CS411 Objects I

A. Demers

13 May 2001

In these Notes we discuss class-based object-oriented languages. This is roughly the material in Chapters 2-3 of Abadi and Cardelli, and is presented at a similar level of informality. We concentrate on class-based languages. Issues about object-based languages (the subject of Abadi and Cardelli Chapter 4 and the subject of Riccardo's lecture of April 26) are mentioned only briefly.

1 Introduction to Classes and Objects

Here is the first example of an Object-Oriented (OO) program from Abadi and Cardelli, translated into a syntax close to the one we've been using:

```
class cell {
        v : var int;
        get : method()int{ self.v };
        set : method(x : int)int{ self.v ← x }
} in ...
```

Intuitively this defines a class of objects each of which has an integer component v and functional components *get* and *set* which respectively interrogate and update the value of v. Conventionally, value components like v are called "fields," and functional components are called "methods."

The syntax for using fields and methods is similar to that for using components of product types, so the above program fragment could continue

```
... {
  let c ~ newobj(cell) in
      c.a ← 11;
      c.set(c.get + 6);
      c.get();
}
```

Within the body of a method, the special name **self** refers to the "current" object instance – the one from which the current method invocation was selected. Semantically, the usual way to think of **self** is as an implicit hidden parameter, which is bound at method invocation time like any other parameter. This interpretation has serious implications for type checking, as we'll see below.

Our general syntax for class definitions is the following:

$$
\begin{array}{l}
\textbf{class}\ C\ \{ \\
\quad \ldots \\
\quad f_i : \textbf{var}\ \tau_i, \\
\quad \ldots \\
\quad m_j : \textbf{method}(x_j : \tau'_j)\tau''_j\{e_j\} \\
\quad \ldots \\
\}\ \textbf{in}\ e
\end{array}
$$

Intuitively, a class definition describes a set of objects, so classes are related to types.

At a very high level, a type defines a set of values and some operations that you know how to apply *a priori* to those values.

With objects the responsibility is divided differently. Each object contains its operations (or methods) in itself. The object conforms to a *signature*, which describes what operations are to be expected and (recursively) what their types are.

You can think of an object as a record (an element of a product type) with components of two kinds: *fields* are **var** components of some type (possibly itself an object type); *methods* are procedure or function valued components that have implit access to the object instance that contains them.

To support classes and objects we make a number of extensions to our expression language. The first is objecct constructions themselves:

$$e \quad ::= \quad \textbf{newobj}(C)$$

Here C is a class name, not a type. Each class definition C implicitly determines a type for the objects that are "of class C." We'll call this type the *instance type* of C, and denote it by $IT(C)$. We'll discuss possible choices for $IT(\cdot)$ below.

In a formal set of type rules we would expect

$$\pi \ \vdash\ \textbf{newobj}(C) : \ IT(C)$$

where π is some generalization of a type assignment that reflects the class declaration for C.

The reserved word **self** is an expression:

$$e \quad ::= \quad \textbf{self}$$

In the body of a method m in class C, the name **self** refers to the current instance – the object from which m was selected and invoked. Thus, we expect

$$\pi \vdash \textbf{self} : IT(C)$$

where π is the type assignment used to type the body of class definition C.

A field can be selected from an object:

$$e \quad ::= \quad e.f_i$$

This has the same interpretation as selection from a value of a product type. For its typing rule, we expect

$$\frac{\pi \vdash e : IT(C)}{\pi \vdash e.f_i : \tau_i}$$

Method invocation involves a combination of selection and application:

$$e \quad ::= \quad e.m_j(e')$$

This invokes selects method m_j from object e, then invokes it passing the argument e'. For the typing rule, we expect

$$\frac{\pi \vdash e : IT(C) \qquad \pi \vdash e' : \tau_j'}{\pi \vdash e.m_j : \tau_j''}$$

Note that a method *selection* like $e.m_j$ is not by itself a well formed (or well typed) expression.

2 Embedding in IMPX

Here is a slightly more formal treatment of the description given above. We show how the object mechanism described thus far can be translated to IMPX in a type-correct way.

The important trick is to exploit the intuition that **self** behaves like an implicit parameter to each method. We translate each method in a class definition to a *pre-method*, in which the **self** parameter is made explicit. Of course, this will result in the methods described by the type $IT(C)$ having parameters of type $IT(C)$ – that is, we'll be using recursive types in an essential way.

We introduce the keyword **Self** to represent the type of **self** inside a class definition. Then we define the "representation type" induced by a class definition C to be

$$\begin{aligned} RT(C) \;=\; & \mathbf{prod}(\;\ldots,\; f_i : \mathbf{var}\;\tau_i,\;\ldots, \\ & \ldots,\; m_j : \mathbf{fun}(\mathbf{self} : \mathbf{Self},\; x_j : \tau'_j)\tau''_j,\;\ldots\;) \end{aligned}$$

This is just the natural record type obtained by replacing each method by its corresponding premethod and using **Self** for the type of **self**. Now a sensible interpretation for the instance type of C is obtained by introducing recursion on **Self**:

$$IT(C) \;\equiv\; \mu\;\mathbf{Self}.RT(C)$$

A translation $T[\![\cdot]\!]$ can be defined as usual by induction on expressions in the language extended with classes. There are only a few interesting cases: object construction, field selection and method invocation. The rule for object construction is:

$$\begin{aligned} T[\![\mathbf{newobj}(C)]\!] \;=\; & \mathbf{abs}(\;\langle \\ & \ldots,\; f_i \;\sim\; \mathbf{newvar}(\tau_i),\;\ldots \\ & \ldots, m_j \;\sim\; \mathbf{lambda}(\mathbf{self} : \mathbf{Self},\; x_j : \tau'_j).(T[\![e_j]\!]),\;\ldots\;\rangle\;) \end{aligned}$$

An object construction is translated to a record construction in which methods are replaced by the corresponding premethods; the record construction is wrapped in an "abs" constructor. This leads to the typing

$$\pi \;\vdash\; T[\![\mathbf{newobj}(C)]\!] : \;\mu\;\mathbf{Self}.RT(C) \;=\; IT(C)$$

Field selection simply requires that the recursive type be unwound properly. Thus, if e is an object expression of class C,

$$T[\![e.f_i]\!] \;=\; (\mathbf{rep}(e)).f_i$$

which leads to the typing

$$\begin{array}{rl} \pi \vdash & T[\![e.f_i]\!] : [(\mu\ \mathbf{Self}.RT(C))/\mathbf{Self}](RT(C)) \\ = & [IT(()C)/\mathbf{Self}](RT(C)) \end{array}$$

Finally, the translation of method invocation must explicitly pass the **self** parameter as well as unwinding the recursive type:

$$T[\![e.m_j(e')]\!] = \mathbf{let}\ z \sim e\ \mathbf{in}\ (\mathbf{rep}(z)).m_j(z,\ T[\![e']\!])$$

This leads to the typing

$$\begin{array}{rl} \pi \vdash & T[\![e.m_j(e')]\!] : [(\mu\ \mathbf{Self}.RT(C))/\mathbf{Self}](\tau_j'') \\ = & [IT(C)/\mathbf{Self}](\tau_j'') \end{array}$$

This is just the method result type with the recursion unwound once.

Example: Recursive types in IMPX are a bit counterintuitive, so here is a short example using the above rules. We use the fairly meaningless class definition

$$\begin{array}{l} \mathbf{class}\ C\ \{ \\ \quad get : \mathbf{method}()\mathrm{int}\{\ 0\ \}; \\ \quad copy : \mathbf{method}()\mathbf{Self}\{\ \mathbf{self}\ \} \\ \}\ \mathbf{in}\ \ldots \end{array}$$

The representation type is

$$\begin{array}{rl} RT(C) = & \mathbf{prod}(\\ & \quad get : \mathbf{fun}(\mathbf{self} : \mathbf{Self})\mathrm{int}, \\ & \quad copy : \mathbf{fun}(\mathbf{self} : \mathbf{Self})\mathbf{Self}\ ) \end{array}$$

and the corresponding instance type is

$$\begin{array}{rl} IT(C) = & \mu\mathbf{Self}\ .\ RT(C) \\ = & \mu\mathbf{Self}\ .\ (\ \mathbf{prod}(\\ & \quad get : \mathbf{fun}(\mathbf{self} : \mathbf{Self})\mathrm{int}, \\ & \quad copy : \mathbf{fun}(\mathbf{self} : \mathbf{Self})\mathbf{Self}\ )\ ) \end{array}$$

Now consider the method invocation

$$(\mathbf{newobj}(C)).get()$$

The invocation translates as

$$T[\![(\mathbf{newobj}(C)).get()]\!] = \\ \quad \mathbf{let}\ z\ \sim\ \mathbf{newobj}(C)\ \mathbf{in}\ (\mathbf{rep}(z)).get(z))$$

Here z has type $IT(C)$, which is an abstract recursive type. By the rules for recursive types,

$$\mathbf{rep}(z)\ :\ [IT(C)/\mathbf{Self}](RT(C))$$

and the selected method *get* has type

$$(\mathbf{rep}(z)).get\ :\ [IT(C)/\mathbf{Self}](\mathbf{fun}(\mathbf{self}\ :\ \mathbf{Self})\mathrm{int}) \\ \quad =\ \mathbf{fun}(\mathbf{self}\ :\ IT(C))\mathrm{int}$$

Now the type of the argument z matches the parameter type in the invocation

$$\ldots\ z.get(z)$$

and the method invocation is properly typed with result int as expected.

A similar argument shows that in the same context a selected *copy* method would have type

$$(\mathbf{rep}(z)).copy\ :\ [IT(C)/\mathbf{Self}](\mathbf{fun}(\mathbf{self}\ :\ \mathbf{Self})\mathbf{Self}) \\ \quad =\ \mathbf{fun}(\mathbf{self}\ :\ IT(C))IT(C)$$

Thus the result type of an invocation of *copy* is $IT(C)$ as it should be.

3 Method Dictionaries

In the representation described above, we conceptually store a copy of every method in every object instance. This representation allows maximum flexibility – in principle, it would be possible to replace individual methods at a per-object granularity. On the other hand, if premethods are used, then clearly the premethod values are identical in every object instance of a given class; so arguably this representation is rather wasteful of space.

It is quite possible to share the premethods – that is, store only a single copy of the premethods of a class, and have all object instances of the class refer to these methods indirectly. One way to do this, used in some Java systems, replaces the representation type $RT(C)$ by three types: a *field* type $FT(C)$,

a *method dictionary* type $DT(C)$, and a *handle* type, $HT(C)$. The field and dictionary types hold, respectively, the fields and the methods of an instance. A handle is the "glue" that holds the fields and methods together: it is a pair of references to the fields and methods. It is simple to arrange that all instances of any particular class can share the same copy of the method dictionary.

Doing this in detail would be somewhat tricky in IMPX, however, because the three types $FT(C)$, $DT(C)$ and $HT(C)$ must be mutually recursive, and this is not well supported in IMPX.

4 Subclasses

If OO programming were restricted to the features we have presented thus far, it might be an amusing programming style, but it would arguably contain nothing of much interest or importance. The power of OO programming begins to appear when we allow *subclassing*. To illustrate, here is an extension of the "cell" example from above:

```
subclass reCell of cell {
        b : var int;
        set : override(x : int)int{ self.v ← self.x; self.v ← x }

qquad restore : method()int{ self.v ← self.b };
} in ...
```

Because reCell is declared to be a subclass of cell, it *inherits* all the fields and methods of cell. It may add new fields (like "b"), add new methods (like "restore"), or override methods (like "set") by providing a new method definition.

Intuitively, what this program does should be pretty clear to all you Java programmers out there. It defines a class named "reCell" (for "restorable cell") that is a *subclass* of cell in the sense that an object of the reCell class behaves like an object of the cell class, provided the client restricts itself to fields and methods defined in the cell class. Clients that know about the reCell class can invoke the "restore" method of a reCell object to undo the effect of the most recent set operation.

The ability to make incremental modifications, to add new behaviors without breaking existing code, and to maximize code reuse through inheritance, is the major strength of OO programming. Any attempt to model it formally *must* provide a satisfactory account of subclassing and inheritance. It's harder than it looks ...

The notion of "usable in place of" is the same one we used earlier to motivate the subtyping relation in IMPX. It suggests that the subclass relationship and the subtype relationship should correspond; that is

$$C \prec C' \qquad \Leftrightarrow \qquad IT(C) \prec IT(C')$$

This property, often called "subclassing *is* subtyping" for obvious reasons, is characteristic of many OO languages, especially older designs.

Observe that our instance types are (and for the remainder of these notes will continue to be) entirely structural. In particular, nowhere in $IT(C)$ does the class name of C actually appear. Instead, we get the generic identifier **Self**. Thus, it is perfectly possible for two classes, unrelated by the subclass relation, to generate identical instance types. Consequently, the implication above fails in one direction for our system; we have a "subclassing *implies* subtyping"

At first glance it would seem that subclassing is essentially the same as subtyping of the record types generated by our premethod-based translation above. Both techniques add new fields or methods to move from a superclass (or supertype) to a subclass (or subtype). Unfortunately, the use of **Self** in a contravariant position (that is, in the parameter position of the types of premethods) makes the typing fail in the presence of subtypes, as we'll see below.

4.1 Premethod Typings Fail

We consider the types of the *set* premethods In the simple cell $\rightarrow$ reCell example. We'll be slightly informal, and use the names "$IT(cell)$" and "$IT(reCell)$" directly rather than using recursive types. Then the type of the *get* premethod of a cell object should be

$$\mathbf{fun}(\mathbf{self} : IT(cell))\ \text{int}$$

while the type of the *get* premethod of a reCell object should be

$$\mathbf{fun}(\mathbf{self} : IT(reCell))\ \text{int}$$

Similarly, the type of the *set* premethod of a cell object should be

$$\mathbf{fun}(\mathbf{self} : IT(cell), x : \text{int})\ \text{int}$$

while the type of the *set* premethod of a reCell object should be

$$\mathbf{fun}(\mathbf{self} : IT(reCell), x : \text{int})\ \text{int}$$

In both cases, these differ by the type of the **self** parameter in the obvious way.

There are several ways in which methods of two classes can be "mixed:"

Inheritance: When we construct an object value of class reCell, it inherits the value of *get* from the value defined for the superclass cell. As we saw above, the types of the premethods cell.*get* and reCell.*get* are not the same. But, with the assumption that $IT(reCell) \prec IT(cell)$, all is well. Since the uses of $IT(\cdot)$ occur in the (contravariant) parameter position in the premethod types, we get

$$\begin{array}{l} IT(reCell) \prec IT(cell) \quad \Leftrightarrow \\ \quad \textbf{fun}(IT(reCell))\text{int} \prec \textbf{fun}(IT(cell))\text{int} \end{array}$$

and the inheritance is legal under the typing rules.

Overriding: Consider binding an object of class reCell (say "someRefCell") to a variable of class cell (say "someCell"), for example by parameter passing. The premethod typeing rules can allow this only if the type of someRefCell.*set* is a subtype of someCell.*set*. Unfortunately, what we can prove is:

$$\begin{array}{l} IT(reCell) \prec IT(cell) \quad \Leftrightarrow \\ \quad \textbf{fun}(IT(reCell), \text{int})\text{int} \prec \textbf{fun}(IT(cell), \text{ int})\text{int} \end{array}$$

which is exactly the reverse of what we need.

Thus, using standard functional typing rules on premethods is too restrictive to allow method overriding, and thus is not satisfactory for describing an object system.

This raises an interesting question: are the rules simply too restrictive to express method overriding, or is method overriding itself unsound in certain circumstances? The answer is "both." We'll argue that the contravariant use of **Self** as the implicit parameter of a premethod could be allowed, indeed must be allowed to enable any method overloading at all. On the other hand, the use of **Self** in any *other* contravariant position is inherently unsound.

4.2 Eliminating Implicit self Parameters

Intuitively, the implicit **self** parameter of a premethod does not lead to unsound typings because premethods can be used only in restricted ways. In particular, it is not possible to pass an arbitrary object as the **self** parameter, or even an arbitrary object of the **Self** type. The *only* object that be passed to the **self** parameter of a premethod is the identical object from which the premethod was selected.

This intuition suggests that we can get a sound type system if we can guarantee, first, that a premethod is always consistent with the particular object of which it

is a part, and second, that a premethod can never be selected from its containing object without binding at least the **self** parameter.

For IMPX this goal is not too difficult to achieve. We exploit a trick we've seen before of using an assignable **prod** variable to achieve the effect of simultaneous recursive definitions. Specifically, we modify $RT(\cdot)$, $IT(\cdot)$ and $T[\![\cdot]\!]$.

The change to $RT(C)$ is simple:

$$\begin{array}{l} RT(C) \;=\; \mathbf{prod}(\;\ldots,\; f_i : \mathbf{var}\;\tau_i,\;\ldots, \\ \qquad \ldots,\; m_j : \mathbf{fun}(x_j : \tau'_j)\tau''_j,\;\ldots\;) \end{array}$$

This is just our previous definition for $RT(C)$ with the **self** parameter eliminated from each premethod type (i.e., with each premethod type changed back to a method type).

As before, the instance type of C is obtained by recursion on **Self**:

$$IT(C) \;\equiv\; \mu\;\mathbf{Self}.RT(C)$$

Note we could now obtain a useful language even without this recursive definition, since the essential use of **Self** – the **self** parameter of each premethod – has been eliminated.

The important changes to the definition of $T[\![\cdot]\!]$ are in the cases for object construction and method application.

The rule for method application is the simplest – we no longer need to pass an explicit **self** parameter, so in effect the rule goes away:

$$T[\![e.m_j(e')]\!] \;=\; (\mathbf{rep}(T[\![e]\!])).m_j(\;T[\![e']\!])$$

This leads to the typing

$$\begin{array}{l} \pi \;\vdash\; T[\![e.m_j(e')]\!] : \;[(\mu\;\mathbf{Self}.RT(C))/\mathbf{Self}](\tau''_j) \\ \qquad =\; [IT(C)/\mathbf{Self}](\tau''_j) \end{array}$$

Note the result type of the application is the same in the original (premethod based) rules and the new ones; the difference is just the absence of the implicit **self** parameter in the new rules.

The rule for object construction changes significantly.

$$\begin{array}{l} T[\![\mathbf{newobj}(C)]\!] \;=\; \mathbf{let}\;z \;\sim\; \mathbf{newvar}(IT(C))\;\mathbf{in} \\ \quad z \;\leftarrow\; \mathbf{abs}(\;\langle \\ \qquad \ldots,\; f_i \;\sim\; \mathbf{newvar}(\tau_i),\;\ldots \\ \qquad \ldots, m_j \;\sim\; \mathbf{lambda}(x_j : \tau'_j).(T[\![([z\uparrow/\mathbf{self}](e_j))]\!]),\;\ldots\;\rangle\;) \end{array}$$

Instead of using premethods with **self** parameters, this translation explicitly binds **self** to the newly constructed object using assignment to a variable in the store. The typing remains

$$\pi \vdash T[\![\mathbf{newobj}(C)]\!] : \mu\, \mathbf{Self}.RT(C) = IT(C)$$

with the new interpretation of $RT(\cdot)$.

Note the run time organization suggested by this discussion. Each object is represented by a tuple including separate copies of each method of its class. Each method is specialized (by a free variable binding) to the particular object that contains it. This organization wastes space, and is not well suited to shared method dictionaries as discussed above. It is not proposed as a real implementation. It is more in the nature of a proof. You should convince yourself that the typings produced for this embedding are identical to those that would be produced by straightforward rules that dealt entirely with method types instead of premethod types. This fact, combined with semantic correctness of the embedding (which is, I hope, pretty clear) and soundness of the IMPX type rules (which we have discussed at some length) shows that a set of typing rules that ignores the **self** parameter of premethods altogether is sound. Thus, the contravariant use of **Self** in the **self** parameter of a premethod is not of concern.

4.3 Other Contravariant Uses of Self

Contravariant uses of the **Self** type other than in the first (**self** parameter) position cannot be "hidden" inside the translation of the object constructor. We argue that they cannot be handled by this style of typing rule. Consider the following simple class definition:

```
class maxClass {
    a : var int;
    max : method(y : Self)Self{
        if self.a > y.a then self else y }
}
```

An object of class "maxClass" has a value field a and a method max. The max method takes, in addition to **self**, an parameter y of the **Self** type. It returns whichever of these two objects (**self** or y) has the greater value in its a field.

Next consider the subclass $minMaxClass$ defined by adding a min method to maxClass:

```
subclass minMaxClass of maxClass {
    min : method(y : Self)Self{
        if self.a < y.a then self else y }
```

This seems like a perfectly natural extension to maxClass, and so far nothing catastrophic has happened.

However, let's consider extending maxClass by overriding as well as adding a method:

$$\begin{array}{l}\textbf{subclass}\ minMaxClass2\ \textbf{of}\ minMaxClass\ \{\\ \quad max:\ \textbf{override}(y:\textbf{Self})\textbf{Self}\{\ y.min(\textbf{self})\}\end{array}$$

Here the max method is redefined in terms of the min method with the operands reversed.

A "sublassing is subtyping" or "subclassing implies subtyping" rule in this case is unsound. Consider

$$\begin{array}{l}\textbf{let}\ a:IT(maxClass)\ \sim\ \textbf{newobj}(minMaxClass2)\ \textbf{in}\\ \quad \textbf{let}\ b:IT(()maxClass)\ sim\ \textbf{newobj}(maxClass)\ \textbf{in}\\ \quad\quad a.max(b)\end{array}$$

If the typing rules imply

$$IT(minMaxClass2) \prec IT(minMaxClass) \prec IT(maxClass)$$

then the binding of a to a $minMaxClass2$ value must be allowed. The application $a.max(b)$ will use the overridden definition of max. This will attempt to invoke the min method of argument b, which is a maxClass object and thus has no min method to invoke.

5 Summary

There's a great deal we haven't touched here. In particular, we haven't discussed multiple inheritance at all, and we certainly haven't said enough about recursive object types. But we have come to a reasonable position to stop: we have a sound single-inheritance "subclassing implies subtyping" system, with **Self** types allowed in any covariant position. This is (nearly) enough to write real programs.

I only wish we'd had time to cover Interfaces ...