Supplementary Notes on Objects

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 19
October 31, 2002

In this lecture we extend the encoding of objects using records and subtyping from the previous lecture to allow open recursion through self. We still follow Chapter 18 of Benjamin C. Pierce: *Types and Programming Languages*, MIT Press, 2002.

Classes with Self. First, we show how "self", that is, invoking of methods part of the current objects, can be encoded directly. This technique does *not* model open recursion (also called late binding of self).

The basic idea of this first approach is to use a fixed point construct in order to allow the methods of in an object to refer to the object itself. The fixpoint operator is orthogonal to all other operators in the language and can be defined with

$$\frac{\Gamma, f{:}\tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\, f.e : \tau} \qquad \frac{}{\mathsf{fix}\, f.e \mapsto \{\mathsf{fix}\, f.e/x\}e}$$

There are no new values. Type preservation requires that all three types involved in the definition of fix be the same.

Now we take a slight modification of the previous example, extending the class to allow `get`, `set`, and `inc` methods. Internally, the `inc` method refers to `set` and `get` instead of accessing the private fields directly.

```
CounterRep = {x:int ref};
SetCounter = {get:1 -> int, set:int -> 1, inc:1 -> 1};
setCounterClass : CounterRep -> SetCounter =
  λr:CounterRep.
    fix self:SetCounter.
      {get = λ_:1. !(r.x),
       set = λi:int. r.x := i,
       inc = λ_:1. self.set (self.get() + 1)};
newSetCounter : 1 -> SetCounter =
  λ_:1. let r = {x = ref 1} in setCounterClass r end;
```

To see this representation in action, we apply the operational semantics to `newSetCounter().inc()` in the empty store. Each line constitutes a state of the machine, although we skip several intermediate steps.

```
<., newSetCounter().inc()>
<., let r = {x=ref 1} in setCounterClass r end.inc()>
<c=1, (setCounterClass {x=c}).inc()>
<c=1, (fix self.
        {..., inc=λ_:1. self.set(self.get()+1)}).inc()>
```

At this point we abbreviate

```
s = fix self. {..., inc = λ_:1. self.set(self.get()+1)}
```

and continue execution with

```
<c=1, {...,inc=λ_:1. s.set(s.get()+1)}.inc()>
<c=1, λ_:1. s.set(s.get()+1)()>
<c=1, s.set(s.get()+1)>
<c=1, s.set((λ_:1. !({x=c}.x))()+1)>
<c=1, s.set(!({x=c}.x)+1)>
<c=1, s.set(2)>
<c=2, ()>
```

Open Recursion through Self. The previous encoding is perfectly adequate, yet it does not model a feature available in many object-oriented languages, namely open recursion. This feature means that in a subclass of `SetCounter` that overrides `set` but not `inc`, the references to `self` will be to the `set` and `get` methods of the *subclass*. This feature is somewhat unfortunate, because it breaks encapsulation: as we will see after we have

modeled the feature, client code will depend on internals of the implementation of a superclass.

To model this feature we move the recursion outside the object itself to the place where it is created.

```
setCounterClass : CounterRep -> SetCounter -> SetCounter =
  λr:CounterRep.
    λself:SetCounter.
      {get = λ_:1. !(r.x),
       set = λi:int. r.x := i,
       inc = λ_:1. self.set (self.get() + 1)};
newSetCounter : 1 -> SetCounter =
  λ_:1. let r = {x = ref 1}
           in fix self. setCounterClass r self end;
InstrCounterRep = {x:int ref, a:int ref}
InstrCounter =
  {get:1 -> int, set:int -> 1,
   inc:1 -> 1, accesses:1 -> int}
instrCounterClass :
  InstrCounterRep -> InstrCounter -> InstrCounter =
  λr:InstrCounterRep.
    λself:InstrCounter.
      let super = setCounterClass r self
       in
          {get = super.get,
           set = λi:int. (r.a := !(r.a)+1; super.set i),
           inc = super.inc,
           accesses = λ_:1. !(r.a)}
      end;
```

The previous problem has now been solved, yet a new problem has arisen, because creating objects of type `instrCounterClass` By using the standard technique of unit-abstractions, we can overcome this problem (see Chapter 18.11 of Pierce's book). We will not go into those details here.

We close the lecture by exhibiting that this form of late binding of self breaks encapsulation and is extremely dangerous when writing a library. It means that the behavior of a subclass can depend on specifics of the (supposedly invisible!) internal of the library class. The example we show here is taken from Item 14 of Joshua Bloch: *Effective Java*, Addison-Wesley, 2001,

The following code uses inheritance inappropriately, as the example at the end shows.

```
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet extends HashSet {
    // The number of attempted element insertions
    private int addCount = 0;
    public InstrumentedHashSet () {
    }
    public InstrumentedHashSet(Collection c) {
        super(c);
    }
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

Now the following sequence

```
InstrumentedHashSet s = new InstrumentedHashSet();
s.addAll(Arrays.asList(new String[]
          {"Snap", "Crackle", "Pop"}));
s.getAddCount();
```

may return either 3 or 6, depending on whether the library implementation of `addAll` has internal calls to `add` or not.

To the writer of this library this means he must either fully document the internal call patterns of the library, or risk breaking a lot of client code when improving internal data structures. Bloch suggests that it is often better to *prohibit* inheritance, for example, making constructors private, and using *composition* instead of inheritance. Precisely the same technique would be used in Standard ML's module system to obtain the effect by inheritance. First, the corresponding Java code.

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet implements Set {
    private final Set s;
    private in addCount = 0;
    public InstrumentedSet(Set s) {
        this.s = s;
    }
    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
    // Forwarding methods
    public void clear()               { s.clear ();           }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty()          { return s.isEmpty();   }
    // ...
    public String toString ()         { return s.toString();  }
}
```

In the place of `// ...` all the relevant public methods of `s` are exported again. In ML we would use a wrapper functor instead (assuming we really wanted the implementation of `Set` to be ephemeral):

```
signature InstrumentedSet =
sig
  include Set
  val getAddCount : unit -> int
end;
functor InstrumentWrapper (structure S : Set)
      : InstrumentedSet =
struct
  val addcount = ref 0
  open S
  fun add(o) = (addcount := !addcount+1; S.add(o))
  fun addAll(c) = (addcount := !addcount+List.length(c); S.addAll(c))
  fun getAddCount() = !addCount
end;
structure InstrumentedSet = InstWrapper (structure S = Set);
```

We would like to emphasize that open recursion is indeed an anti-modularity feature that breaks encapsulation. Any programmer that prepares libraries in a language like Java should be aware of this, and know how to avoid its pitfalls that are sometime difficult to detect.

There are many other features of object-oriented languages that we have not yet modeled. We return to some of them in the next lecture. Here we would like to discuss *overloading*. In Java it refers to the fact that a method name can be reused, as long as all its argument types are different. We only discuss it on simpler examples, namely the overloading of addition. Assume we have two internal functions, $+_{int}$ and $+_{float}$ that add integers and floating point numbers.

In the concrete syntax of the language, we would like to use $+$ and let the type checker sort out which of the two versions of addition should be used. Intersection types, in conjunction with subtyping, allow precisely that. We write $\tau \wedge \sigma$ for the intersection between τ and σ. We have the following rules:

$$\frac{\Gamma \vdash v : \tau_1 \quad \Gamma \vdash v : \tau_2 \quad v \ \mathsf{value}}{\Gamma \vdash v : \tau_1 \wedge \tau_2} \wedge I$$

$$\frac{\Gamma \vdash e : \tau_1 \wedge \tau_2}{\Gamma \vdash e : \tau_1} \wedge E_1 \qquad \frac{\Gamma \vdash e : \tau_1 \wedge \tau_2}{\Gamma \vdash e : \tau_2} \wedge E_2$$

The restriction of the intersection introduction rule to values is necessary for soundness in the presence of mutable references. The counterexample (which we do not show here) echoes the related counterexample that requires the value restriction in Standard ML.

Together with the introduction and elimination rules, we also have three subtyping rules, working on the left or the right hand side.

$$\frac{\tau \leq \sigma_1 \quad \tau \leq \sigma_2}{\tau \leq \sigma_1 \wedge \sigma_2} \wedge R$$

$$\frac{\tau_1 \leq \sigma}{\tau_1 \wedge \tau_2 \leq \sigma} \wedge L_1 \qquad \frac{\tau_2 \leq \sigma}{\tau_1 \wedge \tau_2 \leq \sigma} \wedge L_1$$

With these concepts, we can now declare

$$+ : (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \wedge (\mathsf{float} \to \mathsf{float} \to \mathsf{float})$$

Below are several judgments that check with these declarations.

$$\begin{array}{rcl} 3 + 4 & : & \mathsf{int} \\ 3.0 + 4 & : & \mathsf{float} \\ 3 + 4.0 & : & \mathsf{float} \\ 3.0 + 4.0 & : & \mathsf{float} \end{array}$$

We use the $\wedge E_1$ rule in the first case, and $\wedge E_2$ to extract the appropriate type for $+$ in each judgment. As before, we also need to coerce the integer arguments to floating point numbers, in the middle two examples.

Since we are using a coercion interpretation of subtyping here, we have to show how to interpret intersection types. From the primitive function $+$ we can see that a constant of intersection type actually corresponds to a pair of two functions

$$\mathtt{pair}(+_{int}, +_{float}) \quad : \quad (\mathsf{int} \to \mathsf{int} \to \mathsf{int}) \times (\mathsf{float} \to \mathsf{float} \to \mathsf{float})$$

From this we can extend interpretation through the whole type hierarchy. We achieve this by annotating a type derivation by the fully explicit expression it generates. We also extend the annotation of the subtyping rules to account for the new coercion. The judgment is $\Gamma \vdash e : \tau \implies e'$ where e' is an explicit term without any uses of subtyping or intersection types. It has type τ' which is generated from τ by replacing intersections ($\wedge$) by products. We write $\overline{\tau}$ for this operation.

$$\frac{\Gamma \vdash v : \tau_1 \implies v_1 \quad \Gamma \vdash v : \tau_2 \implies v_2 \quad v \ \mathsf{value}}{\Gamma \vdash v : \tau_1 \wedge \tau_2 \implies \mathtt{pair}(v_1, v_2)} \wedge I$$

$$\frac{\Gamma \vdash e : \tau_1 \wedge \tau_2 \implies e'}{\Gamma \vdash e : \tau_1 \implies \mathtt{fst}(e')} \wedge E_1 \qquad \frac{\Gamma \vdash e : \tau_1 \wedge \tau_2 \implies e'}{\Gamma \vdash e : \tau_2 \implies \mathsf{snd}(e')} \wedge E_2$$

For most other constructs the propagation of the explicitly typed term is straightforward. We show only a few further cases

$$\frac{(\Gamma = \Gamma_1, x{:}\tau, \Gamma_2)}{\Gamma \vdash x : \tau \implies x}\ var \qquad \frac{\Gamma \vdash e : \tau \implies e' \quad f : \tau \leq \sigma}{\Gamma \vdash e : \sigma \implies f(e')}\ sub$$

Subtyping introduces no new ideas.

$$\frac{f_1 : \tau \leq \sigma_1 \quad f_2 : \tau \leq \sigma_2}{\lambda x.\mathtt{pair}(f_1(x), f_2(x)) : \tau \leq \sigma_1 \wedge \sigma_2}\ \wedge R$$

$$\frac{f_1 : \tau_1 \leq \sigma}{\lambda x.f_1(\mathtt{fst}(x)) : \tau_1 \wedge \tau_2 \leq \sigma}\ \wedge L_1 \qquad \frac{f_2 : \tau_2 \leq \sigma}{\lambda x.f_2(\mathtt{snd}(x)) : \tau_1 \wedge \tau_2 \leq \sigma}\ \wedge L_2$$

Recall that $\overline{\tau}$ replaces intersections by products. We extend this operation to context by applying it to each type declaration.

Theorem 1 (Coercions)

(i) If $\tau \leq \sigma$ then $f : \tau \leq \sigma$ for some f.

(ii) If $\Gamma \vdash e : \tau$ then $\Gamma \vdash e : \tau \implies e'$ for some e' with $\overline{\Gamma} \vdash e' : \overline{\tau}$.

Proof: By rule induction on the given derivations. ■

Note that we also want f and e' to be unique. However, this can only be true extensionally, since subtyping derivations (and therefore typing derivations) are not uniquely determined. In other words, we do not formalize here the all-important property of *coherence*.

We cannot execute or define the operational semantics directly on a source expression e, because the process of inserting the coercions performs the overloading resolution. As in Java, this process is completely static, that is, happens before the program is ever executed. The above should therefore be considered a reasonable model for the kind of overloading present in object-oriented languages. Other languages, such as Haskell, permit overloading, but overloading is not resolved until run-time—this requires different techniques for both proving progress and preservation than we discuss here.