mathtt{inst'},\mathtt{meth'}\rangle)$$

More formally,

$$\mathtt{meth'} = \mathtt{Y}[(\mathtt{IV^{ref}}\rightarrow\mathtt{M})\rightarrow(\mathtt{IV^{ref}}\rightarrow\mathtt{M})]\lambda(\mathtt{fm{:}IV^{ref}}\rightarrow\mathtt{M}).\lambda(\mathtt{inst'{:}IV^{ref}}).\mathtt{meth}(\langle \mathtt{inst'},\mathtt{fm}\rangle)$$

Now we have all the pieces necessary to create a new object from a class:

$$\mathtt{new\ class(inst,meth)} = \mathtt{close}[\mathtt{IV^{ref}},\mathtt{M}]\langle \mathtt{inst^{ref}},\mathtt{meth'}\rangle$$

for $\mathtt{inst}^{\mathtt{ref}}$ and `meth'` as defined above.

Let's go back to our `CellClass` definition to make all this formalism more concrete. From the definition, we see that

$$\mathtt{inst} = \{\mathtt{x = 0}\}\mathtt{:\ CellIV}$$

and

```
meth = λ(self:VOT(CellIV^ref,CellM)
           {get = λ(_:void). self.x,
            set = λ(nuVal:int). self.x := nuVal,
            bump = λ(_:void). self.set(self.get()+1}:CellM
```

Thus class `CellClass` will be represented by `class(inst,meth)` for `inst` and `meth` defined above.

Let's now examine what happens when we create new object from class `CellClass`. We begin by creating a record of locations holding the initial values obtained from the class.

$$\texttt{inst}^{\texttt{ref}}\ \texttt{= \{x = ref 0\}: CellIV}^{\texttt{ref}}$$

Next we rewrite the method suite by "compiling in" the method suite to `self`:

```
meth' = λ(ir:CellIV^ref).
            {get = λ(_:void). val ⟨ir,meth'⟩.x,
             set = λ(nuVal:int). ⟨ir,meth'⟩.x := nuVal,
             bump = λ(_:void). ⟨ir,meth'⟩.set(⟨ir,meth'⟩.get()+1}:CellM
```

Notice that we needed to add the `val` in the definition of `get` so that we returned the value of the instance variable `x` rather than the reference or location. As above, our new object is given by $\texttt{close[IV}^{\texttt{ref}}\texttt{,M]}\langle\texttt{inst}^{\texttt{ref}}\texttt{,meth'}\rangle$.

To get a better sense of the meaning of the method suite, `meth'`, of this new object, we simplify the method bodies a bit further. Because $\langle\texttt{inst}^{\texttt{ref}}\texttt{,meth'}\rangle$ is of type $\texttt{VOT(CellIV}^{\texttt{ref}}\texttt{,CellM)}$ we can use the definitions from the last section on sending messages and extracting instance variables. *In general there would also be message sends to other objects in these methods, but in this example we do not have any.*

Using the definitions and simplifying, we get

```
meth' = λ(ir:CellIV^ref).
            {get = λ(_:void). val ir.x,
             set = λ(nuVal:int). ir.x := nuVal,
             bump = λ(_:void). meth'(ir).set(meth'(ir).get()+1}:CellM
```

Of course, we could go further in simplifying the code for `bump`, replacing terms of the form `meth'(ir).m` by their definitions. If we do this then the body of `bump` becomes:

```
bump = λ(_:void). ir.x := val ir.x +1
```

just as expected.

Exercise 3.1 *Evaluate `o.bump()` if o is an object of type `OT(CellM)` which was created through a call of `new Cell()` and whose x instance variable currently has value 47. Recall that you must open the existential before sending the message (and that opening a packed element gives you access to the details of its innards - see the computation rule at the end of section 2.2.2). Also recall that `:=` denotes a change to the contents in memory, i.e., an update in place.*

In the next section we investigate how well this representation works for defining subclasses.

3.3 Modeling subclasses

Now that we have presented our encoding of classes, we investigate how to encode subclasses. Before doing that, however, we'd best understand the legal changes to types of methods and instance variables that can be made in going from a superclass to a subclass.

3.3.1 Why restrict type changes in subclasses?

Why should there be any restrictions on redefining instance variables and methods in subclasses? The problem is that the type-safety of the methods of a class depends both on the types of the instance variables and on the types of the other methods.

Let class `C` have methods `m` and `n` as follows:

```
class C {
   ...
   m(): void {... self.n(arg) ...}
   n(farg:D): E {...}
}
```

In the above we assume that `arg` has type `D` and `self.n(arg)` occurs in a context which expects a value of type `E`.

Now suppose we define class `SC`:

```
class SC {
   n(farg:D'): E' {...}
```

Are there any restrictions on the changes we can make to the types of the arguments and return type of method `n` in the subclass?

Again, method `m` of the superclass uses `n` with an argument `arg` of type `D` in a context expecting a result of type `E`. Since `m` is inherited unchanged in `D`, when message `m` is sent to an object generated from `D`, it will execute the same code. In particular, it will generate a message send of `n` to `self`. However, the method `n` now has type `D'` $\rightarrow$ `E'`, rather than `D` $\rightarrow$ `E`.

If the application of `self.n(arg)` is still going to be legal, the new `n` must be applicable to arguments of type `D`. In other words, we must have `D` $<:$ `D'`. Similarly, the results of the new `n` must be usable in the same contexts as the old, so `E'` $<:$ `E`. Of course, this is just the rule we learned a while back for subtyping of function types: `D'` $\rightarrow$ `E'` $<:$ `D` $\rightarrow$ `E` iff `D` $<:$ `D'` and `E'` $<:$ `E`.

Thus, in defining subclasses, we may only change the return types of methods in a covariant way (e.g., by subtypes) and parameter types in a contravariant way. C++, for example, allows programmers to change the return types in methods in a covariant way. It is extremely difficult to come up with a compelling example of the use of contravariant changes in parameter types in subclasses. As a result, we know of no popular languages which allow parameter types to change.

Given the above reasoning, you will not be surprised that the rules for changing the types of instance variables in subclasses are exactly the same as for subtyping of variable or reference types. That is, no changes are allowed to the types of instance variables in going from a superclass to a subclass.

Exercise 3.2 *Write down an example similar to that above which shows that allowing covariant changes to the types of instance variables will result in type errors. That is, write a class that is well-typed, but a subclass which is not well-typed only due to the change to the type of an instance variable.*

Suppose class `C` has type `Class(IV,M)` and `C'` is a subclass with type `Class(IV',M')`. Recall that `IV` represents the type of the record of initial values of instance variables, while $\mathtt{IV}^{\mathtt{ref}}$ represents the type of a record of instance variables. Because `C'` is a subclass, `IV'` may add new instance variables, but may not change the types of existing instance variables. This is equivalent to saying that $\mathtt{IV'}^{\mathtt{ref}} <: \mathtt{IV}^{\mathtt{ref}}$, since reference types have no subtypes.

Similarly we may add new methods in `C'`, but overridden methods may only change return types in a covariant way and parameter types in a contravariant way. Thus $\texttt{M'} <: \texttt{M}$.

In summary, if class `C` has type `Class(IV,M)` and `C'` is a subclass with type `Class(IV',M')`, then $\texttt{IV'}^{\texttt{ref}} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M'} <: \texttt{M}$.

Notice that we have not *proved* that contravariant changes to parameter types and covariant changes to return types of methods in subclasses will result in subclasses free of type errors. We have only shown that other changes *can* result in type errors. Only when we have finished our encoding will we be able to make such an assertion.

3.3.2 A first try at encoding subclasses

We begin by using the simple encoding of the last section and seeing where we run into trouble. We use the notation $\texttt{class inherit c (inst}_{sub}\texttt{,meth}_{sub}\texttt{)}$ to represent a subclass formed by inheriting the instance variables and methods from `c` and then adding and/or overriding with instance variables and methods from $\texttt{inst}_{sub}$ and $\texttt{meth}_{sub}$.

Suppose $\texttt{c} = \texttt{class(inst,meth)}$, and thus has encoding

$$\texttt{c = }\langle\texttt{inst, }\lambda\texttt{(self:VOT(IV}^{\texttt{ref}}\texttt{,M)).meth}\rangle$$

To avoid type safety problems of the sort discussed in the previous section, we assume that if $\texttt{inst}_{sub}$ redefines the initial value of an instance variable in `inst`, then they have the same type. If a method in `meth` is redefined in $\texttt{meth}_{sub}$, then the redefined method must have a type which is a subtype of the original.

If `m` and `n` are records, then we write $\texttt{m} \oplus \texttt{n}$ for right dominant record combination. That is, if both `m` and `n` have a field with the same label `l` then the value from `n` is associated with `l` in the combination.

Let $\texttt{inst}_{all} = \texttt{inst} \oplus \texttt{inst}_{sub}$ have type $\texttt{IV}_{all}$. Similarly, let $\texttt{meth}_{all} = \texttt{meth} \oplus \texttt{meth}_{sub}$ have type $\texttt{M}_{all}$. Mimicking the encoding of classes from the previous section, one would anticipate that we could encode the subclass as follows:

$$\texttt{class inherit c (inst}_{sub}\texttt{,meth}_{sub}\texttt{) = }\langle\texttt{inst}_{all},\ \lambda(\texttt{self:VOT(IV}_{all}{}^{\texttt{ref}},\texttt{M}_{all})).\texttt{meth}_{all}\rangle$$

Unfortunately, this simple encoding will not work for subclasses because of typing problems with inherited methods..

The problem is that when we type-checked the methods of `meth`, we presumed that the type of `self` was `VOT(I,M)`, whereas now we are using the inherited methods in a context where the type of `self` is $\texttt{VOT(IV}_{all}{}^{\texttt{ref}}\texttt{,M}_{all}\texttt{)}$. This would be fine if $\texttt{VOT(IV}_{all}{}^{\texttt{ref}}\texttt{,M}_{all}\texttt{)} <: \texttt{VOT(IV}^{\texttt{ref}}\texttt{,M)}$, but that is NOT the case.

Recall that

$$\texttt{VOT(IV}^{\texttt{ref}}\texttt{,M) = IV}^{\texttt{ref}} \times \texttt{(IV}^{\texttt{ref}} \rightarrow \texttt{M)}$$

Thus for $\texttt{VOT(IV}_{all}{}^{\text{ref}}\texttt{,M}_{all}\texttt{)} <: \texttt{VOT(IV}^{\texttt{ref}}\texttt{,M)}$, we will need $\texttt{IV}_{all}{}^{\text{ref}} <: \texttt{IV}^{\texttt{ref}}$, and $\texttt{IV}_{all}{}^{\text{ref}} \rightarrow \texttt{M}_{all} <: \texttt{IV}^{\texttt{ref}} \rightarrow \texttt{M}$. The first is no problem since $\texttt{IV}_{all}{}^{\text{ref}}$ is simply an extension of $\texttt{IV}^{\texttt{ref}}$. Recall that subtyping for function types is covariant in the return type and contravariant in the domain type. The return types are in the subtype relation, as $\texttt{M}_{all}$is an extension of `M`, but the domains change in a covariant way, rather than contravariant. That is, we need $\texttt{IV}^{\texttt{ref}} <: \texttt{IV}_{all}{}^{\text{ref}}$ for the function types be subtypes, but we have the reverse.

The result is that the methods in `meth` may not be type safe when used in a context in which `self` has type $\texttt{VOT(IV}_{all}{}^{\text{ref}}\texttt{,M}_{all}\texttt{)}$. What does this imply about the type-safety of languages like Java and C++? Luckily, not very much.

Let's look at an example showing what can go wrong when you use a parameter, `self`, of type `VOT(IV`$_{all}$`ref`,`M`$_{all}$`)` in a context which is type-correct when `self` has type `VOT(IV`$^{\texttt{ref}}$`,M)`.

Let `oself` be a record of type {`j:Ref int`}. Define the class `c` as follows:

```
class C{
   j:int;
   m():void {⟨oself,snd(self)⟩.n()}
   n():int {val self.j}
}
```

The method `m` of `C` builds a new object of type `VOT(IV`$^{\texttt{ref}}$`,M)` by replacing the first component of `self` by `oself`, and then sends it the message `n`. In other words, when method `n` is executed, it is with `self` interpreted by this new object whose instance variables are now represented by `oself`. All the method `n` of `C` does is to return the value of the instance variable `j`. This is no problem at all in `C` as `oself` has a field `j` of type `int`.

Now define subclass `SC` of `c` as follows:

```
class SC inherit c{
   k:int;
   n():int {val self.k}
}
```

Now we have a problem when method `m` is sent to an object generated from class `SC`. Since `m` is not overridden, the inherited code is run, resulting in sending a message `n` to an object whose method suite is the same (i.e., the full method suite of `n`), but whose instance variables have been replaced by `oself`. Thus when the evaluation of method `n` results in an attempt to access `self.k`, an error will result, as `oself` does not have field of type `k`.

Exercise 3.3 *Write out the translation of classes `C` and `SC` in detail, evaluate `new SC` to get an object, and then send that object the message `m` in order to show that a type error results.*

At this point, you are probably objecting that this is a cheat, as no object-oriented language would actually allow you to write a method like `m` above. Luckily, that is correct, for if it was legal, virtually all existing statically type-safe languages would allow type errors. Object-oriented languages do just not allow you to take apart `self` and then build a new object with a different collection of instance variables.

In fact we can only generate type errors with our encoding if we allow the methods to refer to the record of instance variables as a whole. If we only are allowed to access individual instance variables and methods of self, no errors result.

As a result, we could restrict the typed lambda calculus used here to disallow such access and then try to prove the correctness of our translation. However, it is quite difficult to restrict manipulations of `self` in the translation without also restricting other useful constructs in our underlying lambda calculus. Thus, rather than restricting the underlying calculus, we choose instead to modify our encoding of classes so that it will be type safe no matter what manipulations of `self` are made in the method bodies.

4 Rethinking classes to obtain type-safe inheritance

Let us rethink the definition of class from first principles. The problem is that we type-checked the methods presuming that the type of `self` was the same as the (visible) type of the objects generated by the class. Yet, these methods may be inherited in subclasses which generate objects having more instance variables and methods (let alone the fact that existing methods may be overridden). It's no wonder that we got into trouble!

4.1 Defining classes

If we want the methods in classes to be able to be inherited in subclasses without encountering new errors, we should type check them under the assumption that they may be included in any subclass of the class being defined. Of course, at the time we are defining a class, we have no idea what subclasses may later be defined. Though we showed earlier that if class `C` has type `Class(IV,M)` and `C'` is a subclass with type `Class(IV',M')`, then $\texttt{IV'}^{\texttt{ref}} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M'} <: \texttt{M}$.

We can use this to define the encoding of classes in such a way that inherited methods may be used in a type-safe way in subclasses. As before, let `inst` of type `IV` be the record of initial values of instance variables for a class, and let `meth` of type `M` be the record of method bodies (which may include the keyword `self`). We modify the definition of class to explicitly consider the possibility of subclasses by abstracting over the types of the records of instance variables and methods.

$$\texttt{class(inst,meth) = }\Lambda\texttt{(M'}<:\texttt{M).}\ \Lambda\texttt{(IV'}<:\texttt{IV}^{\texttt{ref}}\texttt{).}\ \langle\texttt{inst,}\ \lambda\texttt{(self:VOT(IV',M')).meth}\rangle$$

As noted earlier, the types of the records of instance variables and method suites of subclasses will be subtypes of the superclass, so we need only abstract over those types. (While we have not formally defined this type of bounded quantification, we hope that its use will be clear here.) The type of a class is also modified:

$$\texttt{ClassType(IV,M) = }\forall\texttt{M'}<:\texttt{M.}\ \forall\texttt{IV'}<:\texttt{IV}^{\texttt{ref}}\texttt{.}\ \texttt{IV} \times \texttt{(VOT(IV',M')} \rightarrow \texttt{M)}$$

The record of initial values of instance variables still has type `IV`, but the parameter `self` to the methods suite now has type `VOT(IV',M')`.

The creation of objects from classes is just a simple modification of our previous encoding. Apply the class to the type $\texttt{IV}^{\texttt{ref}}$ of its record of instance variables and `M`, the type of its record of methods, and then proceed exactly as before. Let `C` be a class of type `ClassType(IV,M)`. Then

$$\texttt{new C = close[IV}^{\texttt{ref}}\texttt{,M]}\langle\texttt{inst}^{\texttt{ref}}\texttt{,meth'}\rangle$$

where

$$\texttt{C[IV}^{\texttt{ref}}\texttt{][M] = }\langle\texttt{inst,}\ \lambda\texttt{(self:VOT(IV}^{\texttt{ref}}\texttt{,M)).meth}\rangle \quad \text{and}$$
$$\texttt{meth'(inst':IV}^{\texttt{ref}}\texttt{) = meth(}\langle\texttt{inst',meth'}\rangle\texttt{)}$$

as before.

Going back to our `CellClass` example, we have `CellClass` = `class(inst,meth)` which is

```
Λ(M' <: CellM). Λ(IV' <: CellIV^ref).
     ⟨{x = 0},
      λ(self:VOT(IV',M'))
           {get = λ(_:void). self.x,
            set = λ(nuVal:int). self.x := nuVal,
            bump = λ(_:void). self.set(self.get()+1}⟩
```

As noted earlier, the type of `self` is now `VOT(I',M')`. Thus in order to ensure that the methods suite is well-typed, we must type check the methods under the assumption that `self` has type `VOT(I',M')`, where all we know is that `IV'` <: $\texttt{IV}^{\texttt{ref}}$ and `M'` <: `M`.

This may seem restrictive, but in fact it provides us with exactly the information necessary to type-check accesses to `self`. Because `IV'` <: $\texttt{IV}^{\texttt{ref}}$, `self.x` has type `Ref` τ iff `x` has type τ in `IV`.

Similarly, `self.m(arg)` is guaranteed to be safe only if `m` is in every extension `M'` of `M`, and in each has a type $\sigma_{M'} \rightarrow \tau_{M'}$, where the type of `arg` is some subtype of all of these $\sigma_{M'}$. This is equivalent to having `m` in `M` with type $\sigma \rightarrow \tau$ and `arg` having type σ.

Exercise 4.1 *Prove this last remark characterizing when `self.m(arg)` is type safe.*

Thus it appears we haven't lost any power in type-checking by parameterizing over subtypes of the type of the instance variables and methods. However, this change does eliminate our problem code from the last section which broke subtyping.

Here is the example again:

```
class C {
   j:int;
   m():void {⟨oself, snd(self)⟩.n()}
   n():int {val self.j}
}
```

where `oself` has type `{j:int}`.

Because we may only assume that the type of `self` is `VOT(IV',M')`, the example no longer type-checks. In particular, `⟨oself,snd(self)⟩` has type `IV * (IV' → M')` rather than `IV * (IV → M)`. Because we only know `IV'` <: `IV`, we cannot apply the second component of the pair to the first component. As a result, we cannot send a message to this pair. The class `C` will not type-check, and hence is illegal.

4.2 Defining subclasses: Try 2

So far, so good. Let's go back and define subclasses and make sure that inherited methods really will type check.

Suppose `c` = `class(inst,meth)` has type `ClassType(IV,M)`, and let $\texttt{c}_{sub}$ = `class inherit c` $(\texttt{inst}_{sub}, \texttt{meth}_{sub})$. As before, we assume that if $\texttt{inst}_{sub}$ redefines the initial value of an instance variable in `inst`, then they have the same type, and if a method in `meth` is redefined in $\texttt{meth}_{sub}$, then the redefined method must have a type which is a subtype of the original. Define $\texttt{inst}_{all} = \texttt{inst} \oplus \texttt{inst}_{sub}$ with type $\texttt{IV}_{sub}$ and $\texttt{meth}_{all} = \texttt{meth} \oplus \texttt{meth}_{sub}$ with type $\texttt{M}_{all}$. By the assumptions on overriding initial values of instance variables and methods, $\texttt{IV}_{all}{}^{\texttt{ref}} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M}_{all} <: \texttt{M}$.

Define:

$$\texttt{class inherit c } (\texttt{inst}_{sub}, \texttt{meth}_{sub}) = \texttt{class}(\texttt{inst}_{all}, \texttt{meth}_{all})$$
$$= \Lambda(\texttt{M'} <: \texttt{M}_{all}).\ \Lambda(\texttt{IV'} <: \texttt{IV}_{all}{}^{\texttt{ref}}).\ \langle \texttt{inst}_{all},\ \lambda(\texttt{self:VOT(IV',M')}).\texttt{meth}_{all} \rangle$$

What about those inherited methods? Are they going to be type-correct? Recall that the methods in `meth` were type checked under the assumption that `self` had type `VOT(IV',M')` for `IV'` <: $\texttt{IV}^{\texttt{ref}}$ and `M'` <: `M`.

In the subclass the inherited methods are used where `self` has type `VOT(IV',M')` for `IV'` <: $\texttt{IV}_{all}{}^{\texttt{ref}}$ and `M'` <: $\texttt{M}_{all}$. However, as we noted earlier, $\texttt{IV}_{all}{}^{\texttt{ref}} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M}_{all} <: \texttt{M}$. By transitivity, `IV'` $<:$ `IV`$^{\texttt{ref}}$ and `M'` $<:$ `M`, and we know that the inherited methods type check under that assumption.

In summary, in the subclass, the assumptions on `IV'` and `M'` are stronger than in the superclass, so anything that type-checks under the original, weaker assumptions, is guaranteed to type check under the stricter assumptions in the subclass. As a result, we have now succeeded in modeling objects, classes, and subclasses in such a way that inherited methods are guaranteed to remain type-correct if they were type-correct in the superclass.

4.3 A complication with `self`

Perhaps save this discussion until after discuss rigidities in typing systems.

There is one complication with `self` that we have not yet dealt with. Suppose we have a method which returns `self`. In type-checking methods of a class with type `ClassType(IV,M)`, we assumed that the type of `self` was `VOT(IV',M')` where `IV'` $<:$ `IV`$^{\texttt{ref}}$ and `M'` $<:$ `M`. If the method returns `self`, then it returns an element of type `VOT(IV',M')`.

However, it makes no sense to write `VOT(IV',M')`, as the return type of the method, as this will mean nothing to the user. Even when we create a new object and replace `IV'` by `IV`$^{\texttt{ref}}$ and `M'` by `M`, we cannot use `IV`$^{\texttt{ref}}$ outside of the object. This is forbidden by our typing rules for existentials because `IV`$^{\texttt{ref}}$ is hidden in the existential.

We solve this problem by applying `close` to `self` (or any other object of type `VOT(IV`$^{\texttt{ref}}$`,M)` before returning it or using it as a parameter. Recall that `close[IV`$^{\texttt{ref}}$`,M](self)` has type `OT(M)`, which may occur as a type.

One last complication is that the type of `self` in a method suite is taken as `VOT(IV',M')` for `IV'` $<:$ `IV`$^{\texttt{ref}}$ and `M'` $<:$ `M`. In order for a method returning `self` to type-check properly, we must replace `self` in the method body by `close[IV',M'](self)`, which has type `OT(M')`.

Even this is problematic as the type of a class is written as `ClassType(IV,M)`, and does not include mention of `M'`. Luckily, this can be taken care of by using the fact that if `M'` $<:$ `M`, then `OT(M')` $<:$ `OT(M)`. (This follows easily from the definition of `OT(M)` as an existential and the subtyping rule for existentials found in the homework.) Thus we can treat `close[IV',M'](self)` as though it has type `OT(M)`. Hence the return type of a method returning `self` in a class with type `ClassType(IV,M)` can be given as `OT(M)`. Similarly, `self` can be used as an actual parameter whenever the formal parameter has type `OT(M)`.

This solves our problem, though at the cost of losing type information. Let `C` be a class with type `ClassType(IV,M)` which has a method `m` which returns `self`. As noted above the return type of `m` will be `OT(M)`, the type of objects generated by `C`. Now suppose that `C'` is a subclass of `C` with type `ClassType(IV`$_{all}$`,M`$_{all}$`)` where `IV`$_{all}^{\texttt{ref}}$ $<:$ `IV`$^{\texttt{ref}}$ and `M`$_{all}$ $<:$ `M`. Assume that `C` does not override method `m`. Then the type checker will assume the return type of the inherited method `m` will be `OT(M)`, even though if it were overridden with exactly the same body, it would type check as having return type `OT(M`$_{all}$`)`.

This loss of information is similar to that found in Java, where `clone` is declared in `Object` with return type `Object`. Now in all subclasses, `clone` also must return `Object`. In our encoding, we can actually do a bit better. If the user is willing to override `clone` in a class, we can change its return type to be the object type corresponding to the class. However, inherited methods never have their types automatically modified.

5 Adding F-bounded parametric polymorphism

One of the frustrations for Java programmers is the lack of support for parametric polymorphism. As a result, programmers create data structures holding elements of type Object (or some other type guaranteed to be a supertype of all types of elements likely to be used).

For example, suppose you want to use the built-in java.util.stack. It's declaration is similar to:

```
public class Stack extends Vector{
   public Object push(Object item){...}
   public Object pop(){...}
   public Object peek(){...}
   public boolean empty(){...}
   public int search(Object o){...}
}
```

Suppose we wish to manipulate elements of type `Point` with the stack `myStack`. Creating the stack and inserting items is no problem:

```
Stack myStack = new Stack();;
Point aPoint = new Point(2.0,3.0);
myStack.push(aPoint);
```

The use of a `Point` as the parameter to `push` is well-typed because `Point` is a subtype of `Object`.

However, suppose we wish to remove the top element from the stack and then send it a `move` message. The result of evaluating `myStack.pop()` is statically an element of type `Object`, even though we know we pushed a `Point` on the stack. Because `Object` does not contain the method `move`, we must first cast the element to type `Point` before sending the `move` message:

```
Point stackPoint = (Point)(myStack.pop()).move(1.0,2.5);
```

Of course it is even worse if we wish to put elements of type `int` on the stack, as `int` is not an extension of `Object`. To push an integer `n` on the stack, we must push on the corresponding `Integer` via `myStack.push(new Integer(n))`. Popping off such an element is also painful as you must pop it off, coerce it to `Integer`, and then send it the `intValue()` message before being able to do anything with the value.

In summary, we have two major problems with the stack implementation using `Object`:

1. We can't enforce that the stack only hold items of a fixed type.

2. Whenever we remove something from the stack, we must cast it to the type we expect it to be so that we can send messages to it.

These two problems result in ugly code with lots of casts and leave the code open to errors if somehow an element of some other type is pushed onto the stack.

The Java extension, GJ, attempts to remedy this by adding parametric polymorphism to Java. In GJ we can rewrite Stack as:

```
public class Stack<Elt> extends Vector<Elt>{
   public Object push(Elt item){...}
   public Elt pop(){...}
   public Elt peek(){...}
```

```
    public boolean empty(){...}
    public int search(Elt o){...}
}
```

We can create and add an element to the stack as follows:

```
Stack<Point> myStack = new Stack<Point>();;
Point aPoint = new Point(2.0,3.0);
myStack.push(aPoint);
```

Pushing anything other than a `Point` onto the stack results in a static type error.

The code for popping a point off the stack and moving it is now much clearer, as we don't need casts.

```
Point stackPoint = myStack.pop().move(1.0,2.5);
```

GJ can also handle more sophisticated examples of parameterized classes in which we wish to restrict the collection of type parameters which can be used.

Suppose we wish to implement a class representing ordered lists. In order to insert an element, we must compare it with other elements in the list. As a result, we can only insert elements of classes which provide the appropriate methods comparing elements:

```
interface Comparable {
    public boolean equal(Comparable other);
    public boolean greaterThan(Comparable other);
    public boolean lessThan(Comparable other);
}
```

We can then write a class `OrderedList` as follows:

```
public class OrderedList<Elt implements Comparable> extends ... {
    public void insert(Elt item){...}
    public Elt removeFirst(){...}
    public boolean empty(){...}
    public int search(Elt o){...}
```

Because `Elt` is constrained to implement `Comparable`, we can send messages `equal`, `greaterThan`, and `lessThan` to terms of type `Elt`.

Now we can instantiate `OrderedList` with any class which implements `Comparable`. This sounds fine, but we soon discover that it is hard to write a class implementing `Comparable`. The difficulty is that the comparison operators have to take arguments of any other class implementing `Comparable`. Here is an example of the difficulties:

```
public class Simple implements Comparable {
    protected int x = 0;
    ...
    public boolean greaterThan(Comparable other){
        if (other instanceof Simple){
            return (x > ((Simple)other).x)
        } else { ... raise an exception??...}
    }
    ...
}
```

The code for `greaterThan` is now much more complex than expected. We must test if `other` is of type `Simple`, and then cast it to `Simple` before being able to do anything with it. And what do you do if the `instanceof` test fails?

Moreover, it is pretty clear that you could virtually never use a pre-existing class to instantiate `OrderedList`, because the comparison methods of existing classes are highly unlikely to take parameters of type `Comparable`. Instead they most likely will take parameters of the same type as the class being defined.

For example, `Simple` would normally be defined instead as `SimpleC`, below:

```
public class SimpleC implements Comparable {
   protected int x = 0;
   ...
   public boolean greaterThan(SimpleC other){
        return (x > other.x)
      }
   }
   ...
}
```

The bounded polymorphism illustrated above was invented in the mid-80's. However, by the end of the '80's, examples like the above showed that regular bounded quantification was not expressive enough to capture the restrictions on type parameters needed for many important examples which contained what came to be known as "binary methods". These are methods like `equal`, `greaterThan`, and `lessThan` in which the parameter types are expected to be the same as the class being defined (as they are in `SimpleC` above).

A technique known as F-bounded quantification was invented to handle these more complex cases. We illustrate the technique by example. Rather than defining the interface `Comparable`, we now define a parameterized interface, `ComparableF`:

```
interface ComparableF<T> {
   public boolean equal(T other);
   public boolean greaterThan(T other);
   public boolean lessThan(T other);
}
```

Any class `C` implementing `ComparableF<Elt>`, for type `Elt` must have methods `equal`, `greaterThan`, and `lessThan` which take parameters of type `Elt`.

Notice that the class `SimpleC` above implements `ComparableF<SimpleC>`. In general, if class `Elt` implements `ComparableF<Elt>` then it must support comparison operations which take parameters of type `Elt`. This is a sort of recursive constraint in that the upper bound of `Elt` involves `Elt`, but there is no real difficulty in figuring out if an existing class satisfies such a constraint. You need only check that it has methods of the appropriate types.

We now can rewrite class `OrderedList` as:

```
public class BPOrderedList<Elt implements ComparableF<Elt>> extends ... {
   public void insert(Elt item){...}
   public Elt removeFirst(){...}
   public boolean empty(){...}
   public int search(Elt o){...}
}
```

The constraint on `Elt` is now much harder to read, but it does state exactly the constraints necessary to use `BPOrderedList`.

Because `SimpleC` satisfies these constraints we can instantiate `OrderedList` with `SimpleC`:

```
BPOrderedList<SimpleC> ordListSC = new BPOrderedList<SimpleC>();
SimpleC a = new SimpleC();
ordListSC.insert(a);
...
```

In summary, GJ's addition of F-bounded polymorphism to Java has greatly increased the expressiveness of Java's type system. Whereas before we had to use `Object` or another fixed type to represent a whole collection of possible types which could be used in a data structure, we can now instantiate the class to a specific type as long as it satisfies the given constraints.

Unfortunately, F-bounded quantification does not interact well with the class or subtype hierarchies in object-oriented languages. The problem is that if a class satisfies an F-bounded constraint, its subclasses will not satisfy the constraint. For example, define:

```
public class ComplexC extends SimpleC {
   ...
   public boolean greaterThan(SimpleC other){
        return (x > other.x) && ...
      }
   }
   ...
}
```

No matter how few additions or changes are made in the subclass (and even if no methods from the subclass are overridden), `ComplexC` will implement `ComparableF(SimpleC)`, not `ComparableF(ComplexC)`. As a result it cannot be used with `BPOrderedList`.

The difficulty is that we are not allowed to change the types of method parameters in subclasses. As a result, subclasses conflict with the use of F-bounded polymorphism. We will see later on that the use of `MyType` and match-bounded polymorphism gives similar expressiveness as F-bounded polymorphism, but *is* consistent with the subclass relation.

Several compromises were made in order to get GJ programs to run with the current crop of virtual machines (which do not support polymorphic classes). The implementation of parametric classes with GJ essentially proceeds by translating a parametric class into a standard Java class in which the type parameters are erased and replaced by the upper bound of the type parameter (or `Object` if there is no upper bound). Type casts are then automatically inserted in several places in the translated program in order for the resulting Java program to be type-safe. While all of the casts are guaranteed to succeed, the result is that the translated program ends up looking very similar to the way one would normally write the code in plain Java. Moreover, even though the casts are guaranteed to succeed, they will still be checked at run-time, slowing down the execution of the program.

Other complications arose in implementing GJ, especially with attempts to deal with arrays. I urge you to read the on-line tutorial at the GJ web site (see the link from the class handouts page) in order to get more details on GJ.

In summary, GJ's addition of F-bounded polymorphism to Java helps the programmer to express type constraints in a readable and type-safe way. While the implementation on current virtual machines will not run any faster than the usual ways of writing generic code in Java, future enhancements to the virtual machine should provide for fast, safe execution of GJ programs.

6 More (too) flexible typing with Eiffel

The programming language Eiffel was designed in the mid-80's by Bertrand Russell. It is a statically-typed class-based object-oriented language which supports multiple inheritance, bounded polymorphism, and executable assertions. Like Java, all objects are implicit references and garbage collection is built in.

Here is a portion of a sample Eiffel class.

```
class RATIONAL

inherit
   ORDEREDPAIR [INTEGER]
     rename  x as n, -- can change names of features
             y as d
     redefine same   -- indicates that same will be redefined
     end
   OTHERCLASS
     redefine ...
     end

creation
   CreateRat

feature{ALL}

    error: boolean;

    CreateRat is
            -- create a rational
        do
            d := 1
        end; -- Create

      set(numer, denom : INTEGER) is
            -- set the numerator and denominator
         require -- precondition
            denom /= 0
         local
            gcd : INTEGER
         do

            n := numer;
            d := denom;

            if d < 0 then
                n := -n;
                d := -d
            end;
```

```
            gcd := reduce;

            n := n // gcd;
            d := d // gcd
         ensure  -- postcondition
            d > 0
         end; -- set

      plus(other : like Current) : like Current is
         local
            sumnum, sumden : INTEGER;
         do
            sumnum := n*other.d + other.n*d;
            sumden := d*other.d;
            !!Result.Create;
            Result.set(sumnum,sumden)
         end; -- plus

      ...

   feature {NONE}  -- private method
      reduce : INTEGER is   featureALL
   end

   invariant d /= 0
```

The above example illustrates several features of Eiffel. First it is possible for a class to inherit from multiple classes. Methods in inherited classes can be renamed either for convenience or to resolve name clashes between classes inherited from. Methods that are overridden must be declared to be "redefined". This is required to avoid the problem of having a programmer accidentally override a method in a superclass. It is also possible to change the visibility of inherited methods with a declaration in the `inherit` clause, though we don't show it here.

Creation routines may be used as constructors. If `r` has type `RATIONAL`, then `!!r.CreateRat` will create a new object of type `RATIONAL`. Instance variables not explicitly initialized are given a default value (e.g., 0 or `void` - which corresponds to `null` in Java).

Features include both instance variables and methods. The `feature` clause indicates the visibility of items listed. The default is `ALL`, `NONE` corresponds to Java's private, while you can also list any collection of classes (e.g. "friends") which you wish to be able to have access to particular features. Instance variables are only available as read-only outside of the methods of the class. In fact, from outside the object it is impossible to distinguish between instance variables and parameterless procedures.

In method `set`, the precondition is given using keyword `require` and the postcondition uses `ensure`. The class invariant at the bottom of the class is preceded by `invariant`. It is also possible to specify loop invariants and "variants". A variant is an integer expression which is always positive, but which gets closer to zero after each iteration. It is used to help prove termination of the loop. The assertions can be labeled in order to give good error messages when an assertion fails. Determination of which kinds of assertions are checked at run-time is made by setting a compile-time flag. The default is the pre-conditions are always checked, but others are not.

`Current` is used in Eiffel as the name of the object executing the method rather than Java and C++'s `this` or Smalltalk's `self`. If `id` is a declared identifier, then `like id` can be used to stand for the static type of `id`. Thus the method `plus` in `RATIONAL` uses `like Current` as the type of the argument and result of the method. If `plus` is sent to an object of (static) type `RATIONAL`, then its actual parameter must also be of type `RATIONAL`, and the result will be of type `RATIONAL`.

If `plus` is inherited in a subclass, `SUBRATIONAL` of `RATIONAL`, then the meaning of `like Current` will change appropriately. That is, if `plus` is sent to an object of (static) type `SUBRATIONAL`, then its actual parameter must also be of type `SUBRATIONAL`, and the result will be of type `SUBRATIONAL`.

This is very handy and provides great flexibility in creating subclasses. For example, here are two classes providing for singly and doubly-linked nodes.The following are slightly simplified examples from the Eiffel structure library. They represent singly and doubly-linked nodes.

```
class LINKABLE [G]

feature

        item: G;

        right: like Current;  -- Right neighbor

        putRight (other: like Current) is
        -- Put ‘other’ to the right of current cell.
                do
                        right := other
                ensure
                        chained: right = other
                end;

end -- class LINKABLE
```

Now define the subclass:

```
class BILINKABLE [G] inherit

        LINKABLE [G]
                redefine
                        putRight
                end

feature -- Access

        left: like Current;
                        -- Left neighbor

        putRight (other: like Current) is
                        -- Put ‘other’ to the right of current cell.
                do
```

```
                        right := other;
                        if (other /= Void) then
                                other.simplePutLeft (Current)
                        end
                end;

        putLeft (other: like Current) is
                        -- Put 'other' to the left of current cell.
                do
                        left := other;
                        if (other /= Void) then
                                other.simplePutRight (Current)
                        end
                ensure
                        chained: left = other
                end;

        simplePutRight (other: like Current) is
                        -- set 'right' to 'other'
                do
                         right := other
                end;

        simplePutLeft (other: like Current) is
                        -- set 'left' to 'other' is
                do
                        left := other
                end;

invariant

        rightSymmetry:
                (right /= Void) implies (right.left = Current);
        leftSymmetry:
                (left /= Void) implies (left.right = Current)

end -- class BILINKABLE
```

This is something that seems very handy, as it is frustrating not to be able to write a doubly-linked node as a subclass of a singly-linked node is languages like Java and C++. But now suppose we write the following routine

```
   trouble(p, q : LINKABLE [RATIONAL] ) is
      do
         p.putRight(q);
         ....
      end
```

and suppose we have `sNode : LINKABLE [RATIONAL]` and `biNode: BILINKABLE [RATIONAL]`.

What happens if we write:

```
trouble(biNode,sNode)
```

If `BILINKABLE [RATIONAL]` is subtype of `LINKABLE [RATIONAL]`, then this should work, instead, BANG!!!!! - we get a system crash when the code for `putRight` in `BILINKABLE` attempts to send the message `simplePutLeft` to `sNode`.

Thus Eiffel is not statically type-safe. In fact, Eiffel allows a number of statically unsafe changes in creating subclasses. One is allowed to replace the type of an instance variable, method parameter, or method result type by a subtype of the one occurring in the superclass. We have seen earlier that only method result types can be changed in a covariant way if one wishes to guarantee type safety. Eiffel also allows programmers to hide inherited methods in a subclass that were visible in the superclass, creating yet another type insecurity.

This type insecurity was first pointed out in print in 1989 by William Cook. Bertrand Meyer has had various responses over time. He has consistently claimed that the insecurity is inconsequential because Eiffel users do not write code that would result in this error. That is, while they use features like these, they automatically use them correctly. He also claimed that proposed fixes like eliminating `like` declarations and instead using bounded polymorphism forced programmers to "plan ahead" too much and would result in programmers having to go back and rewrite superclasses.

After a while of hearing further criticism of this, he then proposed a "system validity check" to ensure that these errors did not occur at run-time. Essentially the idea was to perform a conservative dataflow analysis at link-time to make sure no unfortunate combinations of types would arise at run-time due to interactions of the various classes. None of the Eiffel compiler suppliers ever implemented this check. It is not known why they did not implement it, but it is likely that either the test was too expensive to run or it ruled out too many useful programs.

Even if this test had worked, programs would have been quite fragile with respect to the test. A program could pass the system validity check, but then the addition of a new class could cause the system validity check to fail due to an unexpected interaction that was only barely related to the new class.

Three years ago, Meyer proposed an alternative "fix" to Eiffel's static type safety problems called "No polymorphic cat-calls". A call is polymorphic if its target can represent objects of more than one type. According to Meyer: "A routine is a CAT (Changing Availability or Type) if some redefinition changes its export status or the type of one of its arguments. A call is a catcall if some redefinition of the routine would make it invalid because of a change of export status or argument type." His proposal is to ban "catcalls" to terms which can represent objects of more than one type. Essentially this requires that if the type of a method parameter is changed in a subclass, then you must know the exact type of any object which is a target of the corresponding message.

Meyer announced this fix with great fan-fare in an invited talk to OOPSLA 3 years ago. Three days later I found a remaining type insecurity in the language relating to changing the types of instance variables in subclasses. He has since fixed that problem. I do not know if his current proposal results in a statically type-safe language. However, I believe that his proposal is so restrictive that programmers will not be able to write useful programs that both use the dangerous features and pass his test. To this date, no Eiffel compiler implements this proposal.

In spite of its static typing problems, Eiffel was quite an advance over existing object-oriented languages. Its "design-by-contract" philosophy, which encourages the use of pre-conditions and post-conditions may be its most lasting legacy. Like Smalltalk, which is dynamically typed, it also showed how useful certain constructs could be that were not available in other statically-typed languages.

Eiffel had a chance to emerge as an important object-oriented language when experienced object-oriented programmers started losing faith in C++ as a tool to build reliable complex systems, but its type problems and slow compilers and compiled code worked against it. Ironically it lost to Java which, at least in the early days, ran significantly slower than Eiffel.

Eiffel was also one of the first commercially available compilers for object-oriented languages which supported bounded polymorphism. Here is an example.

```
deferred class COMPARABLE -- deferred is the same as abstract in Java
feature specification
    lessThan (other: like Current): BOOLEAN is
        deferred;

    greaterThan (other: like Current): BOOLEAN is
        deferred;
end -- class COMPARABLE
```

Now define:

```
class INTORD inherit COMPARABLE
feature
    value:INTEGER;
    lessThan (other:like Current) is
        do
            Result := value < other.value
        end;
    ...
end -- class INTORD
```

Clearly `INTORD` implements `COMPARABLE`. We can use `COMPARABLE` as an upper bound for polymorphic classes:

```
class Sorting[T -> COMPARABLE]
feature
    sort(thearray:ARRAY[T]):ARRAY[T] is
        local  ....
        do
            ......
            .... if thearray.item(i).lessThan(thearray.item(j)) ....
        end;
end -- Sorting
```

The notation indicates that the type variable `T` must be a subclass of `COMPARABLE`. Notice how much simpler this notation is than the notation for F-bounded quantification.

In summary, while Eiffel's static type system does not catch all type errors, it does suggest constructs that would be quite useful in other statically-typed object-oriented languages. In particular, the use of `like Current` seems to add a great deal of expressiveness to the language. In the next section we investigate the possibility of adding a similar construct in a type-safe way.

7 Adding `MyType` to object-oriented languages

In the previous section we saw how useful Eiffel's `like Current` construct could be in helping programmer expressiveness. The main problem was that because it resulted in covariant changes to method parameter types and instance variables, it could lead to type insecurities.

In this section we go back to our encoding of classes to examine in more detail the problems with typing `self` first discussed in section ??. This will lead us to introduce a keyword, `MyType`, to stand for the type of `self`. It will be equivalent to Eiffel's `like Current`, but we will see how to statically type-check this construct in a way that guarantees type safety. The interesting point here is that understanding our previous encoding of objects and classes will lead us to a statically type-safe extension of object-oriented languages that will allow this style of programming. We will also be able to read off of the encoding the type-checking rules for this enriched language.

Let's take a brief side-track to discuss the use of class names in object oriented languages. Most familiar object-oriented languages use class names to stand for the names of classes, their constructors (e.g., as in `new C()`), and also as the types of objects generated by the class. We won't actually worry about constructors here, but we will take pains to distinguish between a class and the type of objects generated by the class. As a useful naming default, if `C` is the name of a class, then we will usually use `CType` to stand for the type of objects generated by the class. In our examples we will tend to use only interfaces as types in order to accentuate this difference.

Recall that our new translation of `C` = `class(inst,meth)` is

$$\Lambda(\mathtt{M'} <: \mathtt{M}).\ \Lambda(\mathtt{IV'} <: \mathtt{IV}^{\mathtt{ref}}).\ \langle \mathtt{inst},\ \lambda(\mathtt{self{:}VOT(IV',M')}).\mathtt{meth}\rangle$$

with type

$$\mathtt{ClassType(IV,M)} = \forall \mathtt{M'} <: \mathtt{M}.\ \forall \mathtt{IV'} <: \mathtt{IV}^{\mathtt{ref}}.\ \mathtt{IV} \times (\mathtt{VOT(IV',M')} \rightarrow \mathtt{M})$$

Using the notation suggested above for the types of objects generated by a class, the type of objects generated by `C` is `CType` = `OT(M)`.

In type-checking the method suite, `meth`, we assume that `self` has type `VOT(IV',M')` for `M'` <: `M` and `IV'` <: $\mathtt{IV}^{\mathtt{ref}}$. As we mentioned earlier, we can "close" `self` with `close[IV',M'](self)` to get an element of type `OT(M')`. In section ?? we noted that we could then use subsumption and the fact that `OT(M')` <: `OT(M)` = `CType` to deduce that `close[IV',M'](self)` has type `CType`.

Consider the following interface and class:

```
interface CType{
   returnMe():CType;
}

class C implements CType{
   x:int = 0;
   returnMe():CType {return self}
}
```

Because of the remarks above, the method `returnMe` in `C` can safely be given the return type of `CType`.

However, as we noted earlier, this results in the loss of typing information in subclasses. In particular if `D` is a subclass of `C` which inherits `returnMe` (i.e., does not override it), then `returnMe` still has return type `CType` rather than the more accurate `DType`. If we override `returnMe`, we are allowed to change the type in a covariant way (at least in some languages – we know it will be safe), but if we attempt to call `super.returnMe()` in the redefinition in `D`, we run into the same problems.

7.1 Introducing MyType

As we saw above, once we closed `self`, we obtained an element of type `OT(M')`. This is the most reasonable type to use for the return type of `returnMe`, as we don't want the outside world to be able to get access to the object's instance variables.

Unfortunately, the programmer writing a class does not have access to `OT(M')`, since `M'` only occurs in the translation of the class. Let's help out the programmer by creating a name for it. Since `OT(M')` is the best approximation for the type of `self`, let's give it the name `MyType`. Thus a programmer can write:

```
class C {
   x:int = 0;
   returnMe():MyType {return self}
}
```

Intuitively, if `cobj` is an object created from `C` (i.e., with type `CType`), then `cobj.returnMe()` will return an object of type `CType`.

However, the value of this shows more clearly when we define subclasses.

```
class D inherit C {
   ...
}
```

If `dobj` is an object created from `D` (i.e., with type `DType`), then `dobj.returnMe()` will return an object of type `DType`.

The keyword `MyType` will play the same role in our language as `like Current` does in Eiffel. For example, if we have a method `Clone` with signature `Clone():MyType` defined in the "top" class (e.g., class `Object` in Java), then it will be available in all subclasses and will always return an object whose static type is the same as the receiver.

Moreover, we can now write some of the same useful programs as we saw in Eiffel. Here is a simplified version of Eiffel's `Linkable` example:

```
class Node{
   item: int = 0;
   next: MyType = null;  -- Right neighbor

   setNext (other: MyType): void { next := other;}
   getNext (): MyType { return next;}
   setItem (newItem: int): void { item := newItem;}
   getItem ():int  { return item;}
}

class DbleNode inherit Node modifying setNext {
   previous: MyType = null;  -- Right neighbor

   setPrevious (other: MyType): void { previous := other;}
```

```
    getPrevious (): MyType { return previous;}
    setNext (other: MyType): void {
       next := other;
       if (other != null) then other.setPrevious(self);}
}
```

But what about type safety? We saw in the last section that assuming `DbleNodeType` <: `NodeType` resulted in a type error. We certainly don't want to have that happen to us. We will avoid that error by *not* assuming that subclasses always generate subtypes.

Let's take a closer look at the types of these classes and the objects that they generate. When we looked at the meanings of methods, we pulled `self` out as a parameter because it varied as we went from superclass to subclass. Similarly, we will pull out `MyType` from the types or interfaces of the instance variables and methods. Thus will write the interfaces of the instance variables and method suites of `Node` and `DbleNode` as parameterized types:

```
NodeIV(MT) = { item: int; next: MT}
NodeM(MT) =  { setNext: MT → void; getNext: void → MT;
               setItem: int → void; getItem: void → int }

DbleNodeIV(MT) = { item: int; next: MT; previous: MT}
DbleNodeM(MT) =  { setNext: MT → void; getNext: void → MT;
                   setPrevious: MT → void; getPrevious: void → MT;
                   setItem: int → void; getItem:void → int }
```

We can define subtyping between functions from types to types pointwise. That is, `F` <: `G` iff for all types `T`, `F(T)` <: `G(T)`. Looking at the definitions above, we see that $\texttt{DbleNodeIV}^{\texttt{ref}}$ <: $\texttt{NodeIV}^{\texttt{ref}}$ and `DbleNodeM` <: `NodeM`. In fact, if we maintain the restrictions on not allowing (explicit) changes to the types of instance variables, and only allowing method types to be replaced by subtypes in subclasses, we will always have these relations between the corresponding types of instance variables and of method suites.

We now must make sense of `MyType` in objects. This is relatively straightforward. If the object is generated by `Node`, and thus has type `NodeType`, the meaning of `MyType` in that object will just be `NodeType`. Similarly if an object has type `DbleNodeType`, then the meaning of `MyType` in that object will be `DbleNodeType`.

At this point you may be wondering why we don't just use `NodeType` *or* `DbleNodeType` *instead of* `MyType`*. Remember the reason is so that we end up with an accurate type of* `self` *that changes automatically in subclasses!*

Given the above, it shouldn't come as a surprise that the meaning of `MyType` will be determined by a fixed-point. We will use the notation μ`T. F(T)` for the fixed point of type function `F`. In particular, if `fixF` $= \mu$`T. F(T)`, then `fixF` $=$ `F(fixF)`. This is exactly the same equation satisfied by fixed points of regular functions. We won't bother to make up a special **Y** operator for type functions, but will just use the μ notation.

If `M(MyType)` is the type of the method suite of a class `C` and `IV(MyType)` is the type of its collection of initial values of instance variables, then we define:

```
OT(M) = μX. ∃Y. Y × (Y → M(X))
      = ∃Y. Y × (Y → M(OT(M)))
```

Notice that the meaning of `MyType` is set by the fixed point operator to be exactly `OT(M)`, as expected.

The types of objects "from the inside" are exactly as before, except that all occurrences of `MyType` are replaced by `OT(M)`.

$$\texttt{VOT(IV}^{\texttt{ref}}\texttt{,M) = IV}^{\texttt{ref}}\texttt{(OT(M))} \times \texttt{(IV}^{\texttt{ref}}\texttt{(OT(M))} \rightarrow \texttt{M(OT(M)))}$$

The meaning of `MyType` is exactly the same as in the definition of `OT(M)` (namely the meaning *is* `OT(M)`), the only thing that is different from the definition of `OT(M)` is that the instance variables (whose types may mention `MyType`) are visible.

Classes are defined similarly to before. The only difference is that both the instance variables and methods may involve `MyType`.

$$\texttt{class(inst,meth) = }\lambda\texttt{M'} <: \texttt{M. }\lambda\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}\texttt{. <inst, }\lambda\texttt{self:VOT(IV',M').meth>}$$

which has type:

$$\texttt{ClassType(IV,M) = }\forall\texttt{M'} <: \texttt{M. }\forall\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}\texttt{. IV(OT(M'))} \times \texttt{(VOT(IV',M')} \rightarrow \texttt{M(OT(M')))}$$

Notice that `self` is treated as having type `VOT(IV',M')`, for $\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}$ and $\texttt{M'} <: \texttt{M}$. Therefore the type of `close[IV'(OT(M')),M'(OT(M'))](self)` is `OT(M')`. Thus in both instance variables and methods, `MyType` is interpreted as `OT(M')`.

Subclasses are defined exactly as before. We might be concerned that methods inherited in subclasses need not be guaranteed to be type-safe, but the same argument as before shows that if the methods are type safe in the superclass they are guaranteed to be type safe in the subclass. Interestingly, the use of `MyType` has no impact on that argument.

In the object, the meaning of `MyType` will be `OT(M)`, so we will apply both `IV` and `M` to `OT(M)` in order to get the final types of instance variables and methods. As before, we create a new object by building a new collection of instance variables, $\texttt{inst}^{\texttt{ref}}$, whose initial values are given by the class, and by defining a corresponding collection of methods which only take those instance variables as parameters. Let

$$\texttt{meth'(ir:IV}^{\texttt{ref}}\texttt{(OT(M))) = meth[}\langle\texttt{ir,meth'}\rangle\texttt{/self]:IV}^{\texttt{ref}}\texttt{(OT(M))} \rightarrow \texttt{M(OT(M))}$$

Then,

$$\texttt{new class(inst,meth) = close[IV}^{\texttt{ref}}\texttt{(OT(M)),M(OT(M))] }\langle\texttt{inst}^{\texttt{ref}}\texttt{,meth'}\rangle\texttt{: OT(M)}$$

We send messages exactly as before.

Thus adding a `MyType` construct to our language was straightforward. We simply parameterized our instance variables and methods by a type variable that we intend to eventually specialize to `MyType`. In classes, the `MyType` slot is filled with `OT(M')`, which represents possible values of `MyType` for subclasses. When we form an object, `IV'` gets specialized to $\texttt{IV}^{\texttt{ref}}$ and `M'` gets specialized to `M`, resulting in the meaning of `MyType` being fixed as `OT(M)`. Thus the meaning of `MyType` in the object is exactly the type of the object, as desired.

Let's return to our example with the `Node` class. The type of the objects generated by `Node` is written `NodeType`, which we now know is defined as:

$$\begin{aligned}\texttt{NodeType} &= \texttt{OT(NodeM)}\\ &= \mu\texttt{X. }\exists\texttt{Y. Y} \times \texttt{(Y} \rightarrow \texttt{NodeM(X))}\\ &= \exists\texttt{Y. Y} \times \texttt{(Y} \rightarrow \texttt{NodeM(NodeType))}\end{aligned}$$

Now look at the type `DbleNodeType` of objects generated by the subclass `DbleNode`.

```
DbleNodeType = OT(DbleNodeM)
             = μX. ∃Y. Y × (Y → DbleNodeM(X))
             = ∃Y. Y × (Y →
             DbleNodeM(DbleNodeType))
```

Is `DbleNodeType` <: `NodeType`? It is easy to show using the subtyping rules for existentials, pairs, and functions that `∃Y. Y × (Y → MS')` <: `∃Y. Y × (Y → MS)` iff `MS'` <: `MS`. Therefore, `DbleNodeType` <: `NodeType` iff `DbleNodeM(DbleNodeType)` <: `NodeM(NodeType)`.

Because `DbleNodeM` <: `NodeM` as functions from types to types, for every field of `NodeM` there is a corresponding field of `DbleNodeM`. The problem is with the subtyping of individual fields (methods). Suppose, for the purposes of finding a contradiction, that `DbleNodeType` <: `NodeType`, and take a look at the type of the method `setNext` in each of the method suites. In the source language, in both classes the type of the method is `MyType` → `void`. In the encoding, this translates to `NodeType` → `void` in the `setNext` field of `NodeM(NodeType)` and `DbleNodeType` → `void` in the corresponding field of `DbleNodeM(DbleNodeType)`.

There is no difficulty with the range types of these function types since they are identical. But we need the domains of the types to be in a contravariant relation if the function types are to be subtypes. Thus we need `NodeType` <: `DbleNodeType`, but we assumed we had the opposite (and they are certainly not the same!). This contradiction shows that `DbleNodeType` is not a subtype of `NodeType`. A more direct proof can be given that essentially emulates the proof we gave that Eiffel is not type safe.

What went wrong here? The difficulty is that if `MyType` occurs in a negative or contravariant position (e.g., as the type of a subscript), then subtyping will fail. This failure is a result of the fact that the meaning of `MyType` implicitly changes when we create a subclass. Occurrences of `MyType` as return types cause no difficulty, only its use as a parameter type in methods.

Thus we have gained a more expressive type system at the cost of losing the guarantee that subclasses always generate subtypes. However, if we restrict the use of `MyType` to only positive positions, then we retain the guarantee that subclasses generate subtypes. The language Trellis/Owl, designed in the mid-1980's imposed exactly this restriction on the use of `MyType`.

7.2 The matching relation

We noted earlier that if `C'` with type `ClassType(IV',M')` is a subclass of `C` with type `ClassType(IV,M)`, then $\mathtt{IV'}^{\mathtt{ref}}$ <: $\mathtt{IV}^{\mathtt{ref}}$ and `M'` <: `M`. The types of objects generated by the classes are `CType'` = `OT(M')` and `CType` = `OT(M)`. As we have just seen, if `OT(M)` includes a negative occurrence of `MyType`, then these will not be subtypes. However, we will find it useful to characterize their relationship by giving it a name.

We say `OT(M')` *matches* `OT(M)` (written `OT(M')` <# `OT(M)`) iff `M'` <: `M`. Thus two object types are in the matching relation if the smaller of the two has at least the methods of the larger, and corresponding method types are in the subtype relation when `MyType` is treated as an unknown type variable.

By contrast, `OT(M')` <# `OT(M)` iff `M'` <: `M` *and* `M` contains no use of `MyType` in a negative position, i.e., it has no occurrence of `MyType` as a parameter type. Thus if two object types are subtypes then they match, but not vice-versa.

By the definition of matching, if `C'` is a subclass of `C`, then their types must be in the matching relation. In particular, `DbleNodeType` <# `NodeType`. Thus, while the types of objects generated by subclasses need not be subtypes, they must be in the matching relation.

Matching provides information about what methods are available and their types. Let

```
Comparable = OT({equals, lessThan, greaterThan: MyType → boolean})
```

Then `Comparable` has no subtypes because of the occurrence of `MyType` as a parameter type. Suppose we know that `obj` has type `T`, where `T <# Comparable`. Then `obj` has methods including `equals`, `lessThan`, and `greaterThan`. We also know that all of these methods take parameters of type `T` (because that is the meaning of `MyType` for objects of type `T`), and return values of type `boolean`.

While matching provides information on what methods an object supports, it does not guarantee that objects of one type can "masquerade" as objects of another type. The stronger notion of subtyping is necessary in order to provide that assurance. Interestingly, much of the time only information on what methods can be sent to an object (and their types) is needed. This is especially the case for bounded polymorphism.

Let

```
class OrderedList[Elt <# Comparable]{
   insert(item:Elt):void {...}
   removeFirst():Elt{...}
   empty():boolean {...}
   search(o:Elt):int {...}
}
```

We give the name *match-bounded polymorphism* to this use of matching to constrain polymorphism.

Inside the methods of `OrderedList`, we are guaranteed that we can send the messages `equals`, `lessThan`, and `greaterThan` to objects of type `Elt`. This information about messages and their types is exactly what we needed to write this class. Moreover, this way of expressing the constraint is more straightforward and easier to understand than using F-bounded polymorphism.

In distinction to the use of F-bounded polymorphism, if the type of objects generated by class `C` matches a constraint, then objects generated by any subclass will also satisfy the same constraint. Let `C` be a class which generates objects of type `CType`, and suppose `CType <# BdType` for some object type, `BdType`. Let `D` be a subclass of `C`.

Then the type of objects generated from `D`, `DType`, matches `CType`. But `DType <# CType` and `CType <# BdType` implies that `DType <# BdType`. Thus if the type of objects generated by a class satisfies a match-bound, then the type of objects generated by any subclass will satisfy the same match-bound.

In summary, by including the type `MyType` we have lost the guarantee that subclasses always generate subtypes, but we have gained considerably in the expressiveness of the language. It is now easier to define subclasses (*e.g.*, the definition of `DbleNode` as a subclass of `Node`), and it is easier to express bounded polymorphism. When `MyType` occurs only in the return types of procedures, we retain all of the advantages of the simpler languages discussed in previous sections, including the fact that subclasses always generate subtypes.

7.3 Type-checking with `MyType`

In the previous subsection we noted that if the method suite of a class type-checked under our encoding, then it was guaranteed to be type-safe in all subclasses. However, our encoding used bounded type quantifiers representing the type interfaces of the instance variable and method suites. In the real world, programmers would not be happy having to translate their classes into that notation in order to understand whether or not their method bodies would type check. In this section we look for a way to explain the type-checking rules to programmers.

To make our work a bit easier, we will separate the uses of `self` for getting access to instance variables and as a target for methods. Thus we replace `self.x` by `selfinst.x`, if `x` is an instance variable. Similarly replace `self.m(...)` by `selfmeth.m(...)`, if `m` is a method. For simplicity, we'll also add `value` qualifiers to variables when we need their value.

For example, here is code for our earlier `Node` class where we've added a method `bump` to increase the contents of the `item` field by one in order to demonstrate how to handle sending a message to `self`.

```
class BNode{
   item: int = o;
   next: MyType = null;  -- Right neighbor

   setItem (newItem: int): void { selfinst.item := newItem;}
   getItem ():int  { return value selfinst.item;}
   setNext (other: MyType): void { selfinst.next := other;}
   getNext (): MyType { return value selfinst.next;}
   bump():void { selfmeth.setItem(selfmeth.getItem()+1);}
}
```

Its interface is

```
interface BNodeType{
   setItem (newItem: int): void;
   getItem ():int;
   setNext (other: MyType): void;
   getNext (): MyType;
   bump():void;
}
```

While this is annotated a bit more than one would normally find in a programming language, it would be easy for the compiler to add these annotations when parsing the class.

Encoding the class results in a term of the form:

$$\Lambda(\texttt{M'} <: \texttt{M}).\ \Lambda(\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}).\ \langle\texttt{inst},\ \lambda(\texttt{self:VOT(IV',M')}).\texttt{meth}\rangle$$

where

$$\texttt{VOT(IV',M') = IV'(OT(M'))} \times \texttt{(IV'(OT(M'))} \rightarrow \texttt{M'(OT(M')))}$$

Recall that occurrences of `OT(M')` in the encoding correspond to our use of `MyType`. We are given that $\texttt{M'} <: \texttt{M}$, so by the definition of matching, $\texttt{OT(M')} <\# \texttt{OT(M)}$. Thus we may assume that $\texttt{MyType} <\# \texttt{OT(M)}$ in type-checking `inst` and `meth`. In the case of type-checking the class `BNode` we may assume $\texttt{MyType} <\# \texttt{BNodeType}$

We use the fact that the type of `self` in the encoding is `VOT(IV',M')` to determine how to type-check the class. From the definition of `VOT(IV',M')`, we see that the type of `selfinst` is the first component of the meaning of the "visible" type, `IV'(OT(M'))`. Because $\texttt{IV'} <: \texttt{IV}^{\texttt{ref}}$, and no depth subtyping is possible for reference types, we are guaranteed that if `x` is an instance variable declared in $\texttt{IV}^{\texttt{ref}}$, then its type will be the same in `IV'`. Thus in type-checking `BNode`, we can assume that `selfinst`.item has type int and `selfinst`.next has type `MyType`.

For message sends to `selfmeth`, we will actually be a bit sneaky. Rather than interpreting `selfmeth` as the method suite of `self` in type $\texttt{VOT(IV'}^{\texttt{ref}}\texttt{,M')}$, we instead interpret it as `close[IV'`$^{\texttt{ref}}$`,M'](self)`, which has type `(OT(M'))` $=$ `MyType`. We can get away with this slight of hand because the results of sending a message to `close[IV'`$^{\texttt{ref}}$`,M'](self)` is exactly the same as sending it to `self`.

Exercise 7.1 *Check the encoding yourself, remembering the two definitions of message send, which depend on whether the type of the receiver was a "visible" object type or a "packed" object type.*

While it seems a bit strange to pack something up with `close` and then immediately open it up again when sending the message, the advantage is that we can now write down the type of `selfmeth` as `MyType`, something which makes sense to the programmer. (After all, even with this strange encoding, an implementation could optimize and just execute the simpler encoding.)

Thus we will type check the methods of `BNode` under the assumption that `selfmeth` has type `MyType` and that `MyType`matches `BNodeType.`

We can write the formal type-checking rule for classes as follows:

$$\textit{Class} \qquad \frac{\begin{array}{c} C \cup \{\texttt{MyType} \lessdot\!\# \, \texttt{OT}(M)\}, E \vdash \texttt{inst: IV(MyType)} \\ C \cup \{\texttt{MyType} \lessdot\!\# \, \texttt{OT}(M), \texttt{IT} <: \texttt{IV}^{\texttt{ref}}\texttt{(MyType)}\}, \\ E \cup \{\texttt{selfinst: IT}, \texttt{selfmeth: MyType}\} \vdash \texttt{meth: M(MyType)} \end{array}}{C, E \vdash \texttt{class(inst,meth): ClassType}(IV, M)}$$

$$\textit{New} \qquad \frac{C, E \vdash \texttt{C: ClassType}(IV, M)}{C, E \vdash \texttt{new C: OT}(M)}$$

Those rules are a bit hard to read because of the formalism, but concisely summarize the type-checking rules discussed in this subsection.

Finally we present the type checking rule for message sends:

$$\textit{Msg} \qquad \frac{C \vdash \texttt{U} \lessdot\!\# \, \texttt{OT}(\{\texttt{m: T}\}) \qquad C, E \vdash \texttt{obj: U}}{C, E \vdash \texttt{obj.m: T}[\texttt{MyType} \mapsto \texttt{U}]}$$

Put more informally, this rule states that if `obj` has type `U`, where `U` $\lessdot\!\#$ `OT({m: T})`, then the type of `obj.m` is the type `T` of method `m`, where all free occurrences of `MyType` in `T` have been replaced by the type `U` of the receiver. In other words, when you send a message to an object, all occurrences of `MyType` in the method type are replaced by the static type of the receiver. This matches the intuition that `MyType` represents the type of the object executing the method.

Notice that while these type-checking rules mention `OT(M)`, they do not require an understanding of the encoding in order to do the type-checking. We have already discussed the assumptions used in type-checking the class `BNode`. Suppose `bn` is an object with static type `BNodeType`. Then `bn.getNext()` returns a value of type `BNodeType`, while `bn.setNext(newNext)` is well-typed only if `newNext` has type `BNodeType`.

Further, if `CNodeType` $\lessdot\!\#$ `BNodeType`, and `cn` has static type `CNodeType`, then `cn.getNext()` and `cs.setNext(newNext)` respectively return a value of type `CNodeType` and take a parameter of type `CNodeType`.

Exercise 7.2 *Fill in the missing instance variables and code for class* `OrderedList` *in section* **??**, *and show that each of the initial values for instance variables and method bodies is well-typed.*

In this section we showed how to add the keyword `MyType`, representing the type of `self`, to an object-oriented language in order to gain expressiveness and yet retain type-safety. We also added the matching relation to the language, and discovered that match-bounded polymorphism is more useful than plain bounded polymorphism, and is more compatible with the subclass relation than F-bounded polymorphism.

The price paid for these advantages is that subclasses will not always generate subtypes, though they do generate matching types, and the language needs to support both matching and subtyping, two quite similar relations. The research language PolyTOIL [**?**] is very similar to the language described in this section.

In [**?**], we went a step further and designed a language with `MyType`, matching, and match-bounded polymorphism, but without subtyping. In order to recapture some of the benefits of subtyping, the language introduces "hash types", written `#T`. If an expression has type `T` then at run time, the value of the expression will have exactly the type `T`. However, if it has type `#T`, then at run time its value may be of any type which matches `T`.[3] Thus for each variable, parameter, or return type whether we wish it to have an exact or a hash type.

The main disadvantage of using hash types is that if an expression has a hash type then we may not send a binary message to it. For example, suppose that `nd` has type `#NodeType`. It is sensible that `nd.getNext()` returns a value of type `#NodeType`. However, what should the type of `newNext` be in `nd.setNext(newNext)`, if it is to be well-typed? If the value of `nd` at run time is `NodeType`, then the parameter should also be of type `NodeType`, while if the value of `nd` at run time is `DbleNodeType`, then the parameter should be of type `DbleNodeType` (and a type error will occur if the parameter instead has type `NodeType`). Because binary methods require the parameter to have the same type as the receiver, we cannot determine what the correct type of the parameter should be if we do not know the exact type of the receiver.

As discussed in [**?**], this restriction is not a problem in practice because the use of `MyType` does not make sense from a modelling point of view when working with heterogeneous data structures, and these heterogeneous data structures are the main reasons for the use of hash types.

While initial experience with LOOM has been very positive, greater use of the language will be necessary in order to evaluate its success in balancing type safety, expressiveness, and simplicity.

References

[BFP97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP '97*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *ECOOP '95*, pages 27–51. LNCS 952, Springer-Verlag, 1995.

[3] `#T` can be understood as representing the bounded existential type, $\exists \mathtt{t} <\!\# \, \mathtt{T.\ t}$.