

An Object Encoding For SelfType

Steven E. Ganz*
Indiana University

Daniel P. Friedman†
Indiana University

Abstract

SelfType is a programming language feature of object systems which allows both methods and instance variable declarations to refer to the type of `self`. We present an object encoding sufficient to model **SelfType** while maintaining explicit support for instance variables or other private interfaces. The range of the encoding is F_{λ}^{ω} , the typed λ -calculus including polymorphic functions and types, intersection types and subtyping, augmented with recursive types (of kind \star). Intersection types are used to support vertical specialization of methods and multiple inheritance.

1 Introduction

In this paper we present an object encoding supporting **SelfType**. In particular, we allow the use of **SelfType** in instance variable declarations. We first describe the notions of **SelfType** and of object encodings. We then present our encoding in three stages.

1.1 SelfType

Inheritance in object-oriented languages allows code written for the superclass to be reused by the subclass, with dynamic references to `self` helping to increase its relevance to the subclass. Self-references in the superclass are redirected to refer to the subclass object. In addition to self-reference at the level of objects, one might allow self-reference at the level of types. Declaring parameters and return values of methods and instance variables to be of **SelfType** causes them to be considered, from the perspective of a derived class, to be declared as the type of that derived class, not in general the type of the class currently being constructed. This feature dates back to the Eiffel programming language. Bruce describes its use in the prevention of the loss of type information when `self` is returned by a method and in the implementation of binary methods [4]. Bruce, et. al., provide this feature in their PolyTOIL language (derived from TOOPLE [6]) and present an example of these uses as well as the creation of specializable recursively-defined data structures [3]. We next present a functional variant of the latter example in a made-up object-oriented language.

Objects of class `Node` contain two instance variables, one a natural number and the other of **SelfType**. Nodes contain two methods. The `getVal` method dereferences the appropriate instance variable. The `argOrSelfOrNext` method accepts an argument of **SelfType** and a selector and returns either the node argument, the current node, or its next pointer, depending on the selector value. Doubly-linked nodes (of class `DNode`) add a single instance variable of **SelfType** and an access method.

```
Class Node specializes ()
  ivars = { val: Nat,
            next: SelfType },
  methods = { getVal = lambda () val,
              argOrSelfOrNext: =
                lambda ([arg SelfType] [sel Nat])
                  case sel of
                    [0 arg]
                    [1 self]
                    [2 next]};
```

*This work was supported in part by the National Science Foundation under grant CDA-9312614. Author's address: Department of Computer Science, Indiana University, Bloomington, Indiana 47405. sganz@cs.indiana.edu.

†This work was supported in part by the National Science Foundation under grant CCR-9633109. Author's address: Department of Computer Science, Indiana University, Bloomington, Indiana 47405. dfried@cs.indiana.edu.

```

Class DNode specializes (Node)
  ivars = { prev: SelfType },
  methods = { getPrev = lambda () prev };

```

Now, our test. We will see that although `argOrSelfOrNext` was defined in the superclass `Node`, it is able to both accept and return elements of the derived class `DNode`, upon which we may use the new `getPrev` method.

First we define some test nodes. We assume that our `new` operator takes as arguments all instance variables of the class, ordered by a post-order traversal of a depth-first spanning tree of the hierarchy. The object `a3rdDNode` holds the value 3 and points in both directions to `a2ndDNode`, which holds the value 2 and points in both directions to `aDNode`, which holds the value 1 and contains two `nil` pointers.

```

aDNode = new DNode 1, nil, nil;
a2ndDNode = new DNode 2, aDNode, aDNode;
a3rdDNode = new DNode 3, a2ndDNode, a2ndDNode;

```

Given the selector 0, we return the argument, `a2ndDNode`, whose predecessor contains 1.

```
((a3rdDNode.argOrSelfOrNext(a2ndDNode, 0)).getPrev()).getVal() => 1
```

Given the selector 1, return `self`, `a3rdDNode`, whose predecessor contains 2.

```
((a3rdDNode.argOrSelfOrNext(a2ndDNode, 1)).getPrev()).getVal() => 2
```

Given the selector 2, return the successor, `a2ndDNode`, whose predecessor contains 1.

```
((a3rdDNode.argOrSelfOrNext(a2ndDNode, 2)).getPrev()).getVal() => 1
```

Providing a simple node as an argument to `argOrSelfOrNext` on a doubly-linked node yields a type error.

```
(a3rdDNode.argOrSelfOrNext(a2ndNode 0)).getVal() => type error
```

Most statically type-checked object-oriented languages do not provide this degree of controlled reuse. `SelfType` allows great flexibility in interpreting method signatures under subclassing.

1.2 Object Encodings

The above work, however, was presented in terms of a denotational semantics. The use of encodings of object systems into typed λ -calculi was pioneered by Cardelli, Pierce and Turner [8, 17]. High-level object-oriented languages implicitly include some object system that defines the basic structure and interactions of objects and classes. An encoding of an object system into a typed λ -calculus is a way of giving a two-level semantics to such languages. At the higher level, a set of object-oriented support routines are developed within the calculus. Support routines include high-level operators for defining object and class types from interfaces and functions for instantiating and extending classes. At the lower level, the specific rules by which an object-oriented language operates are defined by the way in which these support routines are used by translated programs. As in most previous work in the area, we will provide examples using the support routines directly, leaving the details of the translation from a higher-level language unspecified. An important advantage of the object-encoding methodology is that it demonstrates an upper bound on the complexity of language features in terms of the standard typing systems that are required for their encoding.

The recursive record encoding of objects (OR) is based on ideas introduced by Cook and Palsberg [11]. It treats classes as generators of records of methods, and objects as their fixed points. The encoding presented here is largely based on the recursive record encoding. Pierce and Turner present an encoding of objects using existential types [15] instead of recursive types (OE), demonstrating that the essentials of object-oriented programming do not require recursive types [17]. Their use of type operators with subtyping is similar to the F-bounded polymorphism suggested by Canning, et. al. [7]. This paper closely follows the techniques introduced by Pierce and Turner. Bruce, Cardelli and Pierce [5] standardize the above two object encodings and present two more: a type-theoretic analogue of Bruce's denotational semantics using both recursive and existential types (ORE)¹, and a simplification of Abadi, Cardelli and Viswanathan's encoding [2] of the "calculus of primitive objects" [1] using recursive and bounded existential types [9] (ORBE). The ORBE encoding takes the significant step of assuming that the type of an object's state is a subtype of the object itself. Thus it merges the concepts of a record of methods and an object state, with updates to state treated functionally by returning revised methods. Crary's OREI encoding [12] is even more thorough, using intersection types along with existential types to guarantee that an object's state is precisely the object itself. We believe that there is some merit in distinguishing an object's public and private interfaces and proceed on that basis. Both Bruce, et. al., and Crary provide only object types for the four encodings, and omit the class encodings provided by Pierce and Turner for OE. Our presentation here of a full encoding supporting `SelfType` can be seen as a continuation of the project begun in that work.

¹The ORE encoding was developed to model some aspects of Bruce's work. It does not model his use of `SelfType` in defining instance variables.

1.3 Deficiencies of Current Encodings in Dealing with SelfType

The OR and ORE encodings can handle only approximations of `SelfType`. Bruce, Cardelli and Pierce describe how a function parameterized over an interface and then an object type can simulate binary methods [5]. In the context of our above example, we can define:

```
f =
  λ I <: NodeI.
    λ o : ORE I.
      o <- argOrSelfOrNext o 0 ;
```

Here, `<-` corresponds to message-sending. We can then call `f DNodeI aDNode` if we know that `aDNode : ORE DNodeI`, and `argOrSelfOrNext` will be expected to accept and return an `ORE DNodeI`. Similar techniques should work for OREI, although it is not defined in the context of bounded quantifiers.

While these encodings might allow an instance variable to hold objects of any subclass of the class currently being defined, they do not provide the ability to declare instance variables as belonging to `SelfType`. Thus, although we have achieved the ability to have a method return an object of a known subclass within a defined scope, this victory is hollow since we cannot define the body of the method as an instance-variable reference without getting a type error. This criticism is somewhat unfair for OREI, which has dismissed with instance variables all-together, but we feel that there is benefit in an encoding which allows for type self-reference at distinct implementation levels.

The OE and ORBE encodings have the advantage of monotonicity (so the above “trick” is not needed), but can gain type self-reference only by giving up pointwise subtyping between interfaces (by wrapping a recursive-type declaration around them).

2 An Object Encoding For SelfType

The calculus that we will require is F_{λ}^{ω} with first-order recursive types. This system is the polymorphic λ -calculus, extended with type operators, subtyping and finite intersections, and further extended with recursive types (of kind \star). We also assume the presence of local bindings, record types, bounded quantifiers and products, as these can be implemented from within the calculus. We assume familiarity with these standard features of type systems. Our approach is meant to be intuitive — we present this calculus as a pedagogical tool for explaining object systems. We make no claims of it here beyond that our implementation is sufficient to typecheck the examples presented in this paper. Syntax for the calculus is presented in Figure 1. Below, we omit kind annotations for bounded polymorphic types, bounded type abstractions, intersections and products.

A difficulty of the object-encoding approach is in handling state, a rather central feature of object-oriented systems. One approach, taken by Pierce and Turner [17], is to model side effects by the use of “extractors” that perform functional get and set operations on a record that may be a proper subtype of the current representation. Pierce also describes a solution using reference types and a weakened call-by-value recursion operator [16]. Another possibility is to model state directly in a λ -calculus enhanced with labels. Work was done in this direction using an untyped λ -calculus [13]. We choose to view the issue as a distraction and work with functional examples in the comfort that state could be added by any of these means.

2.1 Basics

We will be using the example from the introduction. We begin by presenting public and private interfaces. The attributes of both may include `SelfType`, so both are defined not as first-order record types but as operators over record types. For example, we define both interfaces of nodes as follows:

```
NodePublic =
  λ SelfType.
    {getVal : Unit → Nat ,
     getValIndirect : Unit → Nat ,
     argOrSelfOrNext : SelfType → Nat → SelfType } ;
DeltaNodePrivate = λ SelfType. {#val : Nat , #next : SelfType } ;
NodePrivate = DeltaNodePrivate;
```

We thus declare that the public interface of nodes contains three methods, two returning a natural and one accepting a node followed by a natural and returning a node. The reason for including `getValIndirect` will become apparent shortly. The private interface of nodes contains a natural and a node. It is specified via a “Delta” operator that gives the portion of the private interface local to the class being defined.

Next we seek a typing in the calculus of objects and their representations. Both must be in terms of `SelfType`. The type of a representation of an object is an application of a private interface to `SelfType`, while the type of an object is an application of a public interface to `SelfType`. In both cases, `SelfType` will be defined as a fixed point of a yet to be determined public interface, `FinalSelfPublic`. Thus, we have:

Figure 1: Kind, Type and Term Syntax for the Calculus

$K ::= \star$		types
$K \rightarrow K$		type operators
$T ::= X$		variable
Unit, Bool, Nat		base types
\perp		bottom type
$T \rightarrow T$		function type
$\forall X \leq T :: K. T$		bounded polymorphic type
$\forall X :: K. T$	$\doteq \forall X \leq \top^K :: K. T$	unbounded polymorphic type
$\forall X. T$	$\doteq \forall X :: \star. T$	unbounded polymorphic type at kind \star
$\lambda X :: K. T$		operator abstraction
$T T$		operator application
$\{l : T \dots l : T\}$		record type
Rec $X.T$		recursive type
$\bigwedge^K [T \dots T]$		intersection at kind K
\top^K	$\doteq \bigwedge^K []$	top at kind K
$T \wedge^K T$	$\doteq \bigwedge^K [T, T]$	binary intersection at kind K
$T \times^K T$		cross product at kind K
$e ::= x$		variable
unit, true, false, n		base terms
nil		bottom term
if e then else		conditional
iszeroe		zero test
prede		predecessor
$\lambda x : T. e$		abstraction
$e e$		application
let $x = e$ in e		local definition
let $X = T$ in e		local type definition
$\lambda X \leq T :: K. e$		bounded type abstraction
$\lambda X :: K. e$	$\doteq \lambda X \leq \top^K :: K. e$	unbounded type abstraction
$\lambda X. e$	$\doteq \lambda X :: \star. e$	unbounded type abstraction at kind \star
$e T$		type application
$\{l = e \dots l = e\}$		record construction
$e.l$		field selection
fix e		recursive expression
$\langle e, e \rangle$		pair
$e.1$		left projection
$e.2$		right projection

```
Representation =
  λ Private :: * → *.
  λ FinalSelfPublic :: * → *.
  Private (Rec X. FinalSelfPublic X) ;
```

```
Object =
  λ Public :: * → *.
  λ FinalSelfPublic :: * → *.
  Public (Rec X. FinalSelfPublic X) ;
```

Objects are derived from representations. Another type operator expresses this relationship.

```
Implementation =
  λ Public :: * → *.
  λ Private :: * → *.
  λ FinalSelfPublic :: * → *.
  Representation Private FinalSelfPublic → ;
  Object Public FinalSelfPublic
```

The similarity between representation types and object types is quite intentional and the above definitions indicate the existence of an implementation hierarchy. Thus, we allow a representation to consist of arbitrary private methods which in another context might be considered an object. Our examples, however, will use traditional representations consisting of instance variables.

The use of an implementation to create an object lexically hides the representation within the object — the object is thus abstract, as in object-oriented languages. An alternative approach used by Pierce and Turner in providing this capability is to pass around structures containing both representations and implementations as objects and build methods only upon method invocation, using existential types for object encapsulation. We choose to enjoy the simplicity of our approach. However, we believe that an encoding analogous to this one could be developed with existential types.

In both cases, though, objects hide more than they do in most object-oriented languages, whose notion of encapsulation is class-based and not object-based [18]. In other words, it is possible in most object-oriented languages to see from within an object’s methods the representation of another object belonging to the same class. A notion of class-based control over representations could be implemented (at runtime) by giving each class a key value and having objects provide instead of a representation, a function from the set of possible key values to representations. We treat this subtlety as inconsequential.

Within the calculus, classes are defined as expressions whose types determine the typing of the methods they provide.² Classes are defined as functions from representations to implementation generators. We next present a type operator describing class types.

```
Class =
  λ Public :: * → *.
  λ Private :: * → *.
  λ FinalSelfPublic :: * → *.
  Representation Private FinalSelfPublic → ;
  Implementation Public Private FinalSelfPublic →
  Implementation Public Private FinalSelfPublic
```

Our encoding differs from others in that classes do not hold an initial object representation. Instead, the representation is provided on object construction. Allowing both forms of initialization is overly complex for our purposes. Without side effects, allowing only class-based initialization would require a separate class for every possible object representation. This is inconsistent with existing object-oriented languages. In any case, we find it awkward to force all objects of a class to begin with the same representation, as there are not always appropriate defaults (although a nil value of type \perp could be used for this purpose) and we do not believe that initialization is an appropriate use for side effects. Notice that since we are not implementing state there is no necessity for each object to have its own representation at all, and we could make do with having each class hold the primary copy of the representation. However, we wish to make our encoding “state-friendly” by allowing for the possibility of updatable representations. The hash marks on our record structures for representations mark fields as mutable and thus treated invariantly under subtyping.

For class creation, we can use a function `extend0`³. This will require method descriptions from the user of the routine, which we type as `MethDesc0`. This type defines a record of methods conforming to the public interface in terms of the local representation and a self object. The self object is provided as a thunk to allow for applicative-order evaluation.

²We make no commitment about the status of classes in the high-level language from which we may have translated.

³0 is the number of parents.

```

MethDesc0 =
  λ Public :: * → *.
  λ DeltaPrivate :: * → *.
  λ FinalSelfPublic :: * → *.
  Representation DeltaPrivate FinalSelfPublic → ;
  (Unit → Object Public FinalSelfPublic) →
  Object Public FinalSelfPublic

extend0 =
  λ Public :: * → *.
  λ DeltaPrivate :: * → *.
  λ FinalSelfPublic <: Public.
  λ methDesc : MethDesc0 Public DeltaPrivateFinalSelfPublic.
  let Private = DeltaPrivate in
  λ selfRepr : Representation Private FinalSelfPublic. ;
  λ selfImpl : Implementation Public Private FinalSelfPublic.
  λ repr : Representation Private FinalSelfPublic.
  methDesc repr ( λ u : Unit. selfImpl selfRepr)

extend0 :
  ∀ Public :: * → *.
  ∀ DeltaPrivate :: * → *.
  ∀ FinalSelfPublic <: Public.
  MethDesc0 Public DeltaPrivate FinalSelfPublic →
  Class Public DeltaPrivate FinalSelfPublic

```

From such method descriptions, `extend0` returns a class which for a given representation and implementation, returns an implementation which will, given a representation, apply the method descriptions to that representation (since with no parents it is entirely local) and to a self object derived from the given implementation and representation. Returning to our node example, we demonstrate the use of `extend0` by generating a node class in terms of the final public interface:

```

closeNodeClass =
  λ FinalSelfPublic <: NodePublic.
  extend0
  NodePublic DeltaNodePrivate FinalSelfPublic
  (λ deltaRepr : Representation DeltaNodePrivate FinalSelfPublic.
  λ selfObjTh : Unit → Object NodePublic FinalSelfPublic.
  {getVal = λ u : Unit. deltaRepr.val ,
  getValIndirect = λ u : Unit. (selfObjTh u).getVal u ,
  argOrSelfOrNext =
    λ arg : Object NodePublic FinalSelfPublic. ;
    λ sel : Nat.
    if iszero sel
    then arg
    if iszero(predsel)
    else then selfObjThunit
    else deltaRepr.next
  }

closeNodeClass :
  ∀ FinalSelfPublic <: NodePublic.
  Class NodePublic NodePrivate FinalSelfPublic

aNodeClass = closeNodeClass NodePublic;
aNodeClass : Class NodePublic NodePrivate NodePublic

```

The `getVal` method returns the natural number from the representation. The `getValIndirect` method calls `getVal` through the self object. The `argOrSelfOrNext` method returns either the node argument, the current node, or its next pointer, based on the selector value⁴. We “close” the class by specifying that `NodePublic` is to be used to type self-references in both objects and representations.

The new function generates an object of a given representation from a class. It applies the class to the representation, takes the fixed point of the implementation generator, and then “closes” the object by providing the representation.

⁴We avoid creating objects of `SelfType` in our example since we do not know the final representation required. Where necessary, this can be handled through the factory method pattern [14].

```

new =
  λ Public :: * → *.
  λ Private :: * → *.
  λ class : Class Public Private Public.
  λ repr : Representation Private Public.
  fix (class repr) repr

new :
  ∀ Public :: * → *.
  ∀ Private :: * → *.
  Class Public Private Public →
  Representation Private Public →
  Object Public Public

```

We can generate a few nodes and test our methods as follows:

```

aNode = new NodePublic NodePrivate aNodeClass {#val = 1 , #next = nil };
aNode : Object NodePublic NodePublic

a2ndNode = new NodePublic NodePrivate aNodeClass {#val = 2 , #next = aNode };
a2ndNode : Object NodePublic NodePublic

a2ndNode.getVal unit;
2 : Nat

a2ndNode.getValIndirect unit;
2 : Nat

```

Given the selector 0, we return the arg, `aNode`, which contains 1.

```

(a2ndNode.arg0rSelf0rNext aNode 0).getVal unit;
1 : Nat

```

Given the selector 1, we return self, `a2ndNode`, which contains 2.

```

(a2ndNode.arg0rSelf0rNext aNode 1).getVal unit;
2 : Nat

```

Given the selector 2, we return the successor, `aNode`, which contains 1.

```

(a2ndNode.arg0rSelf0rNext aNode 2).getVal unit;
1 : Nat

```

2.2 Single Inheritance

For single inheritance, we introduce a natural extension of `MethDesc0` and `extend0`.

```

MethDesc1 =
  λ SuperPublic :: * → *.
  λ Public :: * → *.
  λ DeltaPrivate :: * → *.
  λ FinalSelfPublic :: * → *.
  Representation DeltaPrivate FinalSelfPublic →
  (Unit → Object Public FinalSelfPublic) →
  (Unit → Object SuperPublic FinalSelfPublic) →
  Object Public FinalSelfPublic

```

```

extend1 =
  λ SuperPublic :: * → *.
  λ Public <: SuperPublic.
  λ SuperPrivate :: * → *.
  λ DeltaPrivate :: * → *.
  λ FinalSelfPublic <: Public.
  λ superClass : Class SuperPublic SuperPrivateFinalSelfPublic.
  λ methDesc : MethDesc1 SuperPublic Public DeltaPrivateFinalSelfPublic.
  let Private = SuperPrivate × DeltaPrivate in
  λ selfRepr : Representation Private FinalSelfPublic.
  λ selfImpl : Implementation Public Private FinalSelfPublic.
  λ repr : Representation Private FinalSelfPublic.
  methDesc repr.2
  (λ u : Unit. selfImpl selfRepr)
  (λ u : Unit.
    superClass
    repr.1
    (λ superRepr : Representation SuperPrivate FinalSelfPublic.
      (selfImpl < superRepr, repr.2 >) )
    selfRepr.1)
;

extend1 :
  ∀ SuperPublic :: * → *.
  ∀ Public <: SuperPublic.
  ∀ SuperPrivate :: * → *.
  ∀ DeltaPrivate :: * → *.
  ∀ FinalSelfPublic <: Public.
  Class SuperPublic SuperPrivate FinalSelfPublic →
  MethDesc1 SuperPublic Public DeltaPrivate FinalSelfPublic →
  Class Public (SuperPrivate × DeltaPrivate) FinalSelfPublic

```

Because the user of the routine may refer to the super object within the methods being defined, the `methDesc` function now takes such an object. This requires that `MethDesc1` take the superclass public interface.⁵

The `extend` function now requires a superclass. We assume that the new public interface is a subtype of that of the superclass. Thus, any redefinitions provided in the new interface may only constrain the specified types. We assume that the private interface of the new class is a product of the superclass private interface and its own local private interface (`DeltaPrivate`). A product is appropriate because the local representation of a class must be independent of that of its parent. Thus, in calling `methDesc`, the local component must be extracted for the `deltaPrivate` argument, the self object is easily constructed from the given `selfImpl`, while the super object is constructed from the superclass, using `selfImpl` to form the self object visible within superclass methods.

We quickly test that we have correctly implemented virtual functions by deriving `NewNodeClass` from `NodeClass`. `NewNodeClass` uses the same public interfaces as `NodeClass`, and adds no instance variables.

We use intersection types in creating derived public interfaces to allow for vertical as well as horizontal modifications, i.e., we allow the derived class to specialize the types of methods provided by its parent. This is a departure from standard practice which is to use a biased product, giving up vertical modifications and deferring the introduction of intersection types until they are needed for multiple inheritance. This decision is orthogonal to the rest of the development. We use a cross-product to represent the derived private interface to ensure that the information in representations is preserved in subclass objects. It would be a violation of encapsulation to allow a subclass to override private attributes. An intersection (cross-product) of type operators is a type operator which when applied yields the intersection (cross-product) of the application of each component type operator. Thus, intersections and cross-products are pushed to the record types.

```

DeltaNewNodePublic = λ SelfType. {};
NewNodePublic = NodePublic ∧ DeltaNewNodePublic;
DeltaNewNodePrivate = λ SelfType. {};
NewNodePrivate = NodePrivate × DeltaNewNodePrivate;

```

`NewNodeClass` redefines `getVal` to always return 5 regardless of the stored value, while `getValIndirect` and `argOrSelfOrNext` are simply inherited from the super class. Note that the extension must take place relative to `FinalSelfPublic`, i.e., before the base class is “closed”.

⁵ `MethDesc` expects a private interface and representation. It might also have been designed to take a protected interface and representation, allowing for separate encapsulation over external and subclass interfaces.

```

closeNewNodeClass =
  λ FinalSelfPublic <: NewNodePublic.
  extend1
    NodePublic NewNodePublic NodePrivate DeltaNewNodePrivate FinalSelfPublic
    (closeNodeClass FinalSelfPublic)
    (λ deltaRepr : Representation DeltaNewNodePrivate FinalSelfPublic.
      λ selfObjTh : Unit → Object NewNodePublic FinalSelfPublic.
      λ superObjTh : Unit → Object NodePublic FinalSelfPublic.
      {getVal = λ u : Unit. 5 ,
       getValIndirect = (superObjTh unit).getValIndirect ,
       argOrSelfOrNext = (superObjTh unit).argOrSelfOrNext }
    )

```

```

closeNewNodeClass :
  ∀ FinalSelfPublic <: NewNodePublic.
  Class NewNodePublic NewNodePrivate FinalSelfPublic

aNewNodeClass = closeNewNodeClass NewNodePublic;
aNewNodeClass : Class NewNodePublic NewNodePrivate NewNodePublic

```

To create a `NewNode` object, we must specify both components of the representation as a pair — we place the supertype representation before the local representation. While this may appear awkward, we could always, as in the high-level language in the Introduction, accept the arguments in a flat list and rebuild the hierarchical representation automatically, assuming, for example, a postorder traversal of the class hierarchy.

```

aNewNode = new NewNodePublic NewNodePrivate aNewNodeClass {#val = 0 , #next = nil };
aNewNode : Object NewNodePublic NewNodePublic

aNewNode.getVal unit;
5 : Nat

```

```

aNewNode.getValIndirect unit;
5 : Nat

```

Here, we see that when we redefine `getVal` to always return 5 regardless of the stored value, this redefinition is used even in the superclass method `getValIndirect`.

We can now follow Bruce, et. al., by deriving doubly-linked nodes from singly-linked ones. A doubly-linked node has a public interface including an additional method returning the additional link, and a private interface storing the additional link.

```

DeltaDNodePublic = λ SelfType. {getPrev : Unit → SelfType };
DNodePublic = NodePublic ∧ DeltaDNodePublic;
DeltaDNodePrivate = λ SelfType. {#prev : SelfType };
DNodePrivate = NodePrivate × DeltaDNodePrivate;

```

All methods of class `Node` are inherited unchanged by `DNode`. The `getPrev` method is defined to simply return the new link from the representation.

```

closeDNodeClass =
  λ FinalSelfPublic <: DNodePublic.
  extend1
    NodePublic DNodePublic NodePrivate DeltaDNodePrivate FinalSelfPublic
    (closeNodeClass FinalSelfPublic)
    (λ deltaRepr : Representation DeltaDNodePrivate FinalSelfPublic.
      λ selfObjTh : Unit → Object DNodePublic FinalSelfPublic.
      λ superObjTh : Unit → Object NodePublic FinalSelfPublic.
      {getVal = (superObjTh unit).getVal ,
       getValIndirect = (superObjTh unit).getValIndirect ,
       argOrSelfOrNext = (superObjTh unit).argOrSelfOrNext ,
       getPrev = λ u : Unit. deltaRepr.prev }
    )
closeDNodeClass :
  ∀ FinalSelfPublic <: DNodePublic.
  Class DNodePublic DNodePrivate FinalSelfPublic

```

Here is the test from the Introduction:

```

aDNodeClass = closedNodeClass DNodePublic;
aDNodeClass : Class DNodePublic DNodePrivate DNodePublic

aDNode =
  new DNodePublic DNodePrivate aDNodeClass
    < {#val = 1 , #next = nil } , {#prev = nil } >;
aDNode : Object DNodePublic DNodePublic

a2ndDNode =
  new DNodePublic DNodePrivate aDNodeClass
    < {#val = 2 , #next = aDNode } , {#prev = aDNode } >;
a2ndDNode : Object DNodePublic DNodePublic

a3rdDNode =
  new DNodePublic DNodePrivate aDNodeClass
    < {#val = 3 , #next = a2ndDNode } , {#prev = a2ndDNode } >;
a3rdDNode : Object DNodePublic DNodePublic

```

Given the selector 0, we return the arg, `a2ndDNode`, whose predecessor contains 1.

```

((a3rdDNode.argOrSelfOrNext a2ndDNode 0).getPrev unit).getVal unit;
1 : Nat

```

Given the selector 1, we return self, `a3rdDNode`, whose predecessor contains 2.

```

((a3rdDNode.argOrSelfOrNext a2ndDNode 1).getPrev unit).getVal unit;
2 : Nat

```

Given the selector 2, we return the successor, `a2ndDNode`, whose predecessor contains 1.

```

((a3rdDNode.argOrSelfOrNext a2ndDNode 2).getPrev unit).getVal unit;
1 : Nat

```

Providing a simple node as an argument to `argOrSelfOrNext` on a doubly-linked node yields a type error.

```

(a3rdDNode.argOrSelfOrNext a2ndDNode 0).getVal unit;
Error: Invalid operand type.

```

2.3 Multiple Inheritance

We now extend our encoding to support multiple inheritance. This development is straightforward, along the lines of the extension of the OE encoding to support multiple inheritance suggested by Compagnoni and Pierce [10]. The public interface is now restricted to be a subtype of the intersection of those of the superclasses. The left (super) product component is now itself a product of the two superclass representations. The `methDesc` function now accepts two superclass objects, one constructed from each superclass. Similar functions could be generated for any number of superclasses.

```

MethDesc2 =
  λ SuperPublic1 :: * → *.
  λ SuperPublic2 :: * → *.
  λ Public :: * → *.
  λ DeltaPrivate :: * → *.
  λ FinalSelfPublic :: * → *.
  Representation DeltaPrivate FinalSelfPublic →
  (Unit → Object Public FinalSelfPublic) →
  (Unit → Object SuperPublic1 FinalSelfPublic) →
  (Unit → Object SuperPublic2 FinalSelfPublic) →
  Object Public FinalSelfPublic

```

```

extend2 =
  λ SuperPublic1 :: * → *.
  λ SuperPublic2 :: * → *.
  λ Public <: SuperPublic1 ∧ SuperPublic2.
  λ SuperPrivate1 :: * → *.
  λ SuperPrivate2 :: * → *.
  λ DeltaPrivate :: * → *.
  λ FinalSelfPublic <: Public.
  λ superClass1 : Class SuperPublic1 SuperPrivate1FinalSelfPublic.
  λ superClass2 : Class SuperPublic2 SuperPrivate2FinalSelfPublic.
  λ methDesc : MethDesc2 SuperPublic1 SuperPublic2 Public DeltaPrivateFinalSelfPublic.
  let Private = (SuperPrivate1 × SuperPrivate2) × DeltaPrivate in
  λ selfRepr : Representation Private FinalSelfPublic.
  λ selfImpl : Implementation Public Private FinalSelfPublic.
  λ repr : Representation Private FinalSelfPublic.
  methDesc repr.2
  (λ u : Unit. selfImpl selfRepr)
  (λ u : Unit.
    superClass1 repr.1.1
    (λ superRepr1 : Representation SuperPrivate1 FinalSelfPublic.
      (selfImpl << superRepr1, (repr.1).2 > , (repr.2) > )
      selfRepr.1.1)
    (λ u : Unit.
      superClass2 repr.1.2
      (λ superRepr2 : Representation SuperPrivate2 FinalSelfPublic.
        (selfImpl << repr.1.1, superRepr2 > , repr.2 > )
        selfRepr.1.2)

```

```

extend2 :
  ∀ SuperPublic1 :: * → *.
  ∀ SuperPublic2 :: * → *.
  ∀ Public <: SuperPublic1 ∧ SuperPublic2.
  ∀ SuperPrivate1 :: * → *.
  ∀ SuperPrivate2 :: * → *.
  ∀ DeltaPrivate :: * → *.
  ∀ FinalSelfPublic <: Public.
  Class SuperPublic1 SuperPrivate1 FinalSelfPublic →
  Class SuperPublic2 SuperPrivate2 FinalSelfPublic →
  MethDesc2 SuperPublic1 SuperPublic2 Public DeltaPrivate FinalSelfPublic →
  Class Public ((SuperPrivate1 × SuperPrivate2) × DeltaPrivate) FinalSelfPublic

```

Assuming a primitive type `Color`, and a definition of a `ColorClass` with an instance variable of type `Color` and a `getColor` method returning its contents, we can demonstrate the definition of `ColorNode` using multiple inheritance.

```

DeltaColorNodePublic = λ SelfType. {};
ColorNodePublic = (NodePublic ∧ ColorPublic) ∧ DeltaColorNodePublic;
DeltaColorNodePrivate = λ SelfType. {};
ColorNodePrivate = (NodePrivate × ColorPrivate) × DeltaColorNodePrivate;

```

```

closeColorNodeClass =
  λ FinalSelfPublic <: ColorNodePublic.
  extend2
    NodePublic ColorPublic ColorNodePublic NodePrivate ColorPrivate DeltaColorNodePrivate
    FinalSelfPublic
    (closeNodeClass FinalSelfPublic)
    (closeColorClass FinalSelfPublic)
    (λ deltaRepr : Representation DeltaColorNodePrivate FinalSelfPublic.
      λ selfObjTh : Unit → Object ColorNodePublic FinalSelfPublic.
      λ superObjTh1 : Unit → Object NodePublic FinalSelfPublic.
      λ superObjTh2 : Unit → Object ColorPublic FinalSelfPublic.
      {getVal = (superObjTh1 unit).getVal ,
       argOrSelfOrNext = (superObjTh1 unit).argOrSelfOrNext ,
       getPrev = (superObjTh2 unit).argOrSelfOrNext }
    )
;

closeColorNodeClass :
  ∀ FinalSelfPublic <: ColorNodePublic.
  Class ColorNodePublic ColorNodePrivate FinalSelfPublic

```

In a diamond hierarchy, shared methods may be accessed from either super object. These may or may not be equivalent. With the introduction of state, either repeating (tree-based) or virtual (graph-based) classes [19] can be implemented on the object level, the choice determined by the initial representation structure used.

3 Conclusion

We have presented an object encoding supporting `SelfType`. We conclude by comparing our encoding to the others described by Bruce, Cardelli and Pierce [5], on the basis of the criteria they specify. In our encoding, like in OR, methods need not take an explicit `self` argument on invocation and instance variables are lexically protected. Thus, like OR, our encoding does not suffer from the failure of full abstraction caused by allowing methods to be applied to arguments other than the intended `self` parameter. Of course, this would be changed by inclusion of existential types as in ORE.

Like OR, OE and ORE, our encoding works with the weaker kernel $F_{<}$, subtyping rule for quantifiers, which only compares quantifiers with a common bound. The ORBE encoding requires the more general contravariant $F_{<}$ subtyping rule.

We have decided to allow instance variable values to be specified upon instance creation and not upon class creation. This differs from the existing encodings. It is mostly a matter of style, but makes it possible for us to avoid full support of state without having to create numerous classes. We also differ from the existing encodings in parameterizing `Object` by `FinalSelfPublic` as well as `Public`. This allows us to use our intended rule for subtyping applications of type operators [17] in the presence of recursive types. Finally, we differ in treating `Representation` analogously to our treatment of `Object`, and not as a simple type. Thus, we parameterize `Representation` by both `Private` and `FinalSelfPublic`. This allows for the use of `SelfType` in instance variable declarations.

Comparisons related to self-reference are described in the Introduction. Like OR, ORE and OREI, our encoding supports binary methods. Unfortunately, our object type constructor is still non-monotonic (as with those encodings), although we have perhaps made some progress in changing this by parameterizing `Object` over `FinalSelfPublic`. Once `Object` has been fully applied, however, the resulting types do not preserve subtyping. Some variant of existential types might be useful here.

Unlike any of the existing encodings, we treat representations via interfaces. Also unlike any of the existing encodings, we have decided to allow instance variable values to be specified upon instance creation and not upon class creation. The former decision is fundamental, the latter is mostly a matter of style, but makes it possible for us to avoid full support of state without having to create numerous classes.

Acknowledgments

We thank Michael Levin and Benjamin Pierce for providing an $F_{<}^\omega$ implementation that we could add to and work from in testing the examples presented here. We also thank Jens Palsberg and anonymous referees for comments on an earlier draft of this paper.

References

- [1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [2] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 396–409, New York, NY, USA, 1996. ACM Press.
- [3] Kim B. Bruce, , Adrian Fiech, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. Technical report. Draft technical report available through <ftp://cs.williams.edu/pub/kim/PolyTOIL.ps.gz> Extended abstract appered in ECOOP '95 Proceedings, LNCS 952, Springer-Verlag, pp. 27-51.
- [4] Kim B. Bruce. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. Technical report, Williams College, 1997. Available via URL <http://www.cs.williams.edu/~kim/README.html>.
- [5] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 1999. To appear in a special issue with papers from *Theoretical Aspects of Computer Software (TACS)*, September, 1997. An earlier version appeared as an invited lecture in the Third International Workshop on Foundations of Object Oriented Languages (FOOL 3), July 1996.
- [6] Kim B. Bruce, Jonathan Crabtree, and Gerald Kanapathy. An operational semantics for TOOPLE: A statically-typed object-oriented programming language. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics 9th International Conference, New Orleans, LA, USA, Proceedings*, volume 802 of *Lecture Notes in Computer Science*, pages 603–626. Springer-Verlag, New York, NY, April 1993.
- [7] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Fourth International Conference on Functional Programming and Computer Architecture*. ACM, September 1989. Also technical report STL-89-5, from Software Technology Laboratory, Hewlett-Packard Laboratories.
- [8] Luca Cardelli. Extensible records in a pure calculus of subtyping. Technical Report DEC-SRC-81, Digital Equipment Corporation, Systems Research Centre, January 92.
- [9] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):480–521, December 1985.
- [10] Adriana B. Compagnoni and Benjamin C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, October 1996.
- [11] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 433–443, October 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.
- [12] Karl Crary. Simple, efficient object encoding using intersection types. Technical Report CMU-CS-99-100, School of Computer Science, Carnegie Mellon University, 1999.
- [13] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In ACM, editor, *POPL '87. Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of programming languages, January 21–23, 1987, Munich, W. Germany*, pages 314–314, New York, NY, USA, 1987. ACM Press.
- [14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wellsley Professional Computing Series, 1995.
- [15] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages*, 1985.
- [16] Benjamin C. Pierce. Mutable objects. Working draft, Obtained by anonymous ftp from <ftp.lcs.ac.uk>, May 1993.
- [17] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.
- [18] Alan Snyder. Commonobjects: An overview. *ACM SIGPLAN Notices*, 21(10):19–28, October 1986.
- [19] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In Norman Meyrowitz, editor, *OOPSLA '86 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, pages 38–45. ACM SIGPLAN, ACM Press, 1986.