Understanding Object-Oriented Languages: Semantics and Types *(continued)*

Kim B. Bruce
Princeton University & Williams College

December 1, 1998

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.

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) → 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)
```

[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.)

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:

$$\texttt{o.m(params) = open o as } \langle \texttt{IV, vo} \rangle \texttt{ in (snd(vo) fst(vo)).m(params)}$$

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

$$\texttt{CellType = OT(CellM) = } \exists \texttt{Y. Y} \times \texttt{(Y} \rightarrow \texttt{CellM).}$$

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)`.

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 intial 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 $\texttt{IV}^{\texttt{ref}}$ as follows. If $\texttt{IV} = \{\texttt{l}_1{:}\,\tau_1;\ldots;\texttt{l}_n{:}\,\tau_n\}$, let $\texttt{IV}^{\texttt{ref}} = \{\texttt{l}_1{:}\texttt{Ref}\tau_1;\ldots;\texttt{l}_n{:}\texttt{Ref}\tau_n\}$. Similarly, if $\texttt{inst} = \{\texttt{l}_1 = \texttt{i}_1;\ldots;\texttt{l}_n = \texttt{i}_n\}$, let $\texttt{inst}^{\texttt{ref}} = \{\texttt{l}_1 = \texttt{ref i}_1;\ldots;\texttt{l}_n = \texttt{ref i}_n\}$.

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

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

The type of a class is of the form:

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

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

Notice the initial values of the instance variables have type `IV`, while the visual object type involves type $\texttt{IV}^{\texttt{ref}}$ 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 $\texttt{IV}^{\texttt{ref}}$) as a parameter, while the method suite for classes takes an entire object of type $\texttt{VOT(IV}^{\texttt{ref}}\texttt{,M)}$ 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 $\texttt{VOT(IV}^{\texttt{ref}}\texttt{,M)} \to \texttt{M}$, we create `meth'` of type $\texttt{IV}^{\texttt{ref}} \to \texttt{M}$ more appropriate for an object as follows:

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

More formally,

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

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

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

for $\texttt{inst}^{\texttt{ref}}$ 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

```
inst = {x = 0}: CellIV
```

and

```
meth = λ(self:VOT(CellIVref,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.

```
instref = {x = ref 0}: CellIVref
```

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

```
meth' = λ(ir:CellIVref).
           {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 `close(IV`$^{\texttt{ref}}$`,M)`$\langle$`inst`$^{\texttt{ref}}$`,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 $\langle$`inst`$^{\texttt{ref}}$`,meth'`$\rangle$ is of type `VOT(CellIV`$^{\texttt{ref}}$`,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)` whose instance variable has value 47. Recall that you must unpack the existential before sending the message. Also recall that `:=` denotes an 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.