Scribe: André Allavena & Alex Slivkins Lecturer: Andrew Myers

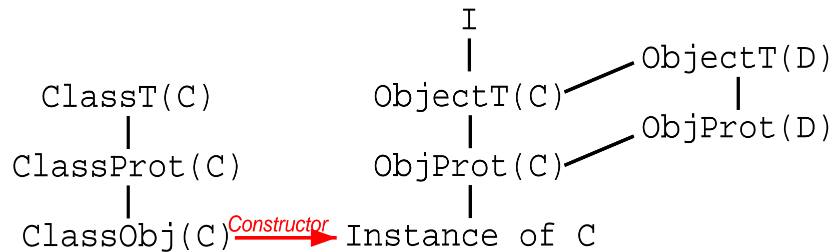
Last time we had an introduction to object oriented languages. A book by Abadi & Cardelli gives more information; first six chapters are nice and easy.

In types we've seen before, one type definition gave rise to one type. However, class definition generates several types and values.

For example, if we assume all methods public and all fields protected, and write

class C extends D implements I

we get the following type hierarchy.



ObjProtT(C) represents the object seen from the *inside* (with all the protected fields) whereas ObjectT(C) only contains the types of the public fields.

There is a cycle relationship on the above scheme. This is because, like in Java, when extending a class D with C, we also create subtype relation between ObjProtT(C) and ObjProtT(D): ObjProtT(C)$\leq$ObjProtT(D).

Can we separate sub-typing and inheritance? This would allow more code reuse since you wouldn't need to worry about your code being used by functions expecting the superclass.

1 Pure inheritance relationship

This is achieved with the C++ private inheritance mode, when only subtypes know they are subtypes; in Modula-3 the subtype relations are encapsulated in modules.

Conformance

When extending D with C, the types have to agree, in a certain way, in order to have ObjProtT(C)$\leq$ObjProtT(D). This agreement is called *conformance.* How much conformance is required when inheriting without subtyping? Mutable fields have to be identical because they are references. Methods are typically functions but when called on the object we have some trouble with the covariance of arguments. So let us introduce a new type *Self* representing the subclass when inherited (in other words the current subclass). Self goes down the hierarchy when inheriting. (*self*: *Self* is like `this` in Java).

A value of type C will not be used at type D, we can relax checking, and covariance is now OK; we can write something like:

```
class D { boolean equals(Self x) }
class C inherits D { boolean equals(Self x) }
```

2 Object Types

The question is to define what is an object.

A first approximation is to consider an object as a recursive record, allowed to use *Self* inside its declaration.

```
class Point {
      int x, y;
      Point movex(int d) {...}
}
```

ObjectT(Point) {
$\mu P.\{x\text{: int}, y\text{: int}, movex\text{: int} \to P\ \}$

This gives satisfactory account of field, method selection and object construction (without inheritance):

```
new_point(xx,yy) = rec self {x=xx, y=yy,
            movex = λd: int. new_point(self.x + d, self.y) }
```

We can find fixed point in CBV language if the object is only in scope of function-typed expressions (methods). (See Homework 4.)

Regarding inheritance, let us consider the following subclass:

```
Class colored_point extends point
    { Color c;
      colored_point(int x, int y, Color cc)
           { point(x,y); c=cc }
      move_x(int dx)
         { return new colored_point(x+dx, y, c); }}
```

Assume we have a record extension operator $e + \{\ldots l_i = e_i \ldots\}$:

$$\{a = 0\} + \{b = 1\} = \{a = 0, b = 1\}$$
$$\{a = 0\} + \{a = 1\} = \{a = 1\}$$

In case of a conflict, RHS wins. In other words, if LHS, RHS contain $l = e$, $l = e'$, resp., the resulting expression contains the latter. The type of e' must be a subtype of the type of e.

```
new_point(xx,yy) = rec self{x = xx, y = yy,
           movex = λd: int. new_point(self.x + d, self.y) }
new_colored_point(xx,yy,cc) = new_point(xx,yy) + { c = cc, movex = ?  }
```

We would like to extend records and do that as a function, and define subtypes. Nice idea, but it doesn't work. As shown in the previous example, we cannot use `point`'s constructor to gobble up a new field. We aren't taking any fixed point with the rec operator. Furthermore self of `color_point` is going to use the `point` version. It won't be linked to the correct object. We need to open up, rebind the recursion of *self* reference in superclass.

3 Constructor Implementation

For simplicity, assume there are only two classes in class hierarchy, $C \leq D$, and that all methods are virtual. Consider a Java-like constructor,

$$\texttt{Constructor}\ C(x_C : \tau_C) = D(e_D); \ldots\ l_j = e_j \ldots$$

Let's see what it does. When it creates an object, methods are initialized immediately, but fields are left uninitialized for a while. The constructor C_{con} calls the superclass constructor D_{con}, which initializes the fields inherited from D. Then the body of C_{con} executes, initializing new fields and possibly changing some of the fields inherited from D.

There is a danger, though: it might be possible to access uninitialized fields. Suppose D_{con} calls a method m_D of D, which doesn't try to access any yet uninitialized fields. Then suppose m_D is overriden by a method m_C in C with the same name. So when C_{con} calls D_{con}, D_{con} actually calls m_C. But how does m_C know which fields it is not supposed to access? Therefore, in order to write methods for C, it does not suffice to know the *signature* of D_{con}; details of its implementation are required. This is bad for OO-language, since

it defeats the point of encapsulation. This is why, for example, in Java virtual functions are not allowed in constructors.

How do we model constructors? Say, we have the following code:

$\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{implements}\ I\ \{$
$\quad\quad\texttt{constructor}\ C(x_c : \tau_c) = D(e_D); \ldots\ l_j = e_j\ \ldots$
$\quad\quad\texttt{public methods}\ \ldots\ m_i = \lambda x_i : \tau_i . e_i\ \ldots$
$\quad\quad\texttt{protected fields}\ \ldots\ l_j : \tau_j \ldots$
$\}$

One option is, the constructor receives a reference to the final result (*self*), and a partially constructed object o to build on:

$$C_{con} : ObjProtT(C) * ObjProtT(C) * \tau_C \to ObjProtT(C)$$

Why need *self*? To close the recursion. In other words, we'll take a fixed point on it.

$$\begin{array}{rcl} C_{con} : (\ldots) & = & \lambda(self, o, x_c).\ D_{con}(self, o, e_D)\ +\ \{\ldots\ l_j = e_j\ \ldots\} \\ \texttt{new}\ C(e_c) & \Rightarrow & \texttt{rec}\ self.\ C_{con}(self,\ \{\ldots\ m_i = \lambda x_i : \tau_i . e_i\ \ldots\},\ e_C) \end{array}$$

However, *self* gets used outside methods. So we need some fancy notion of a pixed point here. This is possible but hard, so we won't go into it here.

Object Calculus

Another option is to use a more powerful construct than recursive records: *object calculus* (see Abadi&Cardelli, ch.7-8). We introduce a special *object* type:

$$\tau ::= \ldots \mid [\ l_i : \tau_i\ ^{i \in 1 \ldots n}]$$

and a new primitive for object creation:

$$o ::= [\ x_1.l_1 = e_1,\ \ldots\ , x_n.l_n = e_n\ ]$$

The idea is that x_i is in scope only in e_i, where it stands for o. This mechanism for (implicit) recursion allows us things we couldn't do with recursive records – rebind *self* in inherited methods:

$$\begin{array}{rcl} \texttt{new_point}(xx, yy) & = & [s.x = xx,\ s.y = yy, \\ & & s.\texttt{movex} = \lambda d : \texttt{int}.\ s + [r.x = s.x + d]\ ] \end{array}$$

Syntax:

$$e\ ::=\ \ldots \mid x \mid e.l \mid o \mid e\ \texttt{with}\ x.l = e' \tag{1}$$

$$\tau\ ::=\ \ldots \mid [\ l_i : \tau_i\ ^{i \in 1 \ldots n}] \tag{2}$$

$$o\ ::=\ [\ x_1.l_1 = e_1,\ \ldots\ , x_n.l_n = e_n\ ] \tag{3}$$

New expressions in (1) can be added to some other language to enrich it. However, it turns out that forming a language from these expressions alone suffices to make it Turing-equivalent.

No distinction is made in object calculus between fields and methods. If $x_i.l_i = e_i$ happend to be a field, it just means that x_i is a dummy variable.

Operational semantics:

$$\frac{}{o.l_i \rightarrow e_i\{o/x_i\}}$$

$$\frac{}{o \texttt{ with } x.l_j = e \rightarrow [x.l_j = e, x_i.l_i = e_i \ ^{i \in \{1 \ldots n\} - \{j\}}]}$$

Typing rules, where $o : \tau_o$ is an object as defined by (2)&(3):

$$\frac{\Gamma, x_i : \tau_o \ \vdash \ e_i : \tau_i}{\Gamma \vdash o : \tau_o}$$

$$\frac{\Gamma \vdash \ o : \tau_o}{\Gamma \vdash \ e.l_i : \tau_i}$$

$$\frac{\Gamma \vdash e_o : \tau_o \quad \Gamma \vdash e : \tau_j}{\Gamma \vdash \ (e_o \texttt{ with } w.l_i = e) : \ \tau_o}$$