Comparing Object Encodings*

Kim B. Bruce† Williams College
Luca Cardelli DEC SRC
Benjamin C. Pierce‡ Indiana University

January 27, 1997

Abstract

Recent years have seen the development of several foundational models for statically typed object-oriented programming. But despite their intuitive similarity, differences in the technical machinery used to formulate the various proposals have made them difficult to compare.

Using the typed lambda-calculus $F^{\omega}_{<:}$ as a common basis, we now offer a detailed comparison of four models: (1) a recursive-record encoding similar to the ones used by Cardelli [Car84], Reddy [Red88, KR94], Cook [Coo89, CHC90], and others; (2) Hofmann, Pierce, and Turner's existential encoding [PT94, HP95]; (3) Bruce's model based on existential and recursive types [Bru94]; and (4) Abadi, Cardelli, and Viswanathan's type-theoretic encoding [ACV96] of a calculus of primitive objects.

1 Introduction

Over the last half decade, several authors have proposed foundational models for statically typed object-oriented programming. Although their motivating intuitions and the technical machinery they use are all strongly related to typed lambda-calculi with subtyping [Car84, CW85, CG92], stylistic differences have made rigorous comparisons difficult. For example, some models are presented as translations from high-level object syntax into the syntax of a typed lambda-calculus; others map high-level syntax directly into a denotational model; still others focus on the object syntax as a primitive calculus in its own right.

In this paper we compare four of these models. The first of these, based on recursively-defined records, was introduced by Cardelli [Car84] and studied in many variations by Kamin and Reddy [Red88, KR94], Cook and Palsberg [CP89], and Mitchell [Mit90]. In its untyped form, this model was used rather effectively for the denotational semantics of untyped object-oriented languages. In its typed form, it was used to encode individual object-oriented examples, but had difficulties with uniform interpretations of typed object-oriented languages. The most successful effort in this direction was carried out by Cook et al. [CHC90, CCH+89].

In 1993, Pierce and Turner [PT94] introduced an encoding that relied only on a type system with existential types, but no recursive types. This led Hofmann and Pierce [HP95] to the first uniform, type-driven interpretation of objects in a functional calculus.

*Corresponding author: Kim Bruce (kim@cs.williams.edu), Department of Computer Science, Williams College, Williamstown, MA 01267

†Partially supported by NSF grant CCR-9424123.

‡Supported by EPSRC grant GR/K 38403.

At the same conference in 1993, Bruce presented a paper [Bru94] on the semantics of a functional object-oriented language. This semantics was originally presented as a direct mapping into a denotational model of $F^{\omega}_{<:}$, but has recently been reformulated as an object encoding that depends on both existential and recursive types.

Meanwhile, frustrated by the difficulties of encoding objects in lambda calculi, Abadi and Cardelli introduced a calculus of primitive objects [AC96]. Later, however, Abadi, Cardelli, and Viswanathan [ACV96] discovered a faithful encoding of that object calculus in terms of bounded existentials and recursive types. (The encoding is simplified in this paper to facilitate comparisons with the other encodings; in particular, method update is only considered in Section 4.8).

In this paper we examine these object encodings and compare their strengths and weaknesses. Points of comparison include the expressiveness of the object-oriented constructs that can be encoded, the simplicity of the encoding, the uniformity of the encoding (e.g., independence of the encoding from the types of the objects and methods), and the power and proof-theoretic tractability of the underlying type theory used by the encoding.

We concentrate, throughout, on the lambda-calculus expressions that form the *targets* of the four encodings, eliding the associated "primitively object-oriented" source languages and the encoding functions mapping these into the lambda-calculus. (There are interesting comparisons to be made at this level too, but they are complicated by many inessential syntactic and stylistic differences between source languages.) Thus, the phrase "object encodings" in the title of the paper can be read as "object-oriented programming styles in typed lambda-calculus."

We also stop short of considering classes and subclassing mechanisms. These are of course supported—in interestingly different ways—by all four encodings, but a detailed comparison falls outside the scope of this study.

2 Technical Preliminaries

The "ambient type theory" in which our four encodings are expressed is the omega-order polymorphic lambda-calculus with subtyping, System $F^{\omega}_{<:}$ [Car90, PT94, HP95, PS97], extended with existential types [MP88], recursively defined types [AC93], recursive functions, and records. In the interest of brevity, we assume that readers have some prior familiarity with $F^{\omega}_{<:}$, with recursive types, and with the use of existential types for information hiding *à la* Mitchell and Plotkin. (Prior familiarity with some of the encodings we discuss will also be helpful, but is not required.) In this section, we sketch the syntax of the language and briefly discuss a few technical points of particular relevance to what follows.

The sets of kinds, types, and terms are given by the following grammar:

`K`	::=	`Type`	kind of types
	|	`K->K`	kind of type operators
`T`	::=	`X`	type variable
	|	`Fun(X:K)T`	type operator
	|	`T T`	application of a type operator
	|	`Top(K)`	maximal type of kind K
	|	`T->T`	function type
	|	`All(X<:T)T`	universally quantified type
	|	`Some(X<:T)T`	existentially quantified type
	|	`Rec(X)T`	recursive type
	|	`{l:T...l:T}`	record type
`e`	::=	`x`	variable
	|	`fun(x:T)e`	abstraction
	|	`e e`	application
	|	`fun(X<:T)e`	type abstraction
	|	`e T`	type application
	|	`pack [X,e] as T`	existential package construction
	|	`open e as [X,x] in e`	existential package use
	|	`{l=e...l=e}`	record construction
	|	`e.l`	field selection
	|	`let x=e in e`	local definition
	|	`letrec x(y:T):T = e in e`	recursive local definition

We assume standard definitions of reduction and conversion, writing $\mathtt{m} =_\beta \mathtt{n}$ to indicate that `m` and `n` are convertible. Although we shall perform conversion steps in whatever order is convenient for the sake of examples, we could just as well impose a call-by-name reduction strategy. (Most of the examples would diverge under a call-by-value strategy. This can be repaired, at the cost of some extra lambda-abstractions and applications to delay evaluation at appropriate points.)

We are informal about kinding throughout the paper. In particular, we omit kind declarations on type abstractions, writing `Fun(X)T` instead of `Fun(X:K)T`.

In the definitions of the encodings, we use pairs in addition to records; these can, of course, be encoded straightforwardly. We write `(m,n)` for the pair of `m` and `n`, and use the selectors `fst` and `snd` to destruct pairs. `S*T` is the type of pairs of `S` and `T`.

Our formulation of existential types is standard, following (for example) Mitchell and Plotkin's. The expression `pack [U,v] as T` creates a value of the existential type `T`. The expression `open o as [X,x] in b` unpacks an existential, yielding bindings for the type variable `X` and the term variable `x`, whose scope is the expression `b`. The rules for introduction and elimination of existentials are the usual ones. Informally:

$$\frac{\mathtt{T} =_\beta \mathtt{Some(X)S} \qquad \vdash \mathtt{v}:[\mathtt{X} \mapsto \mathtt{U}]\mathtt{S}}{\mathtt{(pack\ [U,v]\ as\ T):T}} \qquad \text{(T-PACK)}$$

$$\frac{\vdash \texttt{o:Some(X)S} \qquad \texttt{x : S} \vdash \texttt{b B} \qquad \texttt{X} \notin FV(\texttt{B})}{\texttt{(open o as [X,x] in b):B}} \tag{T-Unpack}$$

Note the important side condition on the rule `T-Unpack` of existentials. If the side condition failed, then the hidden state `X` would escape the abstraction.

In examples, we also use the informal pattern-matching notation

```
open o as [X,(s,m)] in b
```

to abbreviate

```
open o as [X,x] in let s=fst(x) in let m=snd(x) in b.
```

For example the following defines a simple abstraction which can return an integer:

```
abstr  ≝  pack [{x:String},
                ( {x=``source''},
                  fun(s:{x:String}) length(s.x)]
             as Some(X) X x (X -> Int)
          :Some(X) X x (X -> Int)
```

We can use `abstr` by "opening" it and applying the second component to the first component:

```
open abstr as [X,(x,f)] in f(x)
```

Because the type of `f(x)` does not involve `X`, this is legal according to `T-Unpack`. However, replacing `f(x)` by `x` or `f(x++``more'')` is illegal according to `T-Unpack` as they would break the abstraction.

The `letrec` construct allows us to define terms using auxiliary functions (which may be defined recursively):

$$\frac{\texttt{f:S->T, x : S} \vdash \texttt{e : T} \qquad \texttt{f:S->T} \vdash \texttt{b:B}}{\texttt{(letrec f(x:S):T = e in b):B}} \tag{T-LetRec}$$

The following folding and unfolding rules allow us to make use of recursive types:

$$\texttt{Rec(X)I(X) <: I(Rec(X)I(X))} \tag{S-Unfold}$$

$$\texttt{I(Rec(X)I(X)) <: Rec(X)I(X)} \tag{S-Fold}$$

We will use these rules implicitly as needed rather than clutter the presentation.

The "Amber rule" is very useful in determining when recursively-defined types are subtypes:

$$\frac{\texttt{X <: Y} \vdash \texttt{I(X) <: J(Y)}}{\texttt{Rec(X)I(X) <: Rec(Y)J(Y)}} \tag{S-Rec}$$

For subtyping quantifiers, we have a choice of rules. Some of our encodings will work fine with the Kernel Fun fragment. One needs the full Fsub rule. The following is the Kernel Fun rule for bounded polymorphic functions.

$$\frac{\vdash \texttt{A} <: \texttt{C} \qquad \texttt{X} <: \texttt{A} \vdash \texttt{D(X)} <: \texttt{B(X)}}{\texttt{forall(X} <: \texttt{A)D(X)} <: \texttt{forall(X} <: \texttt{A)B(X)}} \quad \text{(Kernel Fun poly <:)}$$

Notice that the bounds on the type parameters are identical for Kernel Fun. To allow more flexibility, the full `Fsub` rule may be used:

$$\frac{\vdash \texttt{A} <: \texttt{C} \qquad \texttt{X} <: \texttt{A} \vdash \texttt{D(X)} <: \texttt{B(X)}}{\texttt{forall(X} <: \texttt{C)D(X)} <: \texttt{forall(X} <: \texttt{A)B(X)}} \quad \text{(Fsub poly <:)}$$

We can extend the subtyping relation to type functions (functions from types to types) by defining subtyping pointwise. Thus if `F` and `G` are type functions then define `F <: G` iff for all types `X`, `F(X) <: G(X)`. For example, if `G(X)` = {`bump:X, eq:X->Bool`} and `F(X)` = {`bump:X, eq:X->Bool, set:Int->X`} then `F <: G`

3 The Encodings

Our running example throughout the paper will be (purely functional) integer reference cell objects.[1] The interface of cell objects is represented by the following type operator:

```
CellI(X)  ≝  {get:Int, set:Int->X, bump:X}
```

Operationally, a cell object has three methods: `get`, which returns its current contents; `set`, which returns a new cell object (we intend that the contents of the resulting object should be set to the integer provided as a parameter, although of course the interface type doesn't guarantee this); and `bump`, which returns a new cell (whose contents should be one greater than the current contents). The role of the parameter `X` varies between the encodings we consider, but it may be thought of intuitively as a placeholder for the "type of self." Given an interface `I`, we write `O(I)` for the type of "objects with interface `I`."

We are interested in the properties of `O(I)` for different values of `O` — i.e. for different ways of encoding objects with interface `I`. The four `O`'s that we consider in detail are:

```
OR(I)    ≝  Rec(X)                        I(X)
OE(I)    ≝          Some(Y)     Y x (Y->I(Y))
ORE(I)   ≝  Rec(X)  Some(Y)     Y x (Y->I(X))
ORBE(I)  ≝  Rec(X)  Some(Y <: X) Y x (Y->I(Y))
```

`OR` is a "classical" recursive-record encoding. `OE` is the "existential encoding" of Hofmann, Pierce, and Turner [PT94, HP95]. `ORE` is a type-theoretic analog of Bruce's denotational semantics for objects based on F-bounded quantification [Bru94]. `ORBE` is a variant of Abadi, Cardelli, and Viswanathan's type-theoretic encoding [ACV96]. The names are designed to remind the reader of the main features of the encodings: `R` stands for recursive types, `E` for existential types, and `BE` for bounded existentials.

[1] We concentrate here on the purely functional versions of each of the encodings. This choice aids both in formulating each of the systems (for example, it allows us to assume a call-by-name reduction strategy, avoiding some extra thunking that would be needed in the corresponding call-by-value variants) and in later comparisons between systems.

The use of type operators (rather than just types) to represent object interfaces uniformly captures two different points of view about the types of the objects methods: the "external view" of the object, in which the methods are abstract services that can only be invoked by an operation of "message sending," and the "internal view" of the object when it is being created, in which the methods are concrete values. The internal view of the methods' types varies from encoding to encoding (in two encodings `I` is applied to the recursively bound type variable `X`, while in the other two it is applied to the existentially bound variable `Y`.) On the other hand, the external view will always the same:

```
CellMessages  def=  O(CellI) -> CellI(O(CellI))
                =   O(CellI) -> {get:Int, set:Int->O(CellI), bump:O(CellI)}
```

That is, the messages supported by cell objects can be viewed as a collection of functions whose first parameter (the "self parameter") is a cell object and whose results are described by `CellI(O(CellI))`.

It is technically convenient to write a single self parameter at the front of the whole collection of messages instead of abstracting each message individually on its self parameter. For example, for most of the paper we will assume that object interfaces are represented by *covariant* type operators, in which the bound variable appears only in positive positions. That is, each method of an object implicitly takes a single self parameter and can then return results of the self parameter type but not take any more arguments of this type. Section 4.6 discusses the implications of relaxing this restriction to allow "binary" methods with parameters of the same type as the receiver. See [BCC+96] for a more extended discussion.

We now develop each of the encodings in detail, using the example of cells to illustrate each one.

3.1 `OR`: Recursive records

The encoding of recursive records is fairly straightforward:

```
OR(I)  def=  Rec(X) I(X)
```

In this case an object is simply a recursive record in which each occurrence of X stands for the type of the entire record. Thus if `T` = `OR(I)` then `T` = `I(T)`.

We can encode a cell object as follows:

```
mycell  =  letrec mkobj(s:{x:Int}) : OR(CellI) =
             { get = s.x,
               set = fun(n:Int) mkobj({x=n}),
               bump = mkobj({x=s.x+1}) }
           in
             mkobj({x=0})
        : OR(CellI)
```

The recursive function `mkobj` creates a new object of type `OR(CellI)`, given a value for the internal state. (Note that, if we wanted to enforce a call-by-value reduction scheme, it would be necessary to change the `bump` field from an object to a function of no arguments returning an object; otherwise a call to `mkobject` would always diverge.)

Let us introduce the syntax `o<=l` for sending a message `l` to an object `o`. Because objects are simply recursively defined records, message sending is represented by field selection (after unfolding the recursive type):

$$\texttt{o<=l} \stackrel{\text{def}}{=} \texttt{o.l}$$

It is easy to see that `(mycell<=bump)<=get` reduces to `1` as follows:

```
(mycell<=bump)<=get
       =β (mkobj({x=0})<=bump)<=get
       =β mkobj({x=({x=0}.x+1)})<=get
       =β mkobj({x=1})<=get
       =β {x=1}.x
       =β 1
```

Instead of implementing `bump` by manipulating the state directly, suppose we want to implement it in terms of the other methods. We can write:

```
mycell  ≝  letrec mkobj(s:{x:Int}) : OR(CellI) =
              let self = mkobj s in
              { get = s.x,
                set = fun(n:Int) mkobj({x=n}),
                bump = self<=set(self<=get + 1) }
            in
              mkobj({x=0})
          : OR(CellI)
```

3.2 `OE`: Existentials

In this encoding, we treat objects as pairs of state (with type `Y`) and methods (with type `Y->I(Y)`), in which the state component is hidden from the outside and methods are functions that depend on the state. Thus

$$\texttt{OE(I)} \stackrel{\text{def}}{=} \texttt{Some(Y) Y x (Y->I(Y))}$$

where the type variable `Y` represents the hidden state.

We can define a cell object as follows:

```
mycell  ≝  pack [{x:Int},
                 ( {x=0},
                   fun(s:{x:Int})
                     {get = s.x,
                      set = fun(n:Int) {x=n},
                      bump = {x=s.x+1} })]
             as OE(CellI)
          : OE(CellI)
```

It is now slightly more complex to send messages as we must "unpack" elements of existential type before we can access their components. Simple message sends like `get` are encoded as:

```
o<=get  def=  open o as [X,(s,m)] in m(s).get
```

That is, we open the existential, apply the method suite to the state, and then extract the appropriate method.

However, messages like bump that return new objects with updated internal state require a bit more, since the resulting object must be re-packed.

```
o<=bump  def=  open o as [X,(s,m)] in
                 pack [X,(m(s).bump,m)]
                 as OE(CellI)
```

The extra pack in the translation follows from the fact that the return type of the method has type Y, rather than the object type. In order to yield a fresh object as the result of the message send, the fresh state returned by the method must be re-packaged (with the original methods and state type) as an existential value. With this abbreviation it is easy to see that (mycell<=bump)<=get evaluates to 1, again:

```
(mycell<=bump)<=get
    =    (open mycell as [X,(s,m)] in pack [X,(m(s).bump,m)] as OE(CellI))<=get
    =β   (pack [{x:Int},
                ((methfun {x=0}).bump,
                 methfun)]
          as OE(CellI))
          <=get
    =β   (pack [{x:Int},
                (get={x=0}.x, set=fun(n:Int){x=n}, bump={x={x=0}.x+1}}.bump,
                 methfun)]
          as OE(CellI))
          <=get
    =β   (pack [{x:Int},
                ({x={x=0}.x+1},
                 methfun)]
          as OE(CellI))
          <=get
    =β   (pack [{x:Int},
                ({x=1},
                methfun)]
          as OE(CellI))
          <=get
    =β   open
            (pack [{x:Int},
                   ({x=1},
                    methfun)]
             as OE(CellI))
         as [X,(s,m)] in m(s).get
    =β   (methfun({x=1})).get
    =β   {get={x=1}.x, set=fun(n:Int){x=n}, bump={x={x=1}.x+1}}.get
```

```
  =β  {x=1}.x
  =β  1
```

where

```
methfun  ≝  fun(s:{x:Int}) {get=s.x, set=fun(n:Int){x=n}, bump={x=s.x+1}}
```

A few notes on this encoding. Because the message-sending code has to repack the object after the send in the case of `bump`, but not in the case of `get`, message-sending boilerplate must be generated from types, rather than being defined independently of types (as in the other encodings).

This encoding technique is closely related to semantic models of ADT languages. See [MP88] for details. This encoding has also been adopted in [MMH96] in order to represent closures as objects in compilers.

In the simple encoding, the `bump` method has no access to the `set` and `get` methods – it's only passed the state as a parameter. But, as for `OR`, we can also build `mycell` in such a way that `bump` is defined in terms of `get` and `set`. This time, though, we have to do it a little differently. It doesn't help to send `get` and `set` to the whole object, since the result of `set` is a whole object, whereas the `bump` method is supposed to return just an element of the state type. Instead, we build just the set of methods recursively:

```
mycell  ≝  letrec m (s:{x:Int}) : CellI({x:Int}) =
                      let selfmeth = m s in
                      {get = s.x,
                       set = fun(n:Int) {x=n},
                       bump = selfmeth.set (selfmeth.get)}
             in
               pack [{x:Int}, ({x=0},m)]
               as OE(CellI)
           : OE(CellI)
```

Note that this encoding can be refined by using a bounded existential to expose some of the instance variables. (This idea will come back later!)

```
OBE(X,R)  ≝  Some(Y <: R) Y x Y->I(Y)
```

In this encoding we are revealing that the state is a subtype of some "public instance variables interface" `R`, but are not specifying exactly what the type of the state is.

3.3 `ORE`: Recursion and Existentials

The intuition behind the `ORE` encoding is the same as `OE` except that any methods that return new objects do the repacking of internal state themselves, rather than requiring that the sender do it. This eliminates the need for different encodings of `o<=m` depending on the type of `m`.

```
ORE(I)  ≝  Rec(X) Some(Y) Y x (Y->I(X))
```

Notice that the types of methods now are expressed in terms of `X`, the type of the entire object, rather than just the type of the Y component. This will make it easier for us to encode message sends in a more uniform way. Thus a method returning a value of type `X` is returning an object, not just its state component. As we shall see in section 4.6, this also provides support for "binary methods."

For convenience, define:

```
close  ≝  fun(internalObj:{x:Int} x ({x:Int}->CellI(ORE(CellI))))
             pack [{x:Int}, internalObj]
             as ORE(CellI)
```

The function `close` takes a value representing the internal view of the object (in general, of type `Y x (Y->I(X))`) and creates an object of type `ORE(I)` by hiding the type of the state.

Now define `mycell` as:

```
mycell  ≝  letrec methfun(s:{x:Int}) : CellI(ORE(CellI)) =
                      {get = s.x,
                       set = fun(n:Int) close ({x=n},methfun),
                       bump = close ({x=s.x+1},methfun)}
            in
               close ({x=0},methfun)
          : ORE(CellI)
```

The function `methfun` takes a value, `s`, representing the state of an object and creates a record of methods in which each method uses `s` as the current state. The `close` function packages up a new state with the method definitions given by `methfun`, producing a new object of type `ORE(CellI)`. The fact that the methods `set` and `bump` return objects of type `ORE(CellI)` is a key difference from the existential encoding, where these methods simply returned the updated state.

As in `OR`, message sending is interpreted uniformly:

```
o<=l  ≝  open o as [X,(s,m)] in (m(s)).l
```

The expression `(mycell<=bump)<=get` evaluates to 1 as usual.

```
(mycell<=bump)<=get
        =   ((close ({x=0},methfun))<=bump)<=get
        =β  (pack [{x:Int}, ({x=0},methfun)] as ORE(CellI))<=bump)<=get
        =β  ((methfun {x=0}).bump)<=get
        =β  (close ({x={x=0}.x+1},methfun))<=get
        =β  (close ({x=1},methfun))<=get
        =β  (pack [{x:Int}, ({x=1},methfun)] as ORE(CellI)))<=get
        =β  (methfun {x=1}).get
        =β  {x=1}.x
        =β  1
```

The main difference from the previous encoding is the use of `close`, which takes a state and a method suite and builds an object.

We can implement bump in terms of set as follows:

```
mycell  ≝  letrec methfun(s:{x:Int}) : CellI(ORE(CellI)) =
              let self = close (s,methfun) in
```

```
                {get = s.x,
                 set = fun(n:Int) close ({x=n},methfun),
                 bump = self<=set (self<=get + 1)}
          in
             close ({x=0},methfun)
       : ORE(CellI)
```

In this definition `s`, with type {`x:Int`}, represents the state while `self`, with type `ORE(CellI)`, represents an object with that state. As before, the methods `set` and `bump` both return values of type `ORE(CellI)`.

3.4 `ORBE`: Recursion and Bounded Existentials

We can understand the `ORBE` encoding by starting with the `OE` encoding and working our way up to the more complex one. The `OE` encoding makes no public commitment about the type of the state: we can choose the state to be a record of "instance variables" as we have done so far, or as an element of any other type, so long as we can write methods that operate on this state in the appropriate way. In particular, we can choose the state type to be the type of the object itself! This may seem a slightly strange thing to do, but note that it allows us to use the `o<=l` syntax in the definition of `bump`:

```
mycell  ≝  letrec mkobj(s:{x:Int}) : OE(CellI) =
            let self = mkobj s in
              pack [OE(CellI),
                (self,
                 fun(self':OE(CellI))
                   {get = s.x,
                    set = fun(n:Int) mkobj {x=n},
                    bump = self'<=set (self'<=get + 1)
                   }
                )]
              as OE(CellI)
           in mkobj {x=0}
```

It would be nice if we could use the more uniform encoding of message sending in `OR` and `ORE`. We can do this if we add a recursive definition of `X` while revealing only some of the information about the actual type of the object. Define

```
ORBE(I)  ≝  Rec(X) Some(Y <: X) Y x (Y->I(Y))
```

In the implementation of `mycell`, `Y` will be the actual type `ORBE(I)` of the entire object, but we do not reveal this publicly. We can now define an object as follows:

```
mycell  ≝  letrec mkobj(s:{x:Int}) : ORBE(CellI) =
             let self = mkobj s in
               pack [ORBE(CellI),
                 (self,
                  fun(self':ORBE(CellI))
```

```
                              {get = s.x,
                               set = fun(n:Int) mkobj {x=n},
                               bump = self'<=set (self'<=get + 1)
                              }
                             )]
                          as ORBE(CellI)
                     in
                        mkobj({x=0})
          : ORBE(CellI)
```

With this extra encoding we can now define message sends uniformly (with the same definition as in the ORE encoding):

$$\texttt{o<=l} \stackrel{\text{def}}{=} \texttt{open o as [X,(s,m)] in (m(s)).l}$$

Note that the assumption `Y <: X` on `Y` is critically used in the typing of `o<=bump`: the body `(m(s)).l` has minimal type `Y`, but in order to satisfy the side condition on the open rule for existential types, this has to be promoted to a `Y`-free supertype—i.e. `ORBE(CellI)`. Note that as long as `Y` appears in positive positions, this subsumption still works.

Recall that a disadvantage of the `OE` encoding was that both the `set` and `bump` methods simply returned a value representing a state with type `{x:Int}`. With the `ORBE` encoding, the state has type `ORBE(CellI)`, so these methods return objects.

4 Comparisons

Having presented these four models as encodings in a common notational framework, we are now in a position to begin comparing them along a number of dimensions.

4.1 Treatment of the self parameter

The four encodings represent four strategies for encoding objects. In `OR`, methods do not take an explicit `self` argument on invocation. Instead, `self` is implicitly bound by a recursive declaration when the object is constructed. In the other three encodings, an argument representing `self` is explicitly passed to the methods. In `OE` and `ORE`, the argument is just the "internal state" of the object, while in `ORBE` the argument is the whole object. In `OE`, methods that return a modified version of `self` (such as `bump`), return just the state part, while in `ORE` and `ORBE`, such methods return a whole object. Summarizing, we can say that `self`-returning methods in `OR` map unit to whole objects, in `OE` they map states to states, in `ORE` they map states to whole objects, and in `ORBE` they map whole objects to whole objects.

4.2 Same information, different packaging

The four encodings "represent the same kind of objects," in the sense that an object in one of the encodings can be wrapped up into an object in any other encoding that reacts to messages in exactly the same way as the original. In two cases, the "wrapping procedure" is actually trivial:

```
ORBE(I) <: OE(I)
ORBE(I) <: ORE(I)
```

This shows that `ORBE` is the most revealing of the three encodings involving existential types, in the sense that `OE` and `ORE` can be viewed as variants of `ORBE` that make fewer public commitments about their implementation.

4.3 Full abstraction

A more subtle—and arguably less important—difference between the `OR` encoding and the encodings based on existentials is that, in the latter three, an "observing context" can perform operations on an object that do not correspond to sending messages and, in some cases, obtain some information about the internal implementation of the methods. With the `OR` encoding, the only test that an observing context can make of an object is to look at the results that are returned by its methods. In the existential encodings, the observer can also apply the methods to a divergent argument, giving it the power to discriminate between objects that cannot be told apart just by sending messages in the ordinary way. This represents a kind of failure of full abstraction for the existential encodings.

4.4 Uniform vs. type-directed message sending

One difference between the encodings is whether or not message passing can be encoded uniformly. In `ORE` and `ORBE`, every message is sent by opening the packed object, applying the second (method) component to the first (state) component, and then extracting the appropriate field of the result. It is even easier in `OR`, since no existential unpacking is needed.

In `OE`, on the other hand, the encoding of message sending depends on the type of the method. If there is no occurrence of the "self type variable" (the bound variable of the type operator representing the interface signature) in the result type, then message sends are encoded as for `ORE` and `ORBE`. However if the return type is the self type variable, then the result of the method must be repackaged as a new existential value (of type `OE(I)`). For more complicated result types involving the self type variable, the appropriate repackaging code can be generated by induction on the method type.

4.5 Strength of underlying type theory

`OE` works in the "most elementary" type theory — $F^{\omega}_{<:}$ with the Kernel Fun subtyping rule [CW85]. If classes and inheritance are omitted, the underlying calculus is even strongly normalizing.

All the other models require recursive types, which entail recursion and loss of strong normalization. `OE` itself requires recursion (on values) to handle classes. Moreover, all the models use recursive values when adapted to allow method invocation through self. In the presence of recursive values, the semantics of the type system becomes more challenging; recursive types also complicate the metatheory.

`OR`, `OE`, and `ORE` work fine with the Kernel Fun subtyping rule for quantifiers. `ORBE` requires the full $F_{<:}$ rule, leading to a substantial increase in the theoretical complexity of the calculus [Ghe95, Ghe93] and the loss of some pragmatically desirable properties such as decidability [Pie94]. See [PS97] for more discussion of variants of this rule.

Karl Crary has observed [personal communication] that it may be possible to ameliorate this deficiency in `ORBE` by introducing a single type constructor combining the behaviors of `Rec` and `Some`.

4.6 Binary Methods

Another difference between the encodings concerns the treatment of *binary methods*—methods taking an argument of the same type as the receiver object. Consider the following object interfaces:

```
CellI(X)      = {get:Int, set:Int->X, bump:X}
EqCellI(X)    = {get:Int, set:Int->X, bump:X, eq:X->Bool}
EqClrCellI(X) = {get:Int, set:Int->X, bump:X, eq:X->Bool, color:Color}
```

`CellI` is our running example of cells; `EqCellI` adds a method `eq` that takes a cell and compares its contents with the contents of the cell to which the `eq` message is sent; `EqClrCell` adds one more method (whose behavior is unimportant). The crucial difference between `CellI` and the other two operators is that `CellI` is *covariant*—that is, `S <: T` implies `CellI(S) <: CellI(T)`—which is not the case for `EqCellI` or `EqClrCellI`, which both contain occurrences of the bound variable `X` in *contravariant* positions. This section and the next explore the consequences of non-covariant operators as object signatures.

Unfortunately, neither the `OE` nor the `ORBE` encoding handles non-covariant interfaces satisfactorily. For example, consider the object type `OE(EqCellI)`—simple existential cell objects with equality methods:

```
EqCell  ≝  OE(EqCellI)
        =  Some(Y) Y x (Y -> {get:Int, set:Int->Y, bump:Y, eq:Y->Bool})
```

We can create objects with this type exactly as we did in section 3.2.

```
myeqcell  ≝  pack [{x:Int},
                    ( {x=0},
                      fun(s:{x:Int})
                        {get = s.x,
                         set = fun(n:Int) {x=n},
                         bump = {x=s.x+1},
                         eq = fun(s':{x:Int}) s.x = s'.x })]
               as OE(CellI)
            : OE(CellI)
```

However, it is not possible to *send* `eq` messages to such objects in the way we would expect. Having unpacked the existential, applied the methods to the state, and projected the `eq` field of the resulting record, we are left with a function that expects a parameter of the same state type. But the second cell object that we want to pass as argument has its own—possibly different—internal state type, so its internal state is not an appropriate argument. The same observation applies to `ORBE(EqCellI)` (even though the state type is partially known).

This defect can be repaired partially by manually introducing a recursion in the interface signatures, binding the contravariant occurrences of the "self variable," and adding explicit object constructors:

```
REqCellI      ≝ Rec(J) Fun(X) {get:Int, set:Int->X, bump:X, eq:OE(J)->Bool}
REqClrCellI   ≝ Rec(J) Fun(X) {get:Int, set:Int->X, bump:X, eq:OE(J)->Bool,
                               color:Color}
```

This step allows binary messages to be sent to objects, but involves a nontrivial extension to the ambient type theory, since it relies on a recursively defined type operator. Moreover, it destroys the important property of pointwise subtyping between interfaces: `REqClrCellI` is not a subtype of `REqCellI` (whereas `EqClrCellI` *is* a subtype of `EqCellI`).

On the other hand, with `OR` and `ORE` we can create objects with interfaces like `EqCellI` and `EqClrCellI` and send them messages exactly as before: support for binary methods is "built in." In a sense, the `ORE` encoding is derived from `OE` precisely by adding support for binary methods. (A similar extension can be made to `ORBE`, yielding a hybrid "mother of all encodings" that may be worth further investigation.)

Furthermore, since `EqClrCellI` is (pointwise) a subtype of `EqCellI`, we can write functions that manipulate both cells and colored cells with equality, by abstracting over subtypes of `EqCellI`:

```
test5  ≝  fun(I<EqCellI)
            fun(o:OR(I))
              if o <= eq (o <= set(5)) then "o contains 5" else "doesn't"
```

4.7 Subtyping of Object Types

Unfortunately, the previous example would not work if we simply wrote

```
test5'  ≝  fun(C<OR(EqCellI))
             fun(o:C)
               if o <= eq (o <= set(5)) then "o contains 5" else "doesn't"
```

using an abstraction over types bounded by the object type `OR(EqCellI)` instead of the abstraction over type operators `I` bounded by `EqCellI`. While this simpler version is well typed as it stands, it is not very useful because `OR(EqCellI)` does not have any nontrivial subtypes!

In general, the pointwise subtyping relation `I <: J` between object interfaces does not imply that the corresponding object types `OR(I)` and `OR(J)` are in the subtype relation. (Nor, similarly, does it follow that `ORE(I) <: ORE(J)`.) On the other hand, it does always follow that `OE(I) <: OE(J)` and `ORBE(I) <: ORBE(J)`. The built-in support for binary methods in `OR` and `ORE` comes at the price of subtyping between object types in some cases. (In particular, it will only be the case that `I <: J` implies `OR(I) <: OR(J)` and `ORE(I) <: ORE(J)` when `J` is covariant.)

This may or may not be viewed as a serious problem, since we can always write functions in the form of `test5` instead of `test5'`. Indeed, this style of "polymorphic programming by bounded abstraction over interfaces" has been proposed in several languages under the name *matching* [BSvG95, AC95].

4.8 Method override

Method update can be incorporated in the `ORBE` encoding by extending the encoding of an object with a collection of method updaters. Roughly, for a method of type `X->B`, where `X` is the type of self and `B` is the result type, a method updater has type `(X->B)->X`. That is, an updater takes a

	`OR`	`OE`	`ORE`	`ORBE`
message-sending	uniform	type-directed	uniform	uniform
internal access to "self methods"	built-in	can be added	built-in	built-in
"fully abstract"	yes	no	no	no
ambient type theory	$F^{\omega}_{<:}$ + `Rec`	pure $F^{\omega}_{<:}$	$F^{\omega}_{<:}$ + `Rec`	$F^{\omega}_{<:}$ + `Rec`
quantifier subtyping	Kernel Fun	Kernel Fun	Kernel Fun	full $F_{<:}$
`I <: J` ⇒ `O(I) <: O(J)`?	if `J` is covariant	yes	if `J` is covariant	yes
support for non-covariant interfaces	yes	limited, using extra `Rec`	yes	limited, using extra `Rec`
method override	no	no	no	in a variant

Table 1: Summary of comparisons

new method and returns an object with the new method in it. In the case of a method returning self, the type of new method must be sufficiently polymorphic. This requires some changes to the `ORBE` encoding given in Section 3.4 (each method must be abstracted on the "self parameter" individually, rather than abstracting the whole record of methods on the self parameter) and the adoption of special typing and subtyping rules, called structural rules, in the underlying type theory. See [ACV96] for details.

The technique of method updaters used for `ORBE` cannot be easily adapted to the other models. Method updaters do not work well for `OE` and `ORE` because the new methods there would have to be too polymorphic (since the representation of the state is completely hidden) and could not do much useful work. Likewise, method updaters do not work for `OR`, because in general they involve contravariant occurrences of the recursive object type, and then object subtyping fails.

5 Conclusions

Table 1 summarizes the major points of comparison between the four encodings we have considered. Interestingly, none of the columns completely dominates all of the others. However, we can make some broad comparisons.

There are two basic encoding techniques and two hybrids. The principal advantage of the basic techniques is straightforward intuition: `OR` represents the most naive view of objects as data values that can be interrogated by named messages; `OE` gives a lower-level picture, showing explicitly that objects consist of state and methods, with the state inaccessible except via the methods. The hybrid encodings—both of which can be viewed as generalizations of `OE`—are more powerful, each offering a different useful refinement: `ORE` adds support for binary methods; `ORBE` can be extended to support method override.

6 Acknowledgments

Peter O'Hearn and Ramesh Viswanathan joined us in early discussions of the material presented here. O'Hearn was particularly helpful in understanding the circumstances under which "full abstraction" of the encodings will fail (cf. Section 4.3). Viswanathan contributed valuable insight into method overriding.

References

[AC93] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[AC95] Martín Abadi and Luca Cardelli. On subtyping and matching. In *Proceedings ECOOP '95*, pages 145–167, 1995.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[ACV96] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Principles of Programming Languages*, pages 396–409, 1996.

[BCC+96] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. A preliminary version appeared in POPL 1993 under the title "Safe Type Checking in a Statically Typed Object-Oriented Programming Language".

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *ECOOP '95*, pages 27–51. LNCS 952, Springer-Verlag, 1995. A complete version of this paper with full proofs is available via http://www.cs.williams.edu/~kim/.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.

[Car90] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[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.

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135, January 1990.

[Coo89] William Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[CP89] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 433–444, 1989.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[Ghe93] Giorgio Ghelli. Recursive types are not conservative over $F_{\leq}$. In *Typed Lambda Calculus and Applications*, March 1993.

[Ghe95] Giorgio Ghelli. Divergence of $F_{\leq}$ type checking. *Theoretical Computer Science*, 139(1,2):131–162, 1995.

[HP95] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[KR94] Samuel N. Kamin and Uday S. Reddy. Two semantic models of object-oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. The MIT Press, 1994.

[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. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[MMH96] Yasuhiko Minamide, Greg Morrisett, and R. Harper. Typed closure conversion. In *Proc. 23rd ACM Symp. on Principles of Programming Languages*, pages 271–283, January 1996.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.

[Pie94] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994). A preliminary version appeared in POPL '92.

[PS97] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 1997. To appear. A preliminary version appeared in IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET), June 1994, and as University of Edinburgh technical report ECS-LFCS-94-280 and Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94, January 1994.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of functional programming*, 4:207–247, 1994. An earlier version appeared in Proc. of POPL '93, pp. 299-312.

[Red88] Uday S. Reddy. Objects as closures: Abstract semantics of object oriented languages. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 289–297, Snowbird, Utah, July 1988.