A Layered Object Oriented Programming Language

A. N. Clark

September 17, 1997

1 Abstract

A layered object oriented programming language (OOPL) is defined using a simple functional notation which has been equipped with a collection of features common to many OOPLs. The resulting language provides a framework within which existing OOPLs can be analysed and new OOPLs can be constructed.

2 Introduction

The object oriented programming paradigm has been seen as significant advance in software engineering quality. Its proponents cite features such as modularity, interfaces, polymorphism, inheritance and reuse as being characteristic of this new method for constructing software. It is, however, notoriously difficult to define exactly what is meant by the term "object oriented programming language"(OOPL); different practitioners will propose different characterising features. This fact and the lack of any formal description for the current object oriented programming languages, has meant that it is difficult to say exactly what, if any, the advances for software engineering are. Furthermore, if a precise understanding of the underlying object oriented primitives is lacking, then the design of new languages becomes a rather "hit and miss" affair.

It is well known that a class of languages may be precisely described by a simple functional language which has been enriched with special primitive features. The benefit of this approach is that functional languages (based upon the λ-calculus) are especially succinct and expressive and that they are readily extended with new and exotic features. Another benefit is that functional languages promote a uniform and "least restrictive" approach to language design; this is specially appropriate with respect to languages which hide many primitive features behind atomic syntactic constructs.

This paper shows how an OOPL can be constructed using a simple functional language which has been enriched with a small collection of characteristic primitives. The OOPL is constructed in layers, each layer contributes a new feature which is built by extending and modifying the previous layer. Where appropriate, variations for a particular layer are pointed out but will not contribute to the next layer. The language which is the result of the final layer contains mechanisms which are characteristic of the broad range of currently used OOPLs and the basic ideas and constructions provide a basis for a wide variety of languages and allows the OOPL design space to be mapped out in a systematic fashion.

The contribution which this paper makes to software engineering is to strip bare the underlying concepts of OOPLs and to show them as a collection of operations which manipulate record structures. This in itself is nothing new but the (perhaps) surprising aspect of OOPLs is the interaction between these operations and the mechanisms by which identifiers become associated with values. Record structures and identifier binding environments become two different sides of the same coin by providing operators which turn one into the other and back again. Such operators provide scope for the design of programming languages which support modularity in sophisticated ways to the extent that programmers can devise their own naming schemes.

The choice of OOPL mechanisms and the order in which they are layered is somewhat arbitrary. There are many more OOPL mechanisms which could be analysed and added to the language which is given here. The intention is that the presentation will give the reader confidence that the approach is sufficiently general in order that any language of a similar nature could be deconstructed into a collection of layers and rebuilt using a simple functional language as the implementation vehicle.

The author wishes to acknowledge the work of the following people in producing this report:

- Cook [8] [9] [10] [12] [13] [11] has been very influential in identifying the underlying concepts of OOPLs and was responsible for first reporting the correspondence between various occurrences of self reference in OOPLS and the **Y** combinator.

- [1] [18] [25] [17] were responsible for first reporting the correspondence between environment *reification* and inheritance.

- [4] [2] [3] have all developed elegant record calculi as a basis for OOPLs, although in most cases these works are more interested in the type theory behind object languages.

- The language which is developed in this report contains a collection of OOPL mechanisms which are loosely based upon those of the following real-life OOPLs: Smalltalk [15], C++ [28], Beta [20], Trellis/Owl [27], Eiffel [24] and Self [29].

- Landin [21] [22] was the first to propose functional languages as the basis for general programming language analysis.

This work in this report originated in a GEC-Marconi sponsored PhD undertaken by the author at Queen Mary and Westfield College, University of London, 1989-1994 [6]. An edited version of this paper is to appear in the GEC Journal of Research.

2.1 Guide to paper

§3 describes a simple functional language which will be used to build OOPL mechanisms. §4 describes the fundamental component of any OOPL which is a representation for named values, sometimes called *records* or *environments*. §6 describes the kernel features which are required for most OOPLs, these include classes, objects, inheritance, instantiation and message passing. §7 describes how the overlap of names is controlled during inheritance. §8 describes how names in classes may be changed. §9 extends the single inheritance of §6 to multiple inheritance and describes some of the problems which are associated with inheriting from more than one place. §10 describes how names are controlled using exports between classes. §11 describes how the association between a usage occurrence of an identifier and a defining occurrence can be deferred using a mechanism called *late binding*. Finally, §12 describes a type of polymorphism which is peculiar to OOPLs called *inclusion* polymorphism.

3 A functional language

This section describes a simple functional language which will be used to express OOPL mechanisms in the rest of the report. The intention is that the language is representative of the class of functional programming languages with imperative features, such as the kernel of ML [16]. The language has all the major components such as λ-functions, pattern matching, currying, **let**, **let rec**, **where** and **where rec** binding, conditional expressions, infix operators and sections (*e.g.* (+1) is the unit addition operator). *This font* is used for program identifiers and **this font** is used for syntactic keywords. Strings, integers and booleans are builtin values. Readers unfamiliar with functional languages are directed to [14].

The combinators **Y**, **I** and **K** are builtin and have their usual meanings. Lists are builtin where [] is the empty list and $\_ :: \_$ is the list constructor. *map* is the list mapping operator. *lh* builds list homomorphisms from an associative binary operator, a unary mapping and a base value. Locations are first class data values constructed with the *loc* operator whose argument is the contents of the location. The contents of a location is produced by the ! operator, the contents of a location is updated by side effect using the $\_ := \_$ operator and a location is shallow copied using the *copy* operator. Expressions which contain side effects are sequenced using the syntax construct $\_ ; \_$. Sets are builtin with $\{\}$ as the empty set, $\{\_, \_, \ldots, \_\}$ and $\_ \cup \_$ as set constructors and $\_ \in \_$ as the membership predicate. Tags are builtin where we assume that there is some mechanism for generating a fresh tag and an equality predicate on tags. The operators *1st*, *2nd* and *3rd* map tuples to the first, second and third components respectively.

4 Layer 1: Records

The first OOPL layer defines *records*. A record, which will synonymously be referred to as an *environment*, is a collection of values which are indexed by keys, which are also values. Sometimes, records are modelled as indexed sets or as functions which map a key to a value. We will represent records as being constructed using three primitive operations: $\{\}$ which is the empty record, $\_ \mapsto \_$ which will construct a record with a single binding from a key k to a value v, $k \mapsto v$, and $\_ \oplus \_$ which appends records together. A key k will be looked up in a record e using the operator $e \bullet k$, if the key is associated with a value then the value will be returned, otherwise the distinguished value ϵ will be returned. If the record associates k with more than one value then the rightmost value will be returned, records are said to "shadow on the right". The set of keys in a record e is produced by $dom(e)$. The keys in a record e are restricted to just those in the set S by e/S. A key k_1 is replaced everywhere by a key k_2 in a record e by $e[k_2/k_1]$.

Records are the basis for all objects in OOPLs, the keys are the names in the interface of the object and the values are the methods and/or attributes. For example, a simple object which implements a two dimensional point is:

$$\textbf{let}\ \ o = (\text{“}x\text{”} \mapsto 0) \oplus (\text{“}y\text{”} \mapsto 0)$$

It rapidly becomes irksome to have to spell out the operators which construct records so we invent the following syntax:

$$\textbf{let}\ \ o = \{x = 0; y = 0\}$$

Records can contain functions, which are termed *methods* when the record is an object, the binding

$$\text{“}i\text{”} \mapsto \lambda j.e$$

will be written

$$\{i(j) = e\}$$

In many OOPLs, an object can be viewed as a finite state machine where the object contains a collection of locations which are updated by side effect when one of the object's methods are activated. These locations are termed *attributes* and are represented here as locations which are data values created using the *loc* operator and side effected using the $\_ := \_$ operator. If the point object is to contain updatable locations for the x and y positions then the object is:

$$\textbf{let}\ \ o = \{x = loc(0); y = loc(0)\}$$

and the x co-ordinate is updated by side effect:

$$(o \bullet \text{“}x\text{”}) := 3$$

Often, OOPLs represent objects as a restricted class of records. An object will not provide external access to some of the names in its interface which will remain "hidden" within its own methods. This is represented using λ-scoping by making the internal names local to the definition of the methods. For example if the point object was to make the x and y co-ordinates hidden and provide methods to get (x and y) and set (set) the co-ordinates then:

$$
\begin{array}{l}
\textbf{let } o = \\
\qquad \textbf{let } x = loc(0) \textbf{ in} \\
\qquad \textbf{let } y = loc(0) \\
\qquad \textbf{in } \{x() = !(x); y() = !(y); set(x', y') = (x := x'); (y := y')\}
\end{array}
$$

5 Layer 2: Prototypes

The second layer defines *prototypes*. A prototype is a "one-off" object which may be *cloned* to produce new objects. Prototypes perform computations in response to *messages* which contain the target, a method name and the message contents. The methods of a prototype o refer to o using the identifier *self*, when the prototype is cloned to produce a new object o', the values of *self* in the methods of o' will have changed to be o'.

In order to support prototypes, the syntax for object construction is extended slightly to introduce the binding for the *self* identifier. The syntax

$$\{i = e_1; f(j) = e_2\}$$

now has the meaning

$$
\begin{array}{l}
\textbf{let rec } g(i)(self) = \\
\qquad (\text{“}i\text{”} \mapsto \lambda().!(i)) \oplus \\
\qquad (\text{“}seti\text{”} \mapsto \lambda i'.i := i') \oplus \\
\qquad (\text{“}f\text{”} \mapsto \lambda j.e_2) \oplus \\
\qquad (\text{“}clone\text{”} \mapsto \lambda().g(copy(i))) \\
\textbf{in } g(loc(e_1))
\end{array}
$$

Interactions between objects occur by sending messages, the operator *send* takes three arguments which are the target of the message, the message name (*i.e.* method name) and the message contents (*i.e.* method argument):

$$\textbf{let } send(o, m, a) = ((o(o)) \bullet m)(a)$$

Notice that the value of *self* gets supplied when the message is sent and not when the object is created – this will be termed "open" self reference and is to be contrasted with "closed" self reference which is defined in §6. The point object from §4 can be implemented as follows:

$$\textbf{let } o = \{x = 0; y = 0\}$$

Prototype objects can be extended with new attributes and methods using the operator *extend*:

$$\textbf{let} \;\; \mathit{extend}(o_1, o_2)(o) = (o_1(o)) \oplus (o_2(o))$$

The *extend* operator acts like $\_ \oplus \_$ except that it threads the value o which is used for self reference through both the objects o_1 and o_2. *extend* gives rise to a binary tree shaped object where the nodes are calls to *extend* and the leaves are prototype record definitions.

5.1 Multiple selves

When a message is sent, the value of *self* is the same at all parts of the tree and this can be counterintuitive, particularly for left biased methods. For example consider the following object definitions:

$$\textbf{let} \;\; o_1 = \{i(x) = \mathit{self} \bullet \text{“}j\text{”}(x)\}$$

$$\textbf{let} \;\; o_2 = \{j(x) = e_1\}$$

$$\textbf{let} \;\; o_3 = \{j(x) = e_2\}$$

$$\textbf{let} \;\; o_4 = \mathit{extend}(o_1, o_2)$$

Now when the expression $\mathit{send}(o_4, \text{“}i\text{”}, e_3)$ is evaluated, the result is the execution of the expression e_1 and when $\mathit{send}(o_3, \text{“}i\text{”}, e_3)$ is evaluated, the result is ϵ because o_3 does not define a method named i. Unfortunately, because of the uniformity of the value of *self* throughout an object, the result of $\mathit{send}(\mathit{extend}(o_4, o_3), \text{“}i\text{”}, e_3)$ is the execution of e_2 because the definition of the method named j in o_3 will shadow that in o_4. It is argued in the OOPL Self that this is counterintuitive and it is suggested that a rule should be defined which causes methods defined by *self* to group in order to reflect their relative positions in the object. An object now becomes a function of a tree structure which has the general form

$$(o, r, t)$$

where o is an object which is used as the value of *self*, r is a record which binds attribute and method names to values and t is a list of trees. An object o is a tree structure which is constructed using the operator *extend*, the tree t which is supplied to o is isomorphic to o and the corresponding nodes will contain the value of *self* which is biased for that part of o. A simple object definition is now a function from a tree to a tree so that $\{i = e_1; f(j) = e_2\}$ is now defined to mean:

$$\begin{array}{l}
\textbf{let rec}\ \ g(i) = \\
\qquad \textbf{let rec}\ \ o(\mathit{self}, _, []) = \\
\qquad\qquad \textbf{let}\ \ m = \ \ \begin{array}[t]{l}(\text{“}i\text{”} \mapsto \lambda().!(i))\oplus \\ (\text{“}\mathit{seti}\text{”} \mapsto \lambda i'.i := i')\oplus \\ (\text{“}f\text{”} \mapsto \lambda j.e_2)\oplus \\ (\text{“}\mathit{clone}\text{”} \mapsto \lambda().g(\mathit{copy}(i)))\end{array} \\
\qquad\qquad \textbf{in}\ \ (o, m, []) \\
\qquad \textbf{in}\ \ o \\
\textbf{in}\ \ g(\mathit{loc}(e_1))
\end{array}$$

Such a definition is a leaf of an object tree (the value supplied as the subtree elements must be []), the value *self* is an object used throughout the methods of o and the result is a tree

$$(o, m, [])$$

where o is the object and m is the record containing the method and attribute bindings for the object.

$\mathit{extend}(o_1, o_2)$ is redefined to compose the objects o_1 and o_2 and produce an object which maps a tree to a tree. The argument tree contains two subtrees which are the arguments for the objects o_1 and o_2 and the resulting trees contain two subtrees which reflect the biasing of *self* in o_1 and o_2. The left hand subtree of the result contains the bindings from o_1 extended on the *left* by the bindings of o_2, so that the bindings of o_1 override those of o_2. The right hand subtree of the result contains the bindings from o_2 extended on the *left* by the bindings of o_1, so that the bindings of o_2 override those of o_1. The new definition for *extend* is as follows:

$$\begin{array}{l}
\textbf{let rec}\ \ \mathit{extend}(o_1, o_2)(_, _, [l, r]) = \\
\qquad \textbf{let}\ \ (o_3, r_1, t_1) = o_1(l)\ \ \textbf{in} \\
\qquad \textbf{let}\ \ (o_4, r_2, t_2) = o_2(r) \\
\qquad \textbf{in}\ \ (\mathit{extend}(o_3, o_4), r_1 \oplus r_2, \\
\qquad\qquad\qquad [(\mathit{extend}(o_4, o_3), r_2 \oplus r_1, \mathit{map}(\mathit{extendleft}(o_4, r_2))(t_1)), \\
\qquad\qquad\qquad\ (\mathit{extend}(o_3, o_4), r_1 \oplus r_2, \mathit{map}(\mathit{extendleft}(o_3, r_1))(t_2))]) \\
\qquad\qquad \textbf{where rec}\ \ \mathit{extendleft}(o_1, r_1)(o_2, r_2, t) = (\mathit{extend}(o_1, o_2), r_1 \oplus r_2, \mathit{map}(\mathit{extendleft}(o_1, r_1)(t)))
\end{array}$$

There is now the problem of how to compute the argument tree which is supplied to an object when a message is sent to it. Fortunately, no further work is necessary because the argument tree is the same as the tree which is produced, so it can be found using a fixed point:

$$\textbf{let}\ \ \mathit{send}(o, m, a) = ((\mathit{2nd}(\mathbf{Y}(o))) \bullet m)(a)$$

5.2 Dynamic inheritance

The prototype OOPL called Self provides a facility which it refers to as “dynamic inheritance”. This is the ability to extend an object with the value of one of its attributes. Since the attributes are all mutable, a change to the attribute causes a corresponding change in the extension. Our model is easily extended to deal with dynamic inheritance by introducing a new form of record binding, $\stackrel{*}{=}$ which defines an attribute binding, the value of which must be an object which is "spliced" in to the containing object just after the attribute definition. Let b_1 and b_2 be collections of arbitrary bindings then

$$\{b_1; i \stackrel{*}{=} e; b_2\}$$

is defined to have the following meaning:

$$
\begin{array}{l}
\mathit{extend}(\{b_1\}, \mathit{extend}(o, \{b_2\})) \\
\quad\quad \textbf{where} \\
\quad\quad\quad\quad o = \ \ \textbf{let} \ \ o = \{i = e\} \\
\quad\quad\quad\quad\quad\quad\ \ \textbf{in} \ \ \mathit{extend}(\lambda t.\textbf{let} \ \ (\_, r, \_) = o(t) \ \ \textbf{in} \ \ ((r \bullet \text{``}i\text{''})())(t), o)
\end{array}
$$

6 Layer 3: Classes

The third layer defines *classes*. A class is a general pattern from which an object is created by filling in some of the missing components of the pattern. Generally a class describes the methods and attribute names of a collection of objects which are termed *instances* of the class. An instance is created by supplying the class with values for the attributes. In such languages there will be operations for constructing and composing classes and objects.

At its simplest, a class is just a function which returns an object. The formal parameters of the function are the attributes of the object and the object is a record of methods. The rest of this paper deals with OOPL mechanisms which create and manipulate classes of increasing complexity. Each new type of class is described by layering some extra facilities on the previous type of class. The mechanisms are defined in terms of five different components which comprise the basics of classes in many OOPLs:

1. An extension which is a new type of function

   $$\Lambda(a_1, a_1, \ldots, a_n).\{i = e_1; f(j) = e_2\}$$

   which may be added to an existing class in order to produce a new class. The identifiers a_i represent new attributes and the body of the function is the record of new definitions. The definition for i is a value and the definition for f is a function. A Λ-function is defined in terms of a *desugaring*, *i.e.* it is a macro which is defined in terms of a translation to another expression whose meaning is well defined.

2. A class constructor which is a two place operator $\mathit{class}(c, x)$ where c is a class and x is an extension. If $c_1 = \mathit{class}(c_2, x)$ then c_2 is a *superclass* of c_1 and c_1 is a *subclass* of c_2.

3. A class instantiation operator which is a two place function $new(c, v)$ where c is a class and v is a value which contains all the attribute initialisation values. If $o = new(c, v)$ then o is an *instance* of c.
4. The empty class which defines no attributes and no methods: *nullclass*.
5. The message passing operator *send* which will invoke a method named n of an object o with the argument a $send(o, n, a)$.

Each section gives a definition for these five class related components (where a definition for a particular component is omitted the most recent definition from a preceding section will prevail). Initially, an extension is translated to the following expression

$$\lambda(a_1, a_2, \ldots, a_n).\lambda self.(\text{“}i\text{”} \mapsto e_1) \oplus (\text{“}f\text{”} \mapsto \lambda j.e_2)$$

both the class constructor and the empty class are undefined and the instantiation operator has the following definition

$$\textbf{let}\ \ new(c, v) = \mathbf{Y}(c(v))$$

The initial message passing operator is

$$\textbf{let}\ \ send(o, n, a) = (o \bullet n)(a)$$

A simple class which creates two dimensional point is defined below:

$$\textbf{let}\ \ point = \Lambda(x, y).\{x() = !(x); y() = !(y); set(x', y') = (x := x'); (y := y')\}$$

and a new point object is created by instantiating the class

$$\textbf{let}\ \ o = new(point(loc(0), loc(0)))$$

The value which is supplied to the object as the value of *self* is fixed when the class is instantiated, this is termed “closed self reference” . The value of *self* which is supplied is the same value as is returned by the instantiation – so closed self reference is achieved by taking a fixed point.

6.1 Inheritance

Classes really come into their own when they are allowed to *inherit* from other classes. Inheritance is not a precisely defined concept but usually covers the process of combining and extending classes to construct new classes. Classes which are constructed using inheritance are composite, *i.e.* they comprise more than one component class and their instances are a corresponding combination of the instances of the component classes. The simplest form of inheritance is *single* inheritance where each class is constructed by extending an existing class. The ultimate existing class is the null class which is defined

$$\textbf{let}\;\; \mathit{nullclass}()(\_) = \{\}$$

An extension which is added to an existing class is just a parameterised record as described above. The operator *class* constructs a new class from an existing class c (called the *superclass*) and an extension e. Each class has an argument which is a tree of initialisation values for the attributes of each of the component extensions. A tree of initialisation values is either () for the null class or a pair $(t, (v_1, v_2, \ldots, v_n))$ where t is the initialisation tree for the superclass and $(v_1, v_2, \ldots, v_n)$ are the initialisation values for the attributes which are defined by the extension.

$$\textbf{let}\;\; \mathit{class}(c, e)(t, v)(o) = (c(t)(o)) \oplus (e(v)(o))$$

A two dimensional point class is defined

$$\textbf{let}\;\; \mathit{point} = \mathit{class}(\mathit{nullclass}, \Lambda(x, y).\{x() = !(x); y() = !(y); \mathit{set}(x', y') = (x := x'); (y := y')\})$$

Now consider defining a class *3dpoint* which adds a z co-ordinate as an extension to *point*, it is not possible to update the *set* method because the values of the x and y attributes are hidden within the *point* extension. Class based inheritance will extend the scope of attribute identifiers to include all the extensions of the subclasses. This is a new type of scoping rule which differs from the rules which are employed by other types of programming language such as Pascal and Ada. In order to describe the meaning of this new type of identifier scoping we use *environment reification*, and then it will be possible to define *3dpoint* as

$$\textbf{let}\;\; \mathit{3dpoint} = \mathit{class}(\mathit{point}, \Lambda z.\{z() = !(z); \mathit{set}(x', y', z') = (x := x'); (y := y'); (z := z')\})$$

6.2 Environment reification

At any stage during the performance of a program there is a current environment which describes associations between program identifiers and their values. An identifier is *in scope* if it is bound to a value in the current environment. Function application extends the current environment as in

$$(\lambda(x, y).x + y)(1, 2)$$

the current environment is extended by associations between x and 1 and y and 2 (overriding any other associations for x and y for the scope of the function body) while the expression $x + y$ is performed. Functions are first class values which may be passed as arguments and returned as results. When a function is created, the current environment is remembered at its boundary so that the environment may be made current and used when the function is eventually applied to an argument. Consider the following example which has been annotated with points of interest a, b and c:

$$
\begin{array}{l}
\textbf{let } z^a = 1 \textbf{ in} \\
\textbf{let } f(g) = \textbf{let } z^c = 2 \textbf{ in } g(1,2) \\
\textbf{in } f(\lambda^b(x,y).x + y + z)
\end{array}
$$

At point a the current environment is extended with an association between z and 1, the function f is then defined and added to the current environment. At point b an anonymous function is constructed which remembers the current environment at its boundary. Inside f the current environment is extended at c with an association between z and 2 which overrides the association for z which was established at a. When g is applied, the environment which is currently at the boundary of the function is made current and extended with the associations for its formal parameters. The result of performing the entire expression is 4 and not 5.

The operator R which is pronounced "reify" will produce the environment which has been saved at the boundary of a function. The environment is produced as a record (actually record $\equiv$ environment) structure. The operator I which is pronounced "install" will replace the environment which has been saved at the boundary of a function and produce a new function. The following equality holds for reification and installation

$$f = I(R(f), f)$$

Given a function f, it is possible for f to manipulate the environment which is saved at its own boundary by making f recursive,

$$\textbf{let rec } f() = R(f)$$

Using record subtraction, it is possible for a function to manipulate the environment which contains just the extensions which are added to the boundary environment just inside the function's application

$$\textbf{let rec } f(x,y,z) = (R(\lambda().())) - (R(f))$$

Now the application $f(1,2,3)$ will produce the environment $(\text{“}x\text{”} \mapsto 1) \oplus (\text{“}y\text{”} \mapsto 2) \oplus (\text{“}z\text{”} \mapsto 3)$.

The operator $\_ \hookrightarrow \_$ is used to extend the environment which is saved at the boundary of a function, its definition is as follows:

$$\textbf{let } e \hookrightarrow f = I(R(f) \oplus e, f)$$

The expression

$$\textbf{open } e_1 \textbf{ in } e_2$$

will have the operational meaning that e_1 is evaluated to produce an environment which is then added to the current environment for the duration of the evaluation of e_2. This semantics is achieved by viewing the **open** expression as having the following definition

$$(e_1 \hookrightarrow \lambda().e_2)()$$

6.3 Inheritance scoping

Attribute identifiers which are defined by superclasses are scoped over the extensions defined by subclasses using environment reification and installation. An extension is modified to be a function whose arguments are the new attributes and which will return a pair (a, m) where a is a record containing the attributes defined by the extension and m is a function (called the m-function) which maps a record called *self* to another record containing the extension definitions. An extension is defined to mean

$$\begin{array}{l} \textbf{let rec}\ \ x(a_1, a_2, \ldots, a_n) = ((R(\lambda().())) - (R(x)), m) \\ \qquad\qquad\qquad \textbf{where} \\ \qquad\qquad\qquad\qquad m(\mathit{self}) = (\text{“}i\text{”} \mapsto e_1) \oplus (\text{“}f\text{”} \mapsto \lambda j.e_2) \end{array}$$

The expression $R(\lambda().())$ produces the current environment and the expression $R(\lambda().()) - R(x)$ subtracts the environment at the boundary of the function x from the current environment – producing the environment which just contains the bindings for the attributes $a_1, a_2, \ldots, a_n$. The m-function has a single argument which is an object called *self* and returns the object which is defined by the extension.

The *class* constructor is updated to incorporate inheritance scoping as follows:

$$\begin{array}{l} \textbf{let}\ \ \mathit{class}(c, e)(v_1, v_2)(o_1) = \\ \qquad \textbf{let}\ \ (a_1, o_2) = c(v_1)(o_1)\ \ \textbf{in} \\ \qquad \textbf{let}\ \ (a_2, m) = (a_1 \hookrightarrow e)(v_2) \\ \textbf{in}\ \ (a_1 \oplus a_2, o_2 \oplus m(o_1)) \end{array}$$

The superclass c is instantiated to produce a pair (a_1, o_2) which contains attribute bindings a_1 and object o_2. The extension function e is then extended on the right by the attributes a_1 and instantiated to produce a pair (a_2, m) where a_2 are the attributes defined by the extension and m is an m-function. Finally, the result is a pair consisting of the combination of attribute environments $a_1 \oplus a_2$ and method environments $o_2 \oplus m(o_1)$.

A class is instantiated using the operator *new*. The argument to new is a pair (c, t) where c is the class to instantiate and t is the tree of initialisation values.

$$\textbf{let}\ \ \mathit{new}(c, t) = \mathit{2nd}(\mathbf{Y}(c(t)))$$

The following definitions show how the two dimensional point class is extended to produce the three dimensional point class. The method *set* is redefined in the class *3dpoint* in order to set the z co-ordinate and the inherited attributes x and y may be referenced without further qualification.

$$\textbf{let } \textit{nullclass}() = (\{\}, \mathbf{K}(\{\}))$$

$$\textbf{let } \textit{point} = \textit{class}(\textit{nullclass}, \Lambda(x,y).\{x() = !(x); y() = !(y); \textit{set}(x', y') = (x := x'); (y := y')\})$$

$$\textbf{let } \textit{3dpoint} = \textit{class}(\textit{point}, \Lambda z, \{z() = !(z); \textit{set}(x', y', z') = (x := x'); (y := y'); (z := z')\})$$

7 Layer 4: Shadowing

The fourth layer defines how to cheat the rules of identifier scoping. When a new class is defined, the extension part may define a method with the same name as a method in the superclass. The rules of record lookup define that the value which is returned when a key is looked up in a record is the rightmost value. This means that the method which is defined by the superclass is lost forever as far as the subclass is concerned, even though the original definition for the method may prove useful. The OOPL Smalltalk-80 allows the shadowed methods to be referenced through a distinguished identifier *super*. Given a class $\textit{class}(c, e)$ the definitions which are given by c are referenced using *super* in the bodies of all the definitions of e.

An extension is modified to have an extra implicit identifier *super* which is scoped over all of the definitions. An extension is defined to mean

$$\begin{array}{l} \textbf{let rec } x(a_1, a_2, \ldots, a_n) = ((R(\lambda().())) - (R(x)), m) \\ \quad \textbf{where} \\ \qquad m(\textit{super}, \textit{self}) = (\text{“}i\text{”} \mapsto e_1) \oplus (\text{“}f\text{”} \mapsto \lambda j.e_2) \end{array}$$

Now the class constructor is modified in order to supply the value of *super* to each extension. This is done by instantiating the superclass to produce a pair (a, o) and using o as the value of *super* in the extension.

$$\begin{array}{l} \textbf{let } \textit{class}(c, x)(v_1, v_2)(o_1) = \\ \quad \textbf{let } (a_1, o_2) = c(v_1)(o) \textbf{ in} \\ \quad \textbf{let } (a_2, m) = (a_1 \hookrightarrow x)(v_2) \\ \quad \textbf{in } (a_1 \oplus a_2, o_2 \oplus (m(o_2, o_1))) \end{array}$$

It is now possible to define a subclass which overrides methods of the superclass and allow the extension methods to access the overridden superclass methods:

$$\textbf{let } \textit{point} = \textit{class}(\textit{nullclass}, \Lambda(x,y).\{x() = !(x); y() = !(y); \textit{set}(x', y') = (x := x'); (y := y')\})$$

$$\textbf{let } \textit{3dpoint} = \textit{class}(\textit{point}, \Lambda z, \{z() = !(z); \textit{set}(x', y', z') = ((\textit{super} \bullet \text{“}\textit{set}\text{”})(x', y')); (z := z')\})$$

Although most OOPLs specify that the definitions of an extension will shadow those with the same name from the superclass, this is not always the case. The

OOPL Beta is an example of a language where the superclass methods shadow the extension methods with the same name. Beta represents an inversion of the Smalltalk mechanism because the extension methods are available to the superclass methods via a distinguished identifier *inner*. The "inner" mechanism has a slightly different purpose to the "super" mechanism. The latter is intended to reuse useful methods which would otherwise be lost due to the shadowing rules and the former is intended to allow methods to control how extensible they are. We show how the Beta mechanism can be implemented and then show how the two mechanisms can be merged into one.

An extension is modified to have an implicit identifier *inner* which is scoped over all of the definitions. An extension is defined to mean

$$\begin{array}{l}\textbf{let rec } x(a_1, a_2, \ldots, a_n) = ((R(\lambda().())) - (R(x)), m) \\ \quad\quad \textbf{where} \\ \quad\quad\quad\quad m(\mathit{inner}, \mathit{self}) = (\text{“}i\text{”} \mapsto e_1) \oplus (\text{“}f\text{”} \mapsto \lambda j.e_2)\end{array}$$

A class c_1 is composed with an extension e to produce a new class c_2. When c_2 is instantiated, e will be supplied with initial values for its attributes and will produce an object o which is supplied to c_1 as the value of *inner*. In general, a class is parameterised with respect to the definitions with which it can be extended. The class is used in the construction of arbitrarily many subclasses by extending it any number of times. When any of these subclasses are instantiated, the original class will be supplied with object containing all of the definitions from the extension parts and uses this object as the value of *inner*:

$$\begin{array}{l}\textbf{let } \mathit{class}(c, e)(v_1, v_2)(o_1)(o_2) = \\ \quad \textbf{let rec } \begin{array}{l}(a_1, o_3) = c(v_1)(o_1 \oplus o_4)(o_2) \\ (a_2, m) = (a_1 \hookrightarrow e)(v_2) \\ o_4 = m(o_1, o_2)\end{array} \\ \quad \textbf{in } (a_1 \oplus a_2, o_3 \oplus o_4)\end{array}$$

The extension object o_1 is supplied when the class is instantiated. When a class, constructed from a superclass c and an extension e, is instantiated, the instantiation of the two components will be mutually dependent: instantiating the superclass depends upon the result of instantiating the extension because this supplies part of the *inner* value and the instantiation of the extension will depend upon the instantiation of the superclass for the inherited attributes a_1. This mutual dependency is solved by taking a fixed point.

Classes are instantiated using the operator *new* which supplies the null environment $\{\}$ as the initial value of *inner* (since when the class c is instantiated there is no extension component):

$$\textbf{let } \mathit{new}(c, v) = \mathit{2nd}(\mathbf{Y}(c(v)(\{\})))$$

It is now possible to define a method named *set* in the *point* class which is extensible. The value v which is supplied to this method will contain at least two components which will be the new values of the x and y co-ordinates. After updating the co-ordinates, the *set* method of the *inner* component will be invoked (in reality there would be some check here to see if the inner method was actually defined – this is omitted here to avoid unnecessary code). If the *point* class has been extended with a class which has a definition for *set* then this will get executed. The *3dpoint* class takes advantage of this by extending the *set* method to update the z co-ordinate.

$$\textbf{let}\ \ point = class(nullclass, \Lambda(x,y).\{x() = !(x); y() = !(y); set(v) = (x := 1st(v)); (y := 2nd(v)); (inner \bullet \text{“}set\text{”}(v))\})$$

$$\textbf{let}\ \ 3dpoint = class(point, \Lambda z, \{z() = !(z); set(v) = z := 3rd(v)\})$$

The two mechanisms for cheating the rules of identifier scope are merged into a single mechanism as follows. An extension is modified to be a function of both the shadowed definitions in *super* and the shadowing definitions in *inner*. An extension is defined to mean

$$\begin{array}{l} \textbf{let rec}\ \ x(a_1, a_2, \ldots, a_n) = ((R(\lambda().())) - (R(x))), m) \\ \quad \textbf{where} \\ \qquad m(super, inner, self) = (\text{“}i\text{”} \mapsto e_1) \oplus (\text{“}f\text{”} \mapsto \lambda j.e_2) \end{array}$$

The class constructor is extended to supply both the *super* and *inner* values to the extension:

$$\begin{array}{l} \textbf{let}\ \ class(c,e)(v_1,v_2)(o_1)(o_2) = \\ \quad \textbf{let rec}\ \ \begin{array}{l} (a_1, o_3) = c(v_1)(o_1 \oplus o_4)(o_2) \\ (a_2, m) = (a_1 \hookrightarrow e)(v_2) \\ o_4 = m(o_3, o_1, o_2) \end{array} \\ \quad \textbf{in}\ \ (a_1 \oplus a_2, o_4 \oplus o_3) \end{array}$$

The superclass c is supplied with all the definitions which are provided by the subclasses $o_1 \oplus o_4$ and the m-function is supplied with all the definitions which are provided by the subclasses o_1 and all the definitions from the superclass o_3. A decision must be made as to whether subclass definitions shadow superclass definitions or *vice versa*. Since these are orthogonal, we present the variation in which the subclass definitions shadow superclass definitions with the same name (*i.e.* $o_4 \oplus o_3$), the alternative variation is defined in exactly the same way with the end results swapped, (*i.e.* $o_3 \oplus o_4$).

8 Layer 5: Naming

The fifth layer shows how to rename attributes and methods which are defined in a class. Given a class c, a new name n_1 and an existing attribute name n_2, *renameattribute* produces a new class in which the attribute have been renamed for any inheriting class:

$$\begin{array}{l} \textbf{let}\ \ renameattribute(c, n_1, n_2)(v)(o_1)(o_2) = \\ \qquad \textbf{let}\ \ (a, o) = c(v)(o_1)(o_2) \\ \qquad \textbf{in}\ \ (a[n_1/n_2], o) \end{array}$$

Renaming a definition is slightly more complex because of the mechanisms which are used to pass the values of *inner* and *self* back to the superclass. These two values are environments containing definitions which have been renamed, but which are referred to using the original names in the superclass. In order to achieve the desired behaviour, the methods are renamed *in reverse* when the records are passed back.

$$\begin{array}{l}\textbf{let } \ \mathit{renamemethod}(c, n_1, n_2)(v)(o_1)(o_2) = \\ \qquad \textbf{let } \ (a, o) = c(v)(o_1[n_2/n_1])(o_2[n_2/n_1]) \\ \qquad \textbf{in } \ (a, o[n_1/n_2])\end{array}$$

9 Layer 6: Multiple inheritance

Inheritance in OOPLs is a mechanism for extending packages of definitions in order to produce new packages; §6 describes how this is done using single inheritance where each new package is constructed by adding an extension to a single existing class. Many classes will contain useful definitions which can be included in subclasses; but since a class may have only one parent in a single inheritance system, useful classes are forced further and further up the inheritance tree. This is similar to forcing identifier binding occurrences further out with respect to the binding scopes so that they are visible to all parts of the program which require them. The disadvantage of this approach is that the information becomes visible over parts of the program which do not require it and this compromises program modularity.

The solution to this problem which has been adopted by many OOPLs is to generalise inheritance so that a class may have more than one parent. In this way modularity can be controlled because bindings can be included in just those definitions which require them, *i.e.* there is a much finer grain control over the scopes of identifier bindings. This section describes the sixth layer which adds the basics of multiple inheritance.

Multiple inheritance uses a new class constructor *siblings* which composes two classes to produce a new single class. *siblings* does not cause either of the two original classes to inherit from the other, but if the resulting class is used as a superclass as in

$$\textbf{let } \ c = \mathit{class}(\mathit{siblings}(c_1, c_2), e)$$

then c will inherit from both c_1 and c_2. A simple definition for *siblings* is as follows:

$$\begin{array}{l}\textbf{let } \ \mathit{siblings}(c_1, c_2)(v_1, v_2)(o_1)(o_2) = \\ \quad \textbf{let } \ (a_1, o_1) = c_1(v_1)(o_1)(o_2) \ \textbf{ and} \\ \quad \phantom{\textbf{let }} \ (a_2, o_2) = c_2(v_2)(o_1)(o_2) \\ \quad \textbf{in } \ (a_1 \oplus a_2, o_1 \oplus o_2)\end{array}$$

There are a number of points to notice about this definition of *siblings*: both classes are instantiated with respect to the same value for *inner*, *i.e.* o_1, both classes can be instantiated independently and the instantiation is not symmetric because the methods and attributes for c_2 will override those with the same name in c_1.

9.1 Mixins

A good example of a technique which is supported by multiple inheritance is "mixins". A mixin is a class which has yet to be supplied with a superclass. The mixin represents a description of a general type of behaviour for a collection of objects and gives just enough implementation detail while leaving the finer points to the different classes into which it will be mixed. The example which we use in order to show how mixins work is that of a border on a geometric shape. Any shape can be given a border which is that portion around the outside of the object which must be treated specially (it is not writable, for example). All the shapes have an area and the effect of adding a border to any shape will be to reduce the area of the shape by the space which the border takes up. A mixin class *bordered* is defined which describes how a border is added to any geometric shape class, the mechanics of how the border is described, the area is computed and how the size of the border affects the area will be up to the classes into which *bordered* is mixed. The following definitions show how *bordered* is added to the class *circle*:

$$\textbf{let}\ \ \mathit{circle} = \mathit{class}(\mathit{nullclass}, \Lambda r.\{\mathit{area}() = \pi * !(r)^2\})$$

$$\textbf{let}\ \ \mathit{bordered} = \mathit{class}(\mathit{nullclass}, \Lambda b.\{\mathit{area}() = (\mathit{super} \bullet \text{“}\mathit{area}\text{”}()) - (\mathit{inner} \bullet \text{“}\mathit{border}\text{”}())\})$$

$$\textbf{let}\ \ \mathit{borderedcircle} = \mathit{class}(\mathit{class}(\mathit{circle}, \mathit{bordered}), \Lambda().\{\mathit{border}() = \pi * !(r)^2 - \pi * (!(r) - !(b))^2\})$$

The class *bordered* is initially defined with a superclass *nullclass* even though it executes the method named *area* with respect to the value of *super*, this is an indication that *bordered* is a mixin class and is waiting to be supplied with different superclasses. The class *bordered* also leaves it up to an extension to supply the details of the *border* method which computes the area of the border for the particular geometric shape class.

Mixins are supported in our model by leaving a class open with respect to a super object (in much the same way as classes are left open to the inner object as described in §7). In fact, there is a pleasing symmetry between the mechanisms which achieve *super* and *inner* and the distinction between classes and extensions is removed, *i.e.* both *class* and Λ-expressions build classes.

A class is modified to be a function which maps a tree of attribute initial values to a pair (a, m) where a is the environment of attribute bindings and m is a curried two-argument function which produces an object. The first argument of m is $(\mathit{super}, \mathit{inner})$ which is the super and inner definitions for the class and the second argument will be *self* which is the entire object. A Λ-expression is modified to construct a class and is defined to mean

$$\begin{array}{l}\textbf{let rec } x(a_1, a_2, \ldots, a_n) = ((R(\lambda().())) - (R(x)), m) \\ \quad\quad \textbf{where} \\ \quad\quad\quad\quad m(\mathit{super}, \mathit{inner})(\mathit{self}) = (\text{“}i\text{”} \mapsto e_1) \oplus (\text{“}f\text{”} \mapsto \lambda j.e_2)\end{array}$$

The *class* constructor is modified as follows:

$$\begin{array}{l}\textbf{let } \mathit{class}(c_1, c_2)(v_1, v_2) = \\ \quad \textbf{let } \ (a_1, m_1) = c_1(v_1) \\ \quad\quad\quad (a_2, m_2) = (a_1 \hookrightarrow c_2)(v_2) \ \textbf{in} \\ \quad \textbf{let } \ m(o_1, o_2)(o_3) = \\ \quad\quad\quad\quad \textbf{let rec } \ o_4 = m_1(o_1, o_2 \oplus o_5)(o_3) \\ \quad\quad\quad\quad\quad\quad\quad\quad o_5 = m_2(o_1 \oplus o_4, o_2)(o_3) \\ \quad\quad\quad\quad \textbf{in } \ o_4 \oplus o_5 \\ \quad \textbf{in } \ (a_1 \oplus a_2, m)\end{array}$$

$$\textbf{let } \mathit{nullclass}() = (\{\}, \mathbf{K}(\mathbf{K}(\{\})))$$

$$\textbf{let } \mathit{new}(c, v) = \mathbf{Y}(\mathit{2nd}(c(v))(\{\}, \{\}))$$

A slight problem exists with the definition of *class* which is given above, the expression $a_1 \hookrightarrow c_2$ will not have the desired effect because the attribute environment a_1 will be installed into the class which may be composite. If this is the case then the environment at the boundary of the function returned by the *class* constructor which created c_2 will be extended with the attributes, but at best this will have no effect and at worst some of the identifier bindings which are required in the body of *class* may be affected in undesired ways. Fortunately, the operator $\_ \hookrightarrow \_$ is easily extended, given the existence of a predicate *isclass* which is true for any function $\mathit{class}(c_1, c_2)$ and a deconstructing operator which is specified as

$$(c_1, c_2) = \mathit{classdecompose}(\mathit{class}(c_1, c_2))$$

The extended definition of $\_ \hookrightarrow \_$ is as follows:

$$\begin{array}{l}\textbf{let rec } e \hookrightarrow f = \\ \quad\quad \textbf{if } \ \mathit{isclass}(f) \\ \quad\quad \textbf{then let } \ (c_1, c_2) = \mathit{classdecompose}(f) \ \textbf{ in } \ \mathit{class}(e \hookrightarrow c_1, e \hookrightarrow c_2) \\ \quad\quad \textbf{else } \ I(R(f) \oplus e, f)\end{array}$$

9.2 Repeated inheritance

Multiple inheritance raises a problem which is how to deal with inheriting the same class twice. Consider the following class definitions:

$$\textbf{let } c_1 = \mathit{class}(\mathit{nullclass}, \Lambda().\{a = \mathit{loc}(0); m(x) = e_1\})$$

$$\textbf{let } c_2 = \mathit{class}(c_1, \Lambda().\{m(x) = e_2\})$$

$$\textbf{let } c_3 = \mathit{class}(c_1, \Lambda().\{\})$$

$$\textbf{let } c_4 = \mathit{class}(\mathit{siblings}(c_2, c_3), \Lambda().\{\})$$

c_1 is a class which defines an attribute a which is a location and a method m. The class c_2 inherits from c_1 and redefines the method named m. Class c_3 inherits from c_1 and finally the class c_4 inherits from both c_2 and c_3. Because both c_2 and c_3 inherit from c_1, c_4 will inherit from c_1 too; but it will inherit from it *twice*. Will c_4 wind up having two locations which are called a or just one? Will the original definition of m from c_1 shadow the overriding definition from c_2 in c_4?

We extend the class construction so that the programmer can have control when resolving issues to do with repeated inheritance. The default rule is that all locations are shared and that the leftmost occurrence of a repeatedly inherited name is the defining occurrence. The programmer may override the default for particular names in individual classes.

A class is represented as a term $C(g, \sigma)$ where g is a "generator" for instances of the class and σ contains the names which are defined by the class and associates each with a unique tag. The C part of the term is just a constructor and is used to distinguish the class from other types of term. A generator g is defined to be a function whose argument is

$$(\mathit{super}, \mathit{inner}, T, \sigma)$$

where *super* and *inner* are the values as defined in the previous section, T is a set of tags and σ associates names with tags. The extra arguments are used to control the repeated inheritance when the class is instantiated. The σ argument contains the tags which are used to identify the definitions in the class, and the set T contains the tags of definitions which have already been instantiated. When a class is instantiated, if it finds that some of its definitions have already been included in the instantiation (due to repeated inheritance) then the class will simply not bother to include those definitions in its result. An extension is defined to have the following meaning:

$$
\begin{array}{l}
C(x, \sigma) \\
\quad \textbf{where} \\
\qquad \sigma = (\text{“}i\text{”} \mapsto \tau_1) \oplus (\text{“}f\text{”} \mapsto \tau_2) \\
\qquad \textbf{rec } x(a_1, a_2, \ldots, a_n) = ((R(\lambda().())) - (R(x)), m) \\
\qquad\qquad \textbf{where} \\
\qquad\qquad\quad m(\mathit{super}, \mathit{inner}, T, \sigma)(\mathit{self}) = (o, T \cup \{\tau_1, \tau_2\}) \\
\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\quad o = (\textbf{if } (\sigma \bullet \text{“}i\text{”}) \in T \textbf{ then } \{\} \textbf{ else } \text{“}i\text{”} \mapsto e_1) \oplus \\
\qquad\qquad\qquad\qquad (\textbf{if } (\sigma \bullet \text{“}f\text{”}) \in T \textbf{ then } \{\} \textbf{ else } \text{“}f\text{”} \mapsto \lambda j.e_2)
\end{array}
$$

The *class* constructor must now arrange to compose both the generators and name/tag associations:

$$
\begin{array}{l}
\textbf{let } \mathit{class}(C(g_1, \sigma_1), C(g_2, \sigma_2)) = C(g, (\sigma_1, \sigma_2)) \\
\quad \textbf{where} \\
\qquad g(v_1, v_2) = \begin{array}[t]{l}
\textbf{let } (a_1, m_1) = g_1(v_1) \textbf{ in} \\
\textbf{let } (a_2, m_2) = (a_1 \hookrightarrow g_2)(v_2) \textbf{ in} \\
\textbf{let } m(o_1, o_2, T, (\sigma_1, \sigma_2))(o_3) = \\
\qquad \textbf{let rec } \begin{array}[t]{l} (o_4, T) = m_1(o_1, o_2 \oplus o_5, T, \sigma_1) \\ (o_5, T) = m_2(o_1 \oplus o_4, o_2, T, \sigma_2) \end{array} \\
\qquad \textbf{in } (o_4 \oplus o_5, T)
\end{array} \\
\qquad\qquad \textbf{in } (a_1 \oplus a_2, m)
\end{array}
$$

$$\textbf{let } \mathbf{1}(f)(x)(y) = \mathit{1st}(f(x)(y))$$

$$\textbf{let } \mathit{new}(C(g, \sigma), v) = \mathbf{Y}(\mathbf{1}(\mathit{2nd}(g(v))(\{\}, \{\}, \{\}, \sigma)))$$

$$
\begin{array}{l}
\textbf{let } \mathit{nullclass} = C(g, \{\}) \\
\quad \textbf{where} \\
\qquad g() = (\{\}, \lambda(\_, \_, T, \_)(\_).(\{\}, T))
\end{array}
$$

The default behaviour for repeated inheritance preserves only the leftmost occurrences of a definition in a class. In order to override this a simple operator *copydef* can be devised which maps a class and a name n to a new class in which the σ component has been modified to associate n with a brand new tag. When both the original and new versions of the class are instantiated as part of a larger class, both of the definitions for n will be executed because the associated tags will be different.

10 Layer 7: Exports

The seventh layer defines how to control name spaces. Inheritance gives a structured mechanism for controlling name spaces, for example the scope of the attribute identifiers which are defined by a class are limited to the methods of the class and all of its subclasses. Eiffel distinguishes between two different types of customer for a class c, the subclasses of c are one type of customer and any class which may refer to an instance of c is another. Inheritance is a way of controlling the names in a program with respect to the first type of customer, *export* is a way of controlling the names with respect to the second.

When a name is referenced in an object, the reference will take place with respect to a *client* which is a class. If the reference takes place in the body of a method then the class will usually be the that which defines the method, but, since classes are fully fledged values, this does not have to be the case. The client in a name reference will control whether the name is visible or not. When a class is defined, the names which are introduced can be exported to any number of client classes. If these classes are used as the clients in a name reference for an instance of the exporting class then the reference will succeed, otherwise it will not. Exporting interacts with inheritance in the following two ways:

1. If c_1 exports a name n to c_2 then, by default, all subclasses of c_1 export n to c_2.

2. A class c_2 will be defined to "import a name n from c_1" when c_1 exports n to c_2. If c_2 imports a name n from c_1 then, by default, all the subclasses of c_2 will import n from c_1.

A name which is defined by a class is exported using the operator *exportdef* which takes the name, the importing class and the exporting class as arguments and returns a new class which is the same as the exporting class except that the name will be exported in all instances of the new class. For example, consider the following class definition

$$\textbf{let}\ \ c = \mathit{exportdef}(\text{“}m\text{”}, \mathit{nullclass}, \Lambda().\{m(a) = e_1\})$$

c is a class which defines a single name m and exports it to *nullclass*, notice however that c does not inherit from *nullclass*. All name references in objects are now performed using the operator *ref* which takes the object, the name and the client as arguments and either signals an error if the reference is illegal or produces the result of looking up the name in the object. The following reference is legal

$$\mathit{ref}(\mathit{new}(c, ()), \text{“}m\text{”}, \mathit{nullclass})$$

because the name m was exported to the class *nullclass*, however the following reference

$$\mathit{ref}(\mathit{new}(c, ()), \text{“}m\text{”}, c)$$

is illegal because, although the class c defines the name m it does not export the name to itself! If the class c is redefined so that it still exports the name to *nullclass* but also inherits from *nullclass* then the reference will become legal,

$$\mathbf{let}\ \ c = \mathit{class}(\mathit{nullclass}, \mathit{exportdef}(\text{“}m\text{”}, \mathit{nullclass}, \Lambda().\{m(a) = e_1\}))$$

The export mechanism is implemented using sequences of tags to control the availability of bindings in records. Each class contains a unique sequence of tags, ω, when c_1 inherits from c_2, the tag sequence of c_2 will be a prefix of the sequence of c_1. An object is an environment

$$(\omega_1 \mapsto e_1) \oplus (\omega_2 \mapsto e_2) \oplus \ldots (\omega_n \mapsto e_n)$$

where each ω_i is a tag sequence and each e_i is an environment which associates names with values. Each environment e is associated with a tag sequence for the class to which the names in e have been exported. A reference to a name n in an object o with respect to a client c is legal when the tag sequence of c is associated with an environment in o.

$$\begin{array}{l}
\mathbf{let}\ \ \mathit{ref}(o, n, C(_, _, \omega, _)) = \\
\qquad \mathbf{if}\ \ \omega \in \mathit{dom}(o) \\
\qquad \mathbf{then}\ \ ((o \bullet \omega) \bullet n) \\
\qquad \mathbf{else}\ \ \mathit{halt}(\text{“}\mathit{illegalreference}\text{”})
\end{array}$$

The message passing operator *send* is trivially modified to use *ref*. Two objects are combined using the operator $\_ \otimes \_$ which coalesces environments which have the same tag sequence.

$$\begin{array}{rl}
\mathbf{let}\ \ o_1 \otimes o_2 = & (\mathit{lh}\ \ (\oplus)\ \ f\ \ \{\}\ \ (\mathit{dom}(o_1) \cap \mathit{dom}(o_2))) \oplus \\
& (\mathit{lh}\ \ (\oplus)\ \ (o_1 \bullet)\ \ \{\}\ \ (\mathit{dom}(o_1) - \mathit{dom}(o_2))) \oplus \\
& (\mathit{lh}\ \ (\oplus)\ \ (o_2 \bullet)\ \ \{\}\ \ (\mathit{dom}(o_2) - \mathit{dom}(o_1)))
\end{array}$$

Given a single tag sequence, ω, the operator G produces the set of all tag sequences which have ω as a prefix. When an environment e is exported to a class c with a tag sequence ω, it will also be exported to all the subclasses of c which are associated with tag sets from $G(\omega)$. The operator *export* takes two arguments, an environment e and a set of tag sequences Ω and produces an object,

$$\mathbf{let}\ \ \mathit{export}(e, \Omega) = \mathit{lh}\ \ (\oplus)\ \ (\lambda\omega.\mathit{lh}\ \ (\oplus)\ \ (\mapsto e)\ \ \{\}\ \ G(\omega))\ \ \{\}\ \ \Omega$$

A class is redefined to have two extra components $C(g, \sigma, \omega, \chi)$ where ω is the unique tag sequence for the class and χ is the export specification for the class. An export specification is an environment which maps the names which are defined by the class to sets containing the tag sequences of classes to which the names are exported. The operator *exportdef* is defined as follows,

$$\mathbf{let}\ \ \mathit{exportdef}(n, C(\_, \_, \omega_1, \_), C(g, \sigma, \omega_2, \chi)) = C(g, \sigma, \omega_2, \chi \oplus (n \mapsto ((\chi \bullet n) \cup \{\omega_1\})))$$

An extension is defined to mean

$$
\begin{array}{l}
C(x, \sigma, \tau, \chi) \\
\quad \textbf{where} \\
\quad\quad \sigma = (\text{“}i\text{”} \mapsto \tau_1) \oplus (\text{“}f\text{”} \mapsto \tau_2) \\
\quad\quad \chi = (\text{“}i\text{”} \mapsto \{\}) \oplus (\text{“}f\text{”} \mapsto \{\}) \\
\quad\quad \textbf{rec}\ \ x(a_1, a_2, \ldots, a_n) = ((R(\lambda().())) - (R(x)), m) \\
\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad m(\mathit{super}, \mathit{inner}, T, \sigma, \chi)(\mathit{self}) = (o, T \cup \{\sigma \bullet \text{“}i\text{”}, \sigma \bullet \text{“}f\text{”}\}) \\
\quad\quad\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad\quad\quad o = \ (\textbf{if}\ \ \sigma \bullet \text{“}i\text{”} \in T\ \ \textbf{then}\ \ \{\}\ \ \textbf{else}\ \ \mathit{export}(\text{“}i\text{”} \mapsto e_1, \chi \bullet \text{“}i\text{”})) \otimes \\
\quad\quad\quad\quad\quad\quad\quad\quad (\textbf{if}\ \ \sigma \bullet \text{“}f\text{”} \in T\ \ \textbf{then}\ \ \{\}\ \ \textbf{else}\ \ \mathit{export}(\text{“}f\text{”} \mapsto \lambda j.e_2, \chi \bullet \text{“}f\text{”}))
\end{array}
$$

The *class* combinator concatenates the two sequences of tags and the export specifications in the resulting class

$$
\begin{array}{l}
\textbf{let}\ \ \mathit{class}(C(g_1, \sigma_1, \omega_1, \chi_1), C(g_2, \sigma_2, \omega_2, \chi_2)) = C(g, (\sigma_1, \sigma_2), \omega_1\omega_2, \chi_1 \oplus \chi_2) \\
\quad\quad \textbf{where} \\
\quad\quad\quad\quad g(v_1, v_2) = \textbf{let}\ \ (a_1, m_1) = c_1(v_1)\ \ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let}\ \ (a_2, m_2) = (a_1 \hookrightarrow g_2)(v_2)\ \ \textbf{in} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let}\ \ m(o_1, o_2, T, (\sigma_1, \sigma_2), \chi)(o_3) = \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{let rec}\ \ \ (o_4, T) = m_1(o_1, o_2 \oplus o_5, T, \sigma_1, \chi)(o_3) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (o_5, T) = m_2(o_1 \oplus o_4, o_2, T, \sigma_2, \chi)(o_3) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ \ (o_4 \otimes o_5, T) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in}\ \ (a_1 \oplus a_2, m)
\end{array}
$$

The class instantiation operator is modified to feed the export specification to the object generator,

$$
\textbf{let}\ \ \mathit{new}(C(g, \sigma, \_, \chi), v) = \mathbf{Y}(\mathbf{1}(\mathit{2nd}(g(v)))(\{\}, \{\}, \sigma, \chi))
$$

Finally, the null class is extended to contain the prefix of all tag sequences, [], and an empty export specification,

$$
\begin{array}{l}
\textbf{let}\ \ \mathit{nullclass} = C(g, \{\}, [], \{\}) \\
\quad\quad \textbf{where} \\
\quad\quad\quad\quad g() = (\{\}, \lambda(\_, \_, T\_, \_)(\_).(\{\}, T))
\end{array}
$$

11 Layer 8: Late binding

The eighth layer defines the distinction between *eager* binding and *late* binding which are rules affecting the association between the usage occurrence of an identifier and the defining occurrence of the same identifier. Many languages define that the innermost textual defining occurrence of an identifier will be used to determine the value of the identifier when it is used in expressions contained within the block headed by the definition. This rule is used by Pascal [31]. It is not the only rule which governs identifiers and languages such as Lisp [30] and ML [16] have their own rules which are very different from textual nesting.

In order to discuss the difference between eager and late identifier binding it is necessary to remove the restriction that names which are defined by a class must be referenced with respect to *self* in the bodies of methods which are defined by the class. Consider the following class

$$\mathbf{let}\ \ c_1 = \mathit{exportdef}(\text{“}m_2\text{”}, \mathit{nullclass}, \mathit{class}(\mathit{nullclass}, \Lambda().\{m_1(a) = e_1; m_2(a) = m_1(a)\}))$$

in which the method m_2 refers to the occurrence of a free identifier m_1 which is defined to mean the method m_1 which is also defined by the class. The distinction between eager and late binding becomes clear when c_1 is used as a superclass where the extension gives a new definition for the method m_1

$$\mathbf{let}\ \ c_2 = \mathit{class}(c_1, \Lambda().\{m_1(a) = e_3\})$$

Now there is a question as to the outcome of the expression

$$(\mathit{ref}(\mathit{new}(c_2, ()), \text{”}m_2\text{”}, \mathit{nullclass}))(e_4)$$

because the defining occurrence for the use of m_1 in the body of m_2 could be that given by c_1 in which case the binding is said to be *eager* or could be given by c_2 in which case it is said to be *late*.

11.1 Eager binding

Eager binding for class definitions means that a usage occurrence refers to the definition which is given in the extension or the most recent superclass. The defining occurrence of an identifier in a class can never be affected by subsequent redefinitions in subclasses. An extension is modified to mean

$$\begin{array}{l}
C(x, \sigma, \tau, \chi) \\
\quad \mathbf{where} \\
\qquad \sigma = (\text{“}i\text{”} \mapsto \tau_1) \oplus (\text{“}f\text{”} \mapsto \tau_2) \\
\qquad \chi = (\text{“}i\text{”} \mapsto \{\}) \oplus (\text{“}f\text{”} \mapsto \{\}) \\
\qquad \mathbf{rec}\ \ x(a_1, a_2, \ldots, a_n) = ((R(\lambda().())) - (R(x)), m) \\
\qquad\quad \mathbf{where} \\
\qquad\qquad m(\mathit{super}, \mathit{inner}, T, \sigma, \omega)(\mathit{self}) = (o, T \cup \{\tau_1, \tau_2\}, \omega\tau) \\
\qquad\qquad\quad \mathbf{where\ rec} \\
\qquad\qquad\qquad o = \ \ \mathbf{open}\ \ (\mathit{super} \otimes o) \bullet \omega\tau \\
\qquad\qquad\qquad\qquad \mathbf{in}\ (\mathbf{if}\ \ \sigma \bullet \text{“}i\text{”} \in T\ \ \mathbf{then}\ \ \{\}\ \ \mathbf{else}\ \ \mathit{export}(\text{“}i\text{”} \mapsto e_1, \chi \bullet \text{“}i\text{”})) \otimes \\
\qquad\qquad\qquad\qquad\quad (\mathbf{if}\ \ \sigma \bullet \text{“}f\text{”} \in T\ \ \mathbf{then}\ \ \{\}\ \ \mathbf{else}\ \ \mathit{export}(\text{“}f\text{”} \mapsto \lambda j.e_2, \chi \bullet \text{“}f\text{”}))
\end{array}$$

The definitions which are given by the extension and the value *super* are opened for the scope of the extension definitions. It is important that only those names which have been exported to the extension or a superclass are placed in scope so the tag sequence $\omega\tau$ is used. ω is an extra parameter which is supplied to each extension and is the tag sequence of the superclass; in order to pass ω round, an m-function produces a triple (o, T, ω) where o is the object, T is the set of tags for all definitions which have been allocated and ω is the tag set for the class which produced o.

The *class* constructor is extended to include the extra ω component which is supplied and returned to each m-function.

$$
\begin{array}{l}
\textbf{let } \ \mathit{class}(C(g_1, \sigma_1, \omega_1, \chi_1), C(g_2, \sigma_2, \omega_2, \chi_2)) = C(g, (\sigma_1, \sigma_2), \omega_1\omega_2, \chi_1 \oplus \chi_2) \\
\quad \textbf{where} \\
\qquad g(v_1, v_2) = \ \textbf{let } (a_1, m_1) = g_1(v_1) \ \textbf{in} \\
\qquad\qquad\qquad\quad \textbf{let } (a_2, m_2) = (a_1 \hookrightarrow g_2)(v_2) \ \textbf{in} \\
\qquad\qquad\qquad\quad \textbf{let } m(o_1, o_2, T, (\sigma_1, \sigma_2), \chi, \omega)(o_3) = \\
\qquad\qquad\qquad\qquad\quad \textbf{let rec } \ (o_4, T, \omega) = m_1(o_1, o_2 \otimes o_5, T, \sigma_1, \chi, \omega)(o_3) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (o_5, T, \omega) = m_2(o_1 \otimes o_4, o_2, T, \sigma_2, \chi, \omega)(o_3) \\
\qquad\qquad\qquad\qquad\quad \textbf{in } (o_4 \otimes o_5, T, \omega) \\
\qquad\qquad\qquad\quad \textbf{in } (a_1 \oplus a_2, m)
\end{array}
$$

The class instantiation operator *new* is extended to pass in the empty tag sequence []

$$\textbf{let } \ \mathit{new}(C(g, \sigma, \_, \chi), v) = \mathbf{Y}(\mathbf{1}(\mathit{2nd}(g(v)))(\{\}, \{\}, \sigma, \chi, []))$$

The null class has an m-function which produces the null object and preserves the T set and the current ω tag sequence,

$$
\begin{array}{l}
\textbf{let } \ \mathit{nullclass} = C(g, \{\}, [], \{\}) \\
\quad \textbf{where} \\
\qquad g() = (\{\}, \lambda(\_, \_, T, \_, \_, \omega).(\{\}, T, \omega))
\end{array}
$$

11.2 Late binding

Late binding is implemented in much the same way as eager binding except that the *self* object is opened for the scope of the methods in the extension rather than the *super* object. This means that the latest definitions for any given name shadow all the others for all method bodies. A problem occurs if the *self* object is opened for the scope of the all method bodies because a given extension should only see the up-to-date definitions for the names which it or one of its superclasses defines. If the *self* object is opened then all names which are defined by all subclasses will be available, even though many of them are names not defined by the extensions or its superclasses. The solution to this problem is for each class to contain the set of names which it defines. The environment restriction operator e/S is used to restrict the domain of the *self* object *before* it is opened for the scope of an extension's method bodies. As before, the tag sequence ω ensures that only those definitions which have been exported to the extension are opened for the scope of the method bodies.

An extension is modified to mean,

$$
\begin{array}{l}
C(x,\sigma,\tau,\chi) \\
\quad \textbf{where} \\
\qquad \sigma = (\text{“}i\text{”} \mapsto \tau_1) \oplus (\text{“}f\text{”} \mapsto \tau_2) \\
\qquad \chi = (\text{“}i\text{”} \mapsto \{\}) \oplus (\text{“}f\text{”} \mapsto \{\}) \\
\qquad \textbf{rec}\ \ x(a_1,a_2,\ldots,a_n) = ((R(\lambda().())) - (R(x)), m) \\
\qquad\quad \textbf{where} \\
\qquad\qquad m(\mathit{super},\mathit{inner},T,\sigma,\omega,S)(\mathit{self}) = (o, T \cup \{\tau_1,\tau_2\}, \omega\tau, S \cup \{\text{“}i\text{”},\text{“}f\text{”}\}) \\
\qquad\qquad\quad \textbf{where rec} \\
\qquad\qquad\qquad o = \ \textbf{open}\ \ (\mathit{self} \bullet \omega\tau)/(S \cup \{\text{“}i\text{”},\text{“}f\text{”}\}) \\
\qquad\qquad\qquad\quad \textbf{in}\ (\textbf{if}\ \ \sigma \bullet \text{“}i\text{”} \in T\ \ \textbf{then}\ \ \{\}\ \ \textbf{else}\ \ \mathit{export}(\text{“}i\text{”} \mapsto e_1, \chi \bullet \text{“}i\text{”}))\otimes \\
\qquad\qquad\qquad\qquad (\textbf{if}\ \ \sigma \bullet \text{“}f\text{”} \in T\ \ \textbf{then}\ \ \{\}\ \ \textbf{else}\ \ \mathit{export}(\text{“}f\text{”} \mapsto \lambda j.e_2, \chi \bullet \text{“}f\text{”}))
\end{array}
$$

A class produces an m-function which takes an extra S argument and produces a result (o,T,ω,S) where o is the object, T is the tag set, ω is the tag sequence and S is the set of names which have been defined in o. The *class* combinator is extended to produce a new class with all the defined names of its argument classes and to produce an m-function which threads the S argument through both classes.

$$
\begin{array}{l}
\textbf{let}\ \ \mathit{class}(C(g_1,\sigma_1,\omega_1,\chi_1),C(g_2,\sigma_2,\omega_2,\chi_2)) = C(g,(\sigma_1,\sigma_2),\omega_1\omega_2,\chi_1 \oplus \chi_2) \\
\quad \textbf{where} \\
\qquad g(v_1,v_2) = \ \textbf{let}\ \ (a_1,m_1) = g_1(v_1)\ \ \textbf{in} \\
\qquad\qquad\qquad\quad \textbf{let}\ \ (a_2,m_2) = (a_1 \hookrightarrow g_2)(v_2)\ \ \textbf{in} \\
\qquad\qquad\qquad\quad \textbf{let}\ \ m(o_1,o_2,T,(\sigma_1,\sigma_2),\chi,\omega,S)(o_3) = \\
\qquad\qquad\qquad\qquad \textbf{let rec}\ \ \ (o_4,T,\omega,S) = m_1(o_1,o_2 \otimes o_5,T,\sigma_1,\chi,\omega,S)(o_3) \\
\qquad\qquad\qquad\qquad\qquad\qquad (o_5,T,\omega,S) = m_2(o_1 \otimes o_4,o_2,T,\sigma_2,\chi,\omega,S)(o_3) \\
\qquad\qquad\qquad\qquad \textbf{in}\ \ (o_4 \otimes o_5,T,\omega,S) \\
\qquad\qquad\qquad\quad \textbf{in}\ \ (a_1 \oplus a_2, m)
\end{array}
$$

The class instantiation operator *new* is extended to pass in the empty set of names,

$$\textbf{let}\ \ \mathit{new}(C(g,\sigma,\_,\chi),v) = \mathbf{Y}(\mathbf{1}(\mathit{2nd}(g(v)))(\{\},\{\},\sigma,\chi,[],\{\}))$$

$$
\begin{array}{l}
\textbf{let}\ \ \mathit{nullclass} = C(g,\{\},[],\{\}) \\
\quad \textbf{where} \\
\qquad g() = (\{\},\lambda(\_,\_,T,\_,\_,\omega,S) = (\{\},T,\omega,S))
\end{array}
$$

12 Layer 9: Inclusion polymorphism

The ninth and final layer defines “inclusion polymorphism” which is a brand of polymorphism occurring in OOPLs. A value is *polymorphic* when it can be viewed as having more than one type. For example the list [] is polymorphic because it can be viewed as the empty list of integers and the empty list of character strings. Functions are said to be polymorphic when they can be applied to more than one type of argument or return more than one type of result, for example the identity function can be viewed as taking and returning integers or characters.

An object in an OOPL can be said to be polymorphic because it can be viewed as being composed of many different objects, each of which contributes some of the definitions in the entire object. Furthermore, there is a relation between some of these objects due to the subclass relationships on the classes which were used to create them. Since each class contains all the definitions of its superclasses, there is an inclusion relation between some of the subobjects of each class instance. An object may be viewed as any one of its subobjects. When the subobjects are drawn from the set which are related by being instances of classes then this is termed "inclusion polymorphism".

Each class is extended with a new tag which is unique to the class. An object is modified to be a pair (e_1, e_2) where e_1 is the environment mapping tags to collections of definitions as given in the previous section. e_2 is an environment mapping class tags to objects which are the particular view of the object as an instance of the class which owns the tag. The object may be viewed as an instance of any of the classes whose tags are bound in e_2. Furthermore, when an object is viewed as an instance of such a class then the object is *projected* to become an instance of the class by selecting the value bound to the tag in e_2.

A class is a value

$$c = C(g, \sigma, \omega, \chi, \tau)$$

where g, σ, ω and χ are the generator, sharing signature, export tag sequence and export specification as described previously and τ is a tag which is unique to the class and is used to project instances of any subclass of c to have exactly those names which are described by c.

The operator *proj* is used to project an object to be an instance of a particular class. If the projection is successful, the instance of the class is returned otherwise an exception is raised.

$$\begin{array}{l} \textbf{let} \;\; \mathit{proj}(C(\_, \_, \_, \_, \tau))(\_, e) = \\ \qquad \textbf{if} \;\; e \bullet \tau = \epsilon \\ \qquad \textbf{then} \;\; \mathit{halt}(\text{“}\mathit{illegaltype}\text{”}) \\ \qquad \textbf{else} \;\; e \bullet \tau \end{array}$$

The *ref* operator is modified to extract the first component of an object,

$$\begin{array}{l} \textbf{let} \;\; \mathit{ref}((o, \_), n, C(\_, \_, \omega, \_, \_)) = \\ \qquad \textbf{if} \;\; \omega \in \mathit{dom}(o) \\ \qquad \textbf{then} \;\; ((o \bullet \omega) \bullet n) \\ \qquad \textbf{else} \;\; \mathit{halt}(\text{“}\mathit{illegalreference}\text{”}) \end{array}$$

A method is a function which provides a type for its formal parameter

$$\lambda(i : e_1).e_2$$

the type e_1 is an expression which will evaluate to produce a class. The value of the formal parameter is projected to have the class as a type. The method is translated to have the meaning

$$((\lambda i.e_2) \circ (\mathit{proj}(e_1))$$

An extension is modified to mean,

$$
\begin{array}{l}
C(x, \sigma, \tau_1, \chi, \tau_2) \\
\quad \textbf{where} \\
\qquad \sigma = (\text{“}i\text{”} \mapsto \tau_3) \oplus (\text{“}f\text{”} \mapsto \tau_4) \\
\qquad \chi = (\text{“}i\text{”} \mapsto \{\}) \oplus (\text{“}f\text{”} \mapsto \{\}) \\
\qquad \textbf{rec}\ \ x(a_1, a_2, \ldots, a_n) = ((R(\lambda().())) - (R(x)), m) \\
\qquad\quad \textbf{where} \\
\qquad\qquad m(\mathit{super}, \mathit{inner}, T, \sigma, \omega, S)(\mathit{self}) = (o, T \cup \{\tau_1, \tau_2\}, \omega\tau_1, S \cup \{\text{“}i\text{”}, \text{“}f\text{”}\}) \\
\qquad\qquad\quad \textbf{where rec}\ \ o = (e, \tau_2 \mapsto o) \\
\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\quad e = \ \ \textbf{open}\ \ (\mathit{1st}(\mathit{self} \bullet \omega\tau))/(S \cup \{\text{“}i\text{”}, \text{“}f\text{”}\}) \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\textbf{if}\ \sigma \bullet \text{“}i\text{”} \in T\ \ \textbf{then}\ \{\}\ \textbf{else}\ \ \mathit{export}(\text{“}i\text{”} \mapsto e_1, \chi \bullet \text{“}i\text{”})) \otimes \\
\qquad\qquad\qquad\qquad\qquad (\textbf{if}\ \sigma \bullet \text{“}f\text{”} \in T\ \ \textbf{then}\ \{\}\ \textbf{else}\ \ \mathit{export}(\text{“}f\text{”} \mapsto \lambda j.e_2, \chi \bullet \text{“}f\text{”}))
\end{array}
$$

The *class* constructor is modified to take three arguments which are the super-class, the extension class and a tag which becomes the tag of the new class.

$$
\begin{array}{l}
\textbf{let}\ \ \mathit{class}(C(g_1, \sigma_1, \omega_1, \chi_1, \tau_1), C(g_2, \sigma_2, \omega_2, \chi_2, \tau_2), \tau_3) = C(g, (\sigma_1, \sigma_2), \omega_1\omega_2, \chi_1 \oplus \chi_2, \tau_3) \\
\quad \textbf{where} \\
\qquad g(v_1, v_2) = \ \ \textbf{let}\ \ (a_1, m_1) = g_1(v_1)\ \ \textbf{in} \\
\qquad\qquad\qquad\quad\ \ \textbf{let}\ \ (a_2, m_2) = (a_1 \hookrightarrow g_2)(v_2)\ \ \textbf{in} \\
\qquad\qquad\qquad\quad\ \ \textbf{let}\ \ m(o_1, o_2, T, (\sigma_1, \sigma_2), \omega, S)(o_3) = \\
\qquad\qquad\qquad\qquad\qquad \textbf{let rec}\ \ (o_4, T, \omega, S) = m_1(o_1, o_2 \oplus o_5, T, \sigma_1, \omega, S) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (o_5, T, \omega, S) = m_2(o_1 \oplus o_4, o_2, T, \sigma_2, \omega, S) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad o = ((\mathit{1st}(o_4)) \otimes (\mathit{1st}(o_5)), (\tau_3 \mapsto o) \oplus (\mathit{2nd}(o_4)) \oplus (\mathit{2nd}(o_5))) \\
\qquad\qquad\qquad\qquad\quad \textbf{in}(o, T, \omega, S) \\
\qquad\qquad\qquad\quad\ \ \textbf{in}\ \ (a_1 \oplus a_2, m)
\end{array}
$$

The class instantiation operator is extended to ignore the extra tag component,

$$\textbf{let}\ \ \mathit{new}(C(g, \sigma, \_, \chi, \_), v) = \mathbf{Y}(\mathbf{1}(\mathit{2nd}(g(v)))(\{\}, \{\}, \sigma, \chi, [], \{\}))$$

The null class is extended to have a unique tag τ_0,

$$
\begin{array}{l}
\textbf{let}\ \ \mathit{nullclass} = C(g, \{\}, [], \{\}, \tau_0) \\
\qquad \textbf{where} \\
\qquad\qquad g() = (\{\}, \lambda(\_, \_, T, \_, \_, \omega, S) = (\{\}, T, \omega, S))
\end{array}
$$

13 Conclusion and further work

This paper has described a simple functional language which includes special primitives for Object Oriented Programming. A functional language is particularly useful for expressing such programming paradigms because of its uniformity and simplicity. An OOPL has been constructed by identifying the kernel features which are required, classes, extensions, objects, the *class* constructor, the class instantiation operator *new* and the message passing operator *send*. OOPLs of increasing sophistication have been constructed by layering language mechanisms in terms of the kernel features. The resulting language is characteristic of a broad range of currently used OOPLs and the basic ideas and constructions provide a basis for a wide variety of languages which will allow the OOPL design space to be mapped out in a systematic fashion.

Although the language which is constructed in this paper is representative of many currently available OOPLs, there are other features which can be added as extra layers including metaclasses [19], [5] [7], generic classes [23], multimethods [26] and different multiple inheritance strategies [2]. We are confident that the functional notation and characteristic primitive operations involving environment reification, variable locations and fixed points will support all of these extra features. In addition to the analysis of existing OOPL features, the functional notation presented in this paper presents the opportunity to invent new OOPL features and to merge or generalise existing ones, within a unifying framework.

Further work is required to analyse the implementation mechanisms required to provide the characteristic primitive operations of OOPLs in a programming language. A naive implementation of environment reification can produce a very inefficient system, [25] discusses some of the issues.

References

[1] Agha, G. 1990 The Structure and Semantics of Actor Languages. *In Foundations of Object-Oriented Languages.* LNCS 489.

[2] Cardelli, L. 1984 A Semantics Of Multiple Inheritance. *In The Semantics Of Data Types.* LNCS 173.

[3] Cardelli, L. and Wegner, P. 1985 On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys, 17* 4.

[4] Cardelli, L. and Mitchell, J. 1991 Operations on records. *Math. Struct. in Comp. Science 1*

[5] Clark, A. N. 1994 Metaclasses and Reflection in Smalltalk. Unpublished research report, GEC-Marconi Research Centre, Chelmsford, Essex.

[6] Clark, A. N. 1994 Semantic Primitives for OOPLS PhD Thesis, Queen Mary and Westfield College, University of London.

[7] Cointe, P. 1987 Metaclasses are First Class: The ObjVLisp Model. *OOPSLA*

[8] Cook, W. 1989 A Proposal For Making Eiffel Type-safe. *The Computer Journal 32* 4.

[9] Cook, W. *et. al.* 1990 Inheritance Is Not Subtyping. *POPL 17.*

[10] Cook, W. 1989 A denotational Semantics Of Inheritance. PhD Thesis, Brown Univ. 1989.

[11] Cook, W. and Bracha, G. 1990 Mixin Based Inheritance. *OOPSLA*.

[12] Cook, W. 1989 A Denotational Semantics Of Inheritance and its Correctness. *OOPSLA*.

[13] Cook, W. 1990 Object-Oriented Programming Versus Abstract Data Types. *Proc. Foundations of Object-Oriented Languages.* LNCS 489.

[14] Field, A. and Harrison, P. 1988 *Functional Programming.* Addison-Wesley.

[15] Goldberg, A. *et. al.* 1983 *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

[16] Harper, R. et. al. 1986 Standard ML. *Edinburgh research report ECS-LFCS-86-2.*

[17] Jagannathan, S. 1994 Metalevel Building Blocks for Modular Systems. *ACM TOPLAS 16* 3.

[18] Jagannathan, S. and Agha, G. 1992 A Reflective Model Of Inheritance. *ECOOP.*

[19] Kiczales, G. *et al.* 1991 *The Art of the Metaobject Protocol.* MIT Press.

[20] Kristensen, B. *et. al.* 1987 The BETA Programming Language. *In Research Directions In Object Oriented Programming ed Bruce Shriver and Peter Wegner.* The MIT Press.

[21] Landin, P. 1966 The Next 700 Programming Languages. *Comm. of the ACM 9* 3.

[22] Landin, P. 1965 A Correspondence between Algol 60 and Church's lambda notation: Part I. *Comm. of the ACM 8* 2.

[23] Meyer, B. 1986 Genericity Vs. Inheritance. *OOPSLA*.

[24] Meyer, B. 1992 *Eiffel The Language.* Prentice Hall Object Oriented Series.

[25] Miller, J. and Rozas, G. 1991 Free Variables and First-Class Environments. *Lisp and Symbolic Computation 4.*

[26] Mugridge, W. *et. al.* Multi-Methods in a Statically-Typed Programming Language. *ECOOP.*

[27] Schaffert, C. *et. al.* 1986 An Introduction To Trellis/Owl. *OOPSLA.*

[28] Stroustroup, B. *et. al. The Annotated C++ Reference Manual.* Addison Wesley.

[29] Ungar, D. and Smith, R. 1987 SELF: The Power of Simplicity. *OOPSLA also in The SELF Papers.*

[30] Steele, G. 1984 *Common Lisp reference manual* Digital Press.

[31] Welsh, J. and Elder, J. 1982 *Introduction to Pascal.* Prentice-Hall International Series in Computer Science.