Mutable Objects

Benjamin C. Pierce
LFCS, Edinburgh

Working draft of May 29, 1993

Abstract

There are now several type-theoretic models for object-oriented languages based on higher-order polymorphic lambda-calculi [Bru92, Bru93, CCH+89, CHC90, Mit90, PT93]. It has been claimed that these models can be extended straightforwardly to handle objects with mutable state. Following a suggestion of Didier Rémy, we verify this claim for the model proposed in [PT93].

1 Overview

[PT93] proposed a statically typed model of "functional-style" object-oriented programming, where an object responds to an updating message by returning a new, appropriately modified object. But, as others had noted in earlier papers, the extension to objects with mutable state was not completely straightforward. In the conclusion, we said:

> In extending our purely functional model of objects to include imperative features, we have encountered one significant technical puzzle. The fixed point combinator used in the `new` function in Section 6 only makes sense in a language with a call-by-name evaluation strategy, which would not form a suitable basis for a statically typed language with mutable objects. This deficiency, which seems to appear in most λ-calculus models of object-oriented programming, has also been pointed out by Wand [Wan89] and Cook, Hill, and Canning [CHC90]. Using the ordinary fixed point combinator at functional types that is found in call-by-value languages causes `new` to diverge in some cases. Thus, it appears to be necessary to introduce either a fixed point combinator at all types of records of functions (which cannot be given a polymorphic typing) or a lazy record constructor.

In a recent discussion, Didier Rémy and I took a closer look at the possibility of defining a call-by-value fixed-point combinator for all records of functions. We concluded, unfortunately, that it probably would not work, since we couldn't see how to implement such a fixed-point combinator, even assuming one could find a way to give it a type. Finally he said, "But you could just do what everybody does when they move from call-by-name to call-by-value: insert a 'delay' somewhere to turn a value into a function." This, of course, works: why didn't we think of it before?

Having dealt with this point, the rest is indeed as straightforward as expected. In fact, it was not hard to modify the code generator of the experimental $F^{\omega}_{\leq}$ implementation to include references and a call-by-value evaluation strategy; this allows the type-correctness of the new encoding to be checked automatically, and gives us the chance to see some running examples.

2 A Call-By-Value Formulation of $F^\omega_\leq$

This note develops a slightly modified version of the statically typed object model of [PT93] in which objects may behave either functionally — returning "updated copies of themselves" as the results of methods — or, as is more common in mainstream object-oriented languages, imperatively — destructively modifying their instance variables in response to update messages.

The first thing we need, of course, is mutable storage cells. These are provided in the usual way, by introducing a type constructor `Ref` with a constructor `ref` of type `All(A) A->Ref(A)`. For example, the expression

```
# r = ref Int 5;
r = <val> : Ref Int
```

creates a storage cell whose initial value is the integer `5`. The contents of a storage cell can be modified with the `assign` operator, whose type is `All(A) Ref(A)->A->Unit`:

```
# assign Int r 3;
<val> : Unit
```

The `deref` operator, of type `All(A) Ref(A)->A`, provides access to the current value of a storage cell:

```
# deref Int r;
3 : Int
```

Next, it is generally agreed that languages with mutable state should use a call-by-value operational semantics, which makes it easier to reason about the time-sequencing of modifications to the store. Since [PT93] treated the operational semantics of its $F^\omega_\leq$ informally, there is no difficulty in substituting — again, informally — a call-by-value reduction order for the original call-by-name semantics. The one important proviso is that the fixed-point combinator `rec`, which has the polymorphic type `All(A) (A->A)->A` in a call-by-name language, is not definable in a call-by-value language; instead, we must use a weaker version that can only be instantiated at functional types:

```
# rec;
<val> : All(A) All(B) ((A->B)->A->B) -> A -> B
```

I shall not develop the formal theory of call-by-value $F^\omega_\leq$ here, except to note that the type system remains identical to the call-by-name variant and that this kind of change in evaluation order and addition of mutable storage cells are reasonably well understood for numerous related calculi.

3 Development

From now on, I will assume that the reader is familiar with the encoding of objects in [PT93], to which I also defer for the definition of the $F^\omega_\leq$. Most readers will probably want to skim the technical development here or simply jump to the next section for some examples. The only important point to note is that the change is essentially trivial.

For concreteness, I use the same encoding of objects as [PT93]:

```
# Object = Fun(M:*->*) Some(A) {|state:A, methods:M(A)|};
Object : (*->*)->*
```

However, nothing in what follows depends on the details of this encoding. An encoding in terms of recursive types and F-bounded quantification should work equally well.

A *class* is a data structure that can be either *instantiated* to form new objects or *extended* to form new classes. Its precise definition depends on the choice of the style of inheritance we want to use; here, I follow the definition given in Section 7 of [PT93], in which the instance variables of a class definition are visible only to the methods defined by that class, and not to the methods of subclasses.

```
# Class =
#   Fun(SelfM:*->*)
#     Some(SelfR)
#       {| localstate: Unit->SelfR,
#          buildM: ClassMethods SelfM SelfR|};
Class : (*->*)->*
```

Our new version remains close to the original: instead of an initial value for the instance variables of new objects, a class contains a *function* for creating the initial value; typically, this function will allocate some storage for the new instance variables. Also, as before, a class contains a function that (when appropriately instantiated) can be used to build the method vector of a new object. The type constructor `ClassMethods` describes the arguments that must be provided to the method-building function: a "final representation" type containing all of the instance variables of the new object being constructed, an "extractor" that can be used to map back and forth between the final representation and the "local representation" type `SelfR` (c.f. [PT93] for details), and a vector of "self methods":

```
# ClassMethods =
#   Fun(SelfM:*->*)
#   Fun(SelfR:*)
#     All(FinalR)
#     (Extractor FinalR SelfR) ->
#     (Unit -> SelfM(FinalR)) ->
#     (SelfM(FinalR));
ClassMethods : (*->*)->*->*
```

The "`Unit->`" in the type of the self method vector is the only essential change we are making to the original construction. Its purpose is to turn the type of the "abstracted method vector" in which recursive references to `self` have not yet been resolved, from a record type into a functional type. This allows the `new` function to be written essentially as before:

```
# new =
#   fun(SelfM: *->*)
#   fun(selfClass: Class SelfM)
#     open selfClass as <SelfR,selfData> in
#       <SelfR,
#        {state = selfData.localstate unit,
#         methods =
#           (rec Unit (SelfM(SelfR))
#             (fun(self:Unit->SelfM(SelfR))
#               fun(x:Unit)
#                 selfData.buildM SelfR (idExtractor SelfR) self))
#           unit}
#        > : Object(SelfM)
#     end;
new = <val> : All(SelfM:*->*) (Class SelfM) -> (Object SelfM)
```

The modifications here are:

- The the state of the new object is created by applying the state-creating function from the class to `unit`, the sole element of the type `Unit`, instead of simply copying its value from the class.
- The call-by-value fixed point constructor `rec` is used to construct a "delayed method vector" of type `Unit->(SelfM(SelfR))`.
- Finally, the delayed method vector is applied to `unit` to yield the actual method vector of the new object.

The `extend` function takes a class and a description of an incremental change to its state and behavior, and builds a new class. Except for two points where the type annotations must be modified (marked `<---`), its definition remains identical to the one in [PT93].

```
# extend =
#   fun(SuperM: *->*)
#   fun(SelfM < SuperM)
#   fun(SelfDeltaR)
#   fun(superClass: Class SuperM)
#   fun(deltastate: Unit->SelfDeltaR)
#   fun(build: All(FinalR)
#                (FinalR->SelfDeltaR) ->
#                (FinalR->SelfDeltaR->FinalR) ->
#                (SuperM(FinalR)) ->
#                (Unit->SelfM(FinalR)) ->                        % <---
#                (SelfM(FinalR)))
#     open superClass as <SuperR,superData> in
#       <Pair SuperR SelfDeltaR,
#        {localstate = fun(u:Unit)
#                        pair SuperR SelfDeltaR
#                          (superData.localstate unit)
#                          (deltastate unit),
#         buildM =
#          fun(FinalR)
#          fun(e: Extractor FinalR (Pair SuperR SelfDeltaR))
#          fun(self: Unit->SelfM(FinalR))                        % <---
#            let esnd = composeExtractors
#                         FinalR (Pair SuperR SelfDeltaR) SelfDeltaR
#                         e (sndExtractor SuperR SelfDeltaR)
#            in build FinalR esnd.get esnd.put
#              (superData.buildM FinalR
#                 (composeExtractors FinalR (Pair SuperR SelfDeltaR) SuperR e
#                                  (fstExtractor SuperR SelfDeltaR))
#                 self)
#              self
#            end}
#        > : Class SelfM
#      end;
extend = <val>
       : All(SuperM:*->*)
         All(SelfM<SuperM)
         All(SelfDeltaR)
              (Class SuperM)
```

```
          -> (Unit->SelfDeltaR)
          -> (All(FinalR)
                  (FinalR->SelfDeltaR)
                ->(FinalR->SelfDeltaR->FinalR)
                ->(SuperM FinalR)
                ->(Unit->(SelfM FinalR))
                ->(SelfM FinalR))
          -> (Class SelfM)
```

Finally, we define an empty object interface `ThingM`,

```
# ThingM = Fun(R:*) {| |};
ThingM : *->*
```

and a class `thingClass`, which forms the root of the inheritance hierarchy:

```
# thingClass =
#   <Unit,
#    {localstate = fun(u:Unit) unit,
#     buildM = fun(FinalR)
#              fun(e: Extractor FinalR Unit)
#              fun(self: Unit->ThingM(FinalR))
#                {} }
#   > : Class ThingM;
thingClass = <val> : Class ThingM
```

4 Example

To illustrate the use of classes to build mutable objects, let's take a variant of the standard example of points and colored points. Points should support the following methods:

- `setX`: set the x coordinate to a new value.
- `getX`: return the current x coordinate.
- `bump`: increase the x coordinate by one.
- `copy`: return a new point with the same x coordinate.

Here, the `setX`, `getX`, and `bump` methods are imperative, so their result types are just `Unit`. The `copy` method, on the other hand, returns a new point; its typing illustrates the fact that we can still program with "functional style" messages to objects that cause them to return altered copies of themselves. (In this case, the alteration is that new storage cells are allocated for the instance variables, so that the original and the copy may be updated separately.)

For the sake of the exercise, we further stipulate that the `bump` method should be implemented in terms of the `setX` and `getX` methods, so that if it is inherited by a new class that overrides `setX` or `getX`, the behavior of `bump` in the subclass will change accordingly.

The colored point class is a subclass of the point class that adds two methods for setting and returning the point's color. In addition, this class overrides the `setX` method of the point class so that the color is changed to blue whenever the point is moved. The copy method copies the storage cell holding the color as well as the x coordinate.

The interfaces of point and colored point objects are specified by the following type operators:

```
# PointM = Fun(R:*) {| setX: R->Int->Unit,
#                      getX: R->Int,
#                      bump: R->Unit,
#                      copy: R->R
#                   |};
PointM : *->*

# CPointM = Fun(R:*) {| setX: R->Int->Unit,
#                       getX: R->Int,
#                       bump: R->Unit,
#                       copy: R->R,
#                       setC: R->Color->Unit,
#                       getC: R->Color
#                    |};
CPointM : *->*
```

The implementations of the message-sending functions `Point'setX`, `Point'bump`, etc. are identical to those in [PT93], so I elide them here. (The types of the imperative ones could be simplified a bit by removing the quantifier, but I leave it for consistency.)

```
# Point'setX;
<val> : All(M<PointM) (Object M) -> Int -> Unit
# Point'getX;
<val> : All(M<PointM) (Object M) -> Int
# Point'bump;
<val> : All(M<PointM) (Object M) -> Unit
# Point'copy;
<val> : All(M<PointM) (Object M) -> (Object M)
# CPoint'setC;
<val> : All(M<CPointM) (Object M) -> Color -> Unit
# CPoint'getC;
<val> : All(M<CPointM) (Object M) -> Color
```

Now let us build the point class. Its representation type is a record containing just one field, a reference to an integer:

```
# PointR = {|x: Ref Int|};
PointR : *
```

We form `pointClass` by applying the `extend` function to `thingClass`. The additional information we have to supply is:

- A function that creates new instances of the representation type.
- A method-building function that takes a final representation type, functions for mapping the final representation to and from the local representation, a vector of superclass methods (empty in this case, since the superclass is `thingClass`), and a vector of "self-methods" (which is abstracted on `Unit` so that the typing of the fixed point in `new` will work out), and returns the vector of point-methods.

The point methods themselves are built in the obvious way: `setX` uses the `assign` primitive to change the value of the `x` component of the state; `getX` uses `deref` to read its value; `bump` just calls `getX` and `setX`; and `copy` creates a new storage cell with the appropriate initial value and packages it together with the rest of the "final state."

```
# pointClass =
#  extend ThingM PointM PointR thingClass
#   (fun(u:Unit) {x = ref Int 0})
#   (fun(FinalR)
#    fun(get: FinalR->PointR)
#    fun(put: FinalR->PointR->FinalR)
#    fun(super: ThingM(FinalR))
#    fun(self: Unit->PointM(FinalR))
#      {setX = fun(s:FinalR) fun(i:Int) assign Int (get s).x i,
#       bump = fun(s:FinalR) (self unit).setX s (plus 1 ((self unit).getX s)),
#       copy = fun(s:FinalR) put s {x = ref Int (deref Int (get s).x)},
#       getX = fun(s:FinalR) deref Int (get s).x})
#   : Class PointM;
pointClass = <val> : Class PointM
```

Similarly, we create the colored point class `cpointClass` from `pointClass` by calling extend with appropriate arguments. Here, the `getX` and `bump` methods are simply copied from the method vector `super` provided by `pointClass`. The `setX` and `copy` methods are overriden: `setX` changes the color field to `blue` and then calls the `setX` method of `pointClass`; `copy` calls the `copy` method of `super` to copy the `x` field and then does the additional work of copying the `color` field.

```
# CPointR = {|color: Ref Color|};
CPointR : *
# cpointClass =
#  extend PointM CPointM CPointR pointClass
#   (fun(u:Unit) {color = ref Color white})
#   (fun(FinalR)
#    fun(get: FinalR->CPointR)
#    fun(put: FinalR->CPointR->FinalR)
#    fun(super: PointM(FinalR))
#    fun(self: Unit->CPointM(FinalR))
#      {setX = fun(s:FinalR) fun(i:Int)
#                let s' = super.setX s i
#                in assign Color (get s).color blue
#                end,
#       getX = super.getX,
#       bump = super.bump,
#       copy = fun(s:FinalR)
#                let s' = super.copy s
#                in put s' {color = ref Color (deref Color (get s).color)}
#                end,
#       setC = fun(s:FinalR) fun(c:Color) assign Color (get s).color c,
#       getC = fun(s:FinalR) deref Color (get s).color})
#   : Class CPointM;
cpointClass = <val> : Class CPointM
```

To test what we've done, let us create a new colored point, check its initial position and color, send it the `bump` message, and check that both position and color have changed:

```
# c = new CPointM cpointClass;
c = <val> : Object CPointM
# Point'getX CPointM c;
0 : Int
# CPoint'getC CPointM c;
```

```
white : Color
# Point'bump CPointM c;
<val> : Unit
# Point'getX CPointM c;
1 : Int
# CPoint'getC CPointM c;
blue : Color
```

To test `copy`, we can make a copy of `c`, bump `c`'s position again, and check that the copy's position is unchanged:

```
# d = Point'copy CPointM c;
d = <val> : Object CPointM
# Point'bump CPointM c;
<val> : Unit
# Point'getX CPointM d;
1 : Int
```

5 Discussion

This goes to show that lambda-calculus models of functional objects can indeed be extended to include the more realistic case of mutable objects in a call-by-value semantics. Of course, the discussion remains at a theoretical level: although these definitions of objects, classes, and inheritance appear to correctly encode the observable behaviors and desired typings of these features in object-oriented languages like Smalltalk, a number of issues remain to be considered before such encodings can form the basis of realistic designs for new high-level languages.

To begin with, all the methods that are reused from a superclass must explicitly be copied into the subclass by the method extension function given to `extend`. In a high-level object-oriented language where all class definitions always appear at the top level, this copying can easily be performed by the compiler. But in a language with "first-class classes," where `extend` can be applied to a class passed as a parameter to a function, extensible records seem to be needed to avoid the explicit copying.

Next, the use of "extractors" here is unpleasantly explicit: essentially, functions are being used to encode the manipulation of "pointers into the instance variable vector" that a compiler for an object-oriented language might generate to tell compiled methods where to access their instance variables. While this has the advantage of allowing the encoding of objects to be expressed in a more economical formal framework, it imposes a heavy burden in terms of readability and it does not correspond closely to real implementations. Again, well-known techniques involving extensible records [Wan87, Rém89, CM91, etc.] could be used here to obtain clearer — and perhaps more realistic — encodings.

A final concern is that, in this encoding, each new object is created with a fresh vector of methods, formed by taking a fixed point of the abstracted method vector from the class and applying the result to `unit`. This has the unrealistic consequence that method vectors are never shared, even among objects from the same class. Fortunately, it is easy to write a version of `new` where the method vector is built separately, before the local state:

```
# new_with_sharing =
#   fun(SelfM: *->*)
#   fun(selfClass: Class SelfM)
#     open selfClass as <SelfR,selfData> in
```

```
#     let methods =
#           (rec Unit (SelfM(SelfR))
#             (fun(self:Unit->SelfM(SelfR))
#               fun(x:Unit)
#                 selfData.buildM SelfR (idExtractor SelfR) self))
#           unit
#     in
#       fun(u:Unit)
#         <SelfR,
#          {state = selfData.localstate unit,
#           methods = methods}
#         > : Object(SelfM)
#     end end;
new_with_sharing = <val>
                 : All(SelfM:*->*) (Class SelfM) -> Unit -> (Object SelfM)
```

When `new_with_sharing` is applied to a class, it returns a *function* that, each time it is applied to `unit`, yields a new instance with a fresh state component but with the same method vector in every case.

It is interesting to note that changing from a call-by-name / functional style of object-oriented programming to a call-by-value / imperative style involves no change whatsoever to the basic model of objects, message passing, and subtyping. Only the definition of classes and the implementation of inheritance are affected. This lends further support to the feeling that inheritance should not be considered a fundamental aspect of objects, but simply as a useful collection of programming idioms for managing the creation of new objects.

References

[Bru92] Kim B. Bruce. A paradigmatic object-oriented language: Design, static typing and semantics. Technical Report CS-92-01, Williams College, January 1992.

[Bru93] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990. To appear in [GM93].

[CM91] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. To appear in [GM93]; also available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[GM93] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, 1993. To appear.

[Mit90] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 109–124, January 1990. To appear in [GM93].

[PT93] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 1993. To appear; a preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title “Object-Oriented Programming Without Recursive Types”.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*, pages 242–249. ACM, January 1989. To appear in [GM93].

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 92–97, Pacific Grove, CA, June 1989.