A Model of Delegation Based on Existential Types

Benjamin C. Pierce
Inria-Roquencourt

Working draft of April 3, 1993

Abstract

Type-theoretic models based on recursive types have been given for both class-based [Bru92, Bru93, Car92, Wan87, CCH+89] and delegation-based [Aba93, Mit90, MHF93] object-oriented languages. [PT93] proposes an alternative encoding of a class-based language, in which existential types rather than recursive types are used to achieve the encapsulation of object states with their methods. We sketch here a similar construction for a delegation-based language.

We assume that the reader is familiar with the encoding of objects presented in [PT93], to which we also defer for the definition of the underlying lambda-calculus $F^{\omega}_{\leq}$ and the notational conventions of the $F^{\omega}_{\leq}$ compiler used to typeset the examples here.

The `Object` type constructor of [PT93]

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

maps an interface type `M` of objects into the type `Object M`, whose elements are objects with the interface `M`. For example, if we have the interfaces

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

then the types

```
# Point = Object PointM;
Point : *
# CPoint = Object CPointM;
CPoint : *
```

describe sets of one-dimensional point and colored point objects. As remarked in [PT93], the methods of such objects cannot be inherited, since they are overly specific in two crucial ways:

1. The methods of an object are specialized to work on internal states of one particular type — typically records with a fixed collection of fields. But subclass definitions can add new fields. In order to deal with this flexibility, the methods of a class must be *polymorphic* in the final representation type.

2. The methods of an object are essentially functions from states to states. But to implement the behavior of `self`, where methods defined in superclasses can "see" overriden definitions of the recursive calls they make to the "self methods," it is necessary to postpone the decision of which record of methods `self` refers to; in a given class, instances of `self` do not necessarily refer to the methods of that class, but perhaps to the methods of some subclass that has not yet been defined. Thus, the methods in a class should be thought of as functions from `self` to functions from states to states.

A *class*, then, is precisely an object with enough of the packaging left off and enough decisions about representation and recursive self-reference postponed that it can be extended in these ways.

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

When the class is instantiated to form an object, the representation and then references to `self` are fixed and the methods all become concrete functions.

For example,

```
# PC = Class PointM;
PC : *
```

is a class whose instances are point objects. To instantiate a class, yielding an object, we must now supply:

- A final representation type `SubR` containing not only the "point state" `Int` but also any other state components that may be needed by some subclass of `PC`.
- A function `get : SubR -> Int` that extracts the "point state" from an element of `SubR`.
- A function `put : SubR -> Int -> SubR` that overwrites the "point state" component of an element of `SubR` with a new integer.
- A record of "self methods" containing implementations of `setX`, `getX`, and `bump` (perhaps not the versions supplied by `PC` but overridden versions provided by some subclass).

Supplying these parameters is the task of the `new` function:

```
# new =
#   fun(SelfM:*->*)
#   fun(selfClass: Class SelfM)
#     open selfClass as <SelfR,selfData> in
#       < SelfR,
#         {state = selfData.localstate,
#          methods =
#            rec (SelfM SelfR)
#              (fun(self:SelfM SelfR)
#                selfData.buildM SelfR
#                   (fun(s:SelfR) s)
```

```
#                    (fun(s:SelfR) fun(s':SelfR) s')
#                    self)}
#        >: Object SelfM
#      end;
new = <val> : All(SelfM<Top(*->*)) (Class SelfM) -> (Object SelfM)
```

If we want to do without classes — the whole point of the present exercise! — then we must loosen the encoding of objects so that it incorporates both of these kinds of flexibility. In particular, the `Object` type constructor must allow that the method vector of an object still be abstracted on a final representation type and a vector of "self methods":

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

This formulation of `Object` can be viewed as a combination of the old `Object` and `Class` constructors.

As before, we have:

```
# Point = Object PointM;
Point : *
# CPoint = Object CPointM;
CPoint : *
```

That is, the interface types of objects are not sensitive to the change we have made in the object model. However, the implementation of an object must provide an appropriately abstracted record of methods. For example, a typical `Point` object is:

```
# p = <Int,
#        {state = 5,
#         methods = fun(SubR)
#                    fun(get: SubR->Int)
#                    fun(put: SubR->Int->SubR)
#                    fun(m: PointM(SubR))
#                       {setX = fun(r:SubR) fun(i:Int) put r i,
#                        getX = fun(r:SubR) get r,
#                        bump = fun(r:SubR) m.setX r (plus (m.getX r) 1)}
#         }
#      > : Object(PointM);
p = <val> : Object PointM
```

Note that the behavior of `bump` here depends on implementations of the `setX` and `getX` methods that it obtains from the "self method vector" `m`.

In the object model of [PT93], sending messages to objects could be accomplished simply by opening the existential quantifier, extracting the implementations of the methods, applying the appropriate one ot the internal state, and repackaging the result as a new object if necessary. Here, there is a bit more work to do, since, to obtain a function that can be applied to the state, we must instantiate the abstracted record of methods with the type of the state and (by taking a fixed point) with the record of methods itself.

```
# Point'getX =
#   fun(M < PointM)
#     fun(p: Object(M))
#       open p as <R,r> in
#         let m = rec (M(R))
#                   fun(mr:M(R))
#                     r.methods
#                       R
#                       (fun(r:R) r)
#                       (fun(r:R) fun(r':R) r')
#                       mr
#         in m.getX r.state
#         end
#       end;
Point'getX = <val> : All(M<PointM) (Object M) -> Int
```

These tasks were formerly the responsibility of the `new` function. Basically, the move from a class-based formulation of inheritance to a delegation-based formulation consists precisely in postponing "the moment where the knot is tied" from object-creation time to message-sending time.

Similarly, we can implement a function for sending the `bump` message. It is identical to `Point'getX` except that after building the record of methods and applying the appropriate one to the state, it must package the resulting state with the old methods — the original abstracted ones, not the concrete ones it has built! — to form a new instance of the same object type as its argument:

```
# Point'bump =
#   fun(M < PointM)
#   fun(p: Object(M))
#       open p as <R,r> in
#         let m = rec (M(R))
#                   fun(mr:M(R))
#                     r.methods
#                       R
#                       (fun(r:R) r)
#                       (fun(r:R) fun(r':R) r')
#                       mr
#         in <R,
#             {state = m.bump r.state,
#              methods = r.methods}
#            > : Object(M)
#         end end;
Point'bump = <val> : All(M<PointM) (Object M) -> (Object M)
```

The implementation of `Point'setX` is analogous.

Programs that manipulate points by sending them messages have exactly the same form as in [PT93]:

```
# Point'getX PointM (Point'bump PointM (Point'setX PointM p 3));
4 : Int
```

Note that the variation in object model is visible only inside the parts of the code concerned with *building* objects and implementing the low-level details of sending them messages; the parts of the code that *use* objects cannot tell the difference.

Now we have reached the point where we can actually demonstrate an instance of *delegation*, in which the behavior of one object is defined as an incremental extension of the behavior of another object. We construct a colored point `c` that delegates the `Point` operations `bump` and `getX` to the point object `p`, while, for the sake of the example, providing its own `setX` that changes the color to `red`:

```
# c = open p as <R,r> in
#     let MyR = {| mystate:Color, delegated_state: R |} in
#     <MyR,
#       {state = {mystate = blue, delegated_state = r.state},
#        methods = fun(SubR)
#                   fun(get: SubR->MyR)
#                   fun(put: SubR->MyR->SubR)
#                   fun(m: CPointM(SubR))
#                     let super = r.methods SubR
#                                   (fun(s:SubR) (get s).delegated_state)
#                                   (fun(s:SubR) fun(s':R)
#                                       put s {mystate = (get s).mystate,
#                                              delegated_state = s'})
#                                   m
#                     in
#                       {setC = fun(r:SubR) fun(c:Color)
#                                 put r {mystate=c,
#                                        delegated_state=(get r).delegated_state},
#                        getC = fun(r:SubR) (get r).mystate,
#                        setX = fun(r:SubR)
#                                 super.setX
#                                  (put r
#                                    {mystate=red,
#                                     delegated_state=(get r).delegated_state}),
#                        getX = super.getX,
#                        bump = super.bump}
#                     end
#        }
#     > : Object(CPointM)
#     end end;
c = <val> : Object CPointM
```

Since we are working with a rather low-level style of lambda-calculus encoding, without the benefit of record extension operators or special syntax for inheritance, this code may strike the eye as a bit dense. But its behavior is very simple:

- First, we open `p` to reveal (abstractly) its internal representation type `R` and its state and methods `r`.
- We build a representation type `MyR` for the new colored point by pairing together the old `R` (the field `delegated_state`) with the type of the new information needed to represent the color of colored points (the field `mystate : Color`).
- We then build a new colored point object by packaging together
  - the representation type `MyR`;
  - an instance of `MyR` formed by choosing an initial value (`blue`) for the color part and copying the initial value (`r.state`) of `p`.
  - a record of methods, abstracted as usual on a type `SubR`, two functions `get` and `put`, and a record `m` of "self methods." Given these parameters, the implementations of `c`'s

methods are formed by first building a record of "super-object" methods by instantiating `r.methods` with `SubR`, `m`, and appropriately modified versions of `get` and `put`. Then `setC`, `getC`, and `setX` are implemented directly, while `getX` and `bump` are inherited by copying them directly from `super`. (Note that `setX` also uses a call to the `setX` of `super`.)

Note that, as expected, the overriden `setX` method of `c` is visible in the behavior of the `bump` method that `c` inherits from `p`:

```
# CPoint'getC CPointM c;
blue : Color
# c1 = Point'bump CPointM c;
c1 = <val> : Object CPointM
# CPoint'getC CPointM c1;
red : Color
```

References

[Aba93] Martín Abadi. Baby Modula-3 and a theory of objects. Research Report 95, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1993.

[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.

[Car92] Luca Cardelli. Extensible records in a pure calculus of subtyping. Research report 81, DEC Systems Research Center, January 1992. To appear in [GM93].

[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.

[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.

[MHF93] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *1993 IEEE Symposium on Logic in Computer Science*, June 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".

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