Semantic Primitives For
Object-Oriented Programming Languages

Anthony Neil Clark

Submitted for the degree of Doctor of Philosophy in Computer Science

Department of Computer Science
Queen Mary and Westfield College
University of London
Mile End Road
London
E1 4NS

Abstract

The λ-calculus is given an operational semantics in terms of the SECD machine. This is then extended with built-in operators and new transition rules to produce the Λ-calculus which characterises object-oriented programming languages. Although there are many languages which can be called object-oriented, they are difficult to compare because the operations which provide facilities such as inheritance and message passing are hidden behind linguistic abstractions which make the languages rigid and difficult to modify. By identifying a collection of primitive operations which underly many object-oriented languages, the task of comparing languages and designing new ones will be made easier.

The primitive characteristic features of object-oriented programming languages are identified as: locations, sharing and update, constructing cyclic terms by taking a fixed point, environment reification and installation and constructing program points using the J operator. The constructs of object-oriented languages which are analysed, taken from concrete languages such as Smalltalk, Eiffel, Beta, Self and C++, include: classes, single inheritance, multiple inheritance, message passing, run-super, run-inner, open and closed self reference, private names, prototypes, dynamic inheritance, late binding, repeated inheritance and inclusion polymorphism.

Acknowledgments

I wish to thank GEC-Marconi for their sponsorship which enabled me to do this work.
I wish to thank my colleagues, Graham Hill and Ceri Rees, for reading and commenting on earlier drafts of this thesis.
Finally, I wish to thank my supervisor, Professor Peter Landin, who patiently explained fundamental concepts of programming during many Friday afternoon tutorials.

Contents

List of Figures

Chapter 1

Introduction

Object-oriented programming first emerged as part of the Simula family of programming languages around 1965. Since then object orientation has steadily developed a following in terms of both programming languages and design methodologies; until in 1989 it was reported that there was 88 distinct object-oriented languages [63]. Even though there has obviously been much effort directed towards the development of object-oriented artifacts, there has been relatively little development of a precise understanding of the fundamental aspects of object-oriented language design. Part of the reason for this is that there seems to be no consensus of opinion as to what exactly comprises such a language, and since these languages have emerged through natural selection, there is a difficulty in disentangling existing programming language concepts from the genuinely new object-oriented concepts with which they co-exist.

This thesis precisely describes the main features of object-oriented programming languages. This will be done by showing how these features can be represented in a simple functional language which has a precisely described operational semantics. The features will be given as a collection of simple programming constructs and operators which can be combined together to achieve many different object-oriented variants. The resulting functional language may be used to analyse and compare object-oriented languages, extend existing languages and assist in the design of completely new languages.

1.1 Ingredients of an Object-Oriented Language

It is notoriously hard to precisely describe what is meant by the term object-oriented programming language. It is commonly accepted that objects are a combination of operations and data. The data acts as state for the operations to access and update and the object interacts with the rest of the program through an interface of names. Less certain is whether all the operations are in the interface of an object or whether the data forms part of the interface at all. Can the object refer to itself as part of an operation? There is no consensus whatsoever as to how the object comes into existence, is it a "one-off", a copy of another object or cut from some raw material by instantiating a class? If classes are present then what do they define and how are new classes constructed? Inheritance is often cited as a major component of an object-oriented language, but different languages exhibit wild variations as to the effect of inheritance. There is single inheritance and multiple inheritance; classes may be modified during inheritance by renaming their components, redefining their components or removing their components altogether. The following quotes are taken from publications written by many of the leading researchers into object-oriented languages:

> "The term **object-oriented** is derived from the **object** concept of the Simula67 programming language. In that language the execution of a computer program is organised as the joint execution of a (possibly variable) collection of objects. The collection as a whole is represented as a system object and objects sharing a common structure are said to constitute a **class** described in the program by a common **class declaration**." [57]
>
> "The essence of object-oriented programming is *procedural data abstraction*, in which procedures are used to represent data and procedural interfaces provide information hiding and abstraction. This technique is complementary to abstract data types, in which concrete algebras are used to represent data, and type abstraction provides information hiding." [24]
>
> "A language is "object-oriented" when it offers a set of features, [...], called "encapsulation" "inheritance", and "object identity"; these features help following the "object-oriented programming methodology", i.e. building a software system as a set of communicating objects" grouped in "classes"." [30]
>
> "[...] the essence of object-oriented programming is the encapsulation of data and operations in objects and the protection of individual objects against each other." [3]
>
> "A language will be called object-based if it provides linguistic support for objects, where objects have the following properties:
>
> **Object** An object has a set of operations and a state that remembers the effect of the operations. Objects communicate by sending each other messages to perform operations. Objects may be contrasted to functions which have no memory. Whereas function values are completely determined by their arguments, an object may learn from experience, its reaction to an operation being determined by its invocation history.
>
> A language will be called object-oriented if it is object-based and additionally requires that objects have classes and classes have inheritance:
>
> *object-oriented = objects + object classes + class inheritance*
>
> Classes and inheritance have the following properties:
>
> **Class** Classes specify an interface of operations. They may serve as templates from which objects with the given interface may be created. They may have "class bodies" that specify the implementation of the operations in the class interface and the representation of the state. [...]
>
> **Class Inheritance** Class inheritance is a mechanism for composing the interface of one or more inherited classes with the interface of the inheriting class. Inherited classes are called super-classes and inheriting classes are called

> sub-classes. Operations of a super-class can be accessed by its sub-classes providing there are no name conflicts. Occurrences of "self" in an operation definition refer to the object on behalf of which the operation is being executed [...]" [74]

> "Because the term "object-oriented" has been used to describe various techniques - even Ada has been claimed to be an object-oriented language! - it is useful to distinguish the various steps that lead to true object-orientedness. Needless to say, only systems that reach the last step are, in my opinion, worthy of the name.
>
> **Level 1** (*Object-based modular structure*): Systems are modularised on the basis of their data structures.
>
> **Level 2** (*Data abstraction*): Objects should be described as implementations of abstract data types.
>
> **Level 3** (*Automatic memory management*): Unused objects should be deallocated by the underlying language system, without programmer intervention.
>
> **Level 4** (*Classes*): Every non-simple type is a module, and every high-level module is a type.
>
> **Level 5** (*Inheritance*): A class may be defined as an extension or restriction of another.
>
> **Level 6** (*Polymorphism and dynamic binding*): Program entities should be permitted to refer to object of more than one class, and operations should be permitted to have different realizations in different classes.
>
> **Level 7** (*Multiple and repeated inheritance*): It should be possible to declare a class as heir to more than one class, and more than once to the same class." [53]

> "...This is where object-oriented techniques come into their own. These techniques work by (i) defining a set of object types and operations on them, which can support not just a single behaviour, but a space or region of behaviours – this is commonly called a protocol; (ii) defining a default behaviour, a single point in the region, in terms of the protocol – this is the role of the default classes and methods; and (iii) making it possible to effect incremental adjustments from the default behaviour to other points in the region – this is the role of inheritance and specialisation." [41]

> "What is object-oriented programming? My guess is that object-oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products by supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no-one will know just what it is." [62]

Two observations can be made of the quotes given above, firstly there is no accepted single precise definition of what the term object-oriented programming means and secondly many of the definitions contain common elements such as classes, inheritance, interfaces etc. One suggestion is that object-oriented languages represent an amalgam of features all of which are worthy of study in their own right, i.e. one has to understand the parts in order to sensibly talk about the whole.

This thesis will not attempt to give a unifying definition for object-oriented programming, rather it will analyse certain characteristic features of object-oriented languages which crop up time and again in definitions such as those given above. These features may be combined in different ways in order to exhibit the required variations in object-oriented language definitions.

1.2 Guide to Thesis

This thesis deals with the design and implementation of object-oriented interpretive mechanisms and makes a contribution to the following:

- Identifying the underlying primitive operations which characterise object-oriented programming languages. This thesis provides a unifying open framework within which such primitives can co-exist.
- The design and implementation of object-oriented interpretive mechanisms. There is currently no standard representation for describing such mechanisms, for example the steps which are taken during class instantiation or method lookup.
- The definition of basic representations for objects, classes and methods and a variety of related operations for field reference, message passing, class instantiation and inheritance. Many examples of these features are given in the thesis which can be used as the basis for experimentation with a wide variety of object-oriented mechanisms.

1.3 Thesis Overview

Church's λ-notation in conjunction with an operational semantics in terms of the SECD machine provides a convenient representation for the semantics of a broad class of programming languages. Categories of programming language may be captured by extending the λ-notation with new types of expression and the SECD machine with new types of value, built-in operators and new transition rules. The extensions are characteristic of the language category if they facilitate translations from special purpose language descriptions to the single language described by the extended λ-notation and SECD machine.

The thesis is that object-oriented programming languages are characterised by providing the following extensions to λ-notation and SECD machine: locations and side effects which are used for the private state of an object; first class environments, environment reification and installation which are used for object representation; cyclic data values constructed using the Y operator which are used to produce self referential objects; and program points constructed using the J operator which are used to implement object oriented exception mechanisms.

Many object-oriented features can be represented using a simple functional programming language which has these characteristic extensions. Furthermore, because the functional language does not hide these primitive features behind unnecessary linguistic sugar, it is easy to combine the primitives in novel ways in order to experiment with new programming language facilities. The extended λ-notation will be referred to as Λ-notation (or Λ-programs), its state transition machine as the Λ-machine and the resulting language (meaning both syntax and semantics) as the Λ-calculus.

Object-oriented programming languages rely upon the representation of classes, objects and methods, and the mechanisms which achieve class instantiation and method lookup, as their characteristic features. Such languages differ with respect to the ways in which they implement these features. The thesis is supported by using the Λ-calculus to define a collection of programming features which are typical of many current object-oriented languages. Each feature is based upon currently used programming languages such as Smalltalk and Eiffel.

The thesis is structured as follows:

- Chapter 2 surveys a number of object-oriented programming languages and identifies the main features which they have in common.
- Chapter 3 defines the Λ-calculus. This is a simple functional programming language based upon the SECD machine which has been extended with primitive facilities for object-oriented programming.
- Chapter 4 analyses the features identified in chapter 2 and shows how they are implemented in the Λ-calculus.
- Chapter 5 reviews and analyses the main points of the thesis and suggests areas for further work.

1.4 Relationship to Other Work

This section describes how the results which are presented in this thesis relate to other work in the area of object-oriented semantics and programming language analysis.

The work of Cook *et al.* [19] [20] [21] [22] [23] [24] has been very influential with respect to this work, especially describing inheritance and self reference in terms of compositional functional abstractions over record structures, F-bounded polymorphism, the relationship between *run super* and *run inner* and *co*variance and its relationship to *contra*variance. Cook identifies a number of primitive operations for object-oriented language description but the approach used in this thesis of presenting object-oriented primitives in terms of a machine allows greater freedom to precisely capture the underlying concepts.

Agha *et al.* [39] [2], [29] and [54] have identified first class environments as being a foundational component of inheritance in object-oriented languages. None of these approaches provide the full generality of operations which are described in this thesis. [39] uses natural semantics to explain the behaviour of first class environments.

Wegner *et al.* [73] [74] [8] [75] have contributed greatly to the understanding of the components of object-oriented languages and object oriented programming. Of particular relevance is [75] which discusses the design space of such languages and provides a framework within which the results of this thesis will fit.

There is a large body of work which describes the relationship of object-oriented languages to small elegant record calculi, [7] [8] [9] [33] [10]. Whilst these systems are mainly directed at constructing type theories for object-orientation, much of which is outside the scope of this thesis, they have been used as the starting point for the representation and manipulation of the record structures which form the backbone of our representation.

There has been some work in adding object-oriented facilities to Lisp-like languages, notably [61] and [1], which rely upon the semantics of the underlying language which is used to represent object-oriented features.

Landin first described the use of the λ-calculus as the basis of a family of programming languages in [47]. This thesis uses this approach to describe object-oriented languages in terms of the primitive operations which must be added to λ in order to support object-orientation.

The representation of the operational semantics of a programming language by machine started with the SECD machine [46] and has since been used as the basis of concrete functional language implementations, for example [28], and the design of abstract machines [32].

Several researchers have defined the semantics of object-oriented languages in terms of the Scott-Strachey *denotational* semantics [23] [77] [78] [40].

A characteristic collection of object-oriented features have been represented in the Λ-calculus by analysing a cross section of programming languages. Most of these languages are in current use; they are listed here with the major sources which have been used as the definition of their features:

- Beta [42] [43] [50]
- Smalltalk [31]
- Eiffel [53] [52]
- CLOS [4] [41]
- Loops [5]
- C++ [70]
- Self [71] [12] [13]
- Trellis/Owl [64]
- ObjVLisp [17] [6] [18]
- CommonObjects [66]

Chapter 2

Survey and Analysis

The aim of this chapter is to survey a representative collection of object-oriented programming languages in order to identify a collection of characteristic object-oriented programming features. The features will be used to drive the design of primitive extensions to a simple functional programming language in chapter 3. We claim that the resulting language captures the essential characteristics of object-oriented languages and we implement the features using the functional language in chapter 4.

The languages which are analysed in this chapter have been chosen to reflect major categories of object-oriented programming language. They are as follows: Smalltalk, Eiffel, C++, BETA and Self. The aim is to identify the main features of each language and then to identify the main themes which occur in all languages.

Each language is analysed with respect to an operational model of programming languages. Such a model is particularly suited to representing issues relating to data value life-cycles, identifier scoping, associations between identifiers and values, program locations and sharing, program control and mechanisms by which data values are referenced.

The structure of this chapter is as follows: §2.1 – §2.5 gives an overview of each object-oriented language and identifies the major features; §2.6 presents an analysis and identifies common features; the result in §2.7 is a table of features which we claim to be characteristic of the object-oriented category of programming languages.

2.1 Smalltalk

The language Smalltalk was one of the first to call itself object-oriented and emerged from Xerox in the late 1970's. The language took the revolutionary step of viewing all values relating to programming a computer as *objects* and using *message passing* between objects as the main programming metaphor.

Smalltalk is a programming environment rather that just a language, including many features which are normally viewed as being beyond those which are ordinarily provided by a programming language. It is outside the scope of this thesis to deal with the facilities and libraries which support systems programming, graphics programming, meta-programming *etc.*; we concern ourselves with the core programming features of the language. See chapter 5 for a discussion of further work.

A *class* in Smalltalk is defined by sending a `subclass:` message to an existing class. The result of evaluating the following expression:

```
object subclass: point instanceVariableNames: 'x y'
```

is a new class named `point` which defines two *instance variables* which are named `x` and `y` respectively. The new class is said to *inherit* from the class `object` which is named in the expression as its *super-class*. The class named `point` is a *sub-class* of the class named `object`; any class will have exactly one super-class but may have many sub-classes. An instance of the class `point` is created by sending it a `new` message:

```
point new
```

When a class is defined there are no values associated with the instance variable names. Each time a class is instantiated, fresh program locations are associated with each of the instance variable names. The locations are different for each instance of a class and are used as the private storage of the instance which may inspect the values of the locations and update them as it wishes. The locations are not available outside the instance.

A class may also define *methods* which are named operations. In the following expression, `point` is redefined in order to add a method `update:` which is used to change the values of the instance variables in a particular instance:

```
object subclass: point
    instanceVariableNames: 'x y'
    update: newx and: newy
        x ← newx.
        y ← newy
```

Methods are invoked using a keyword syntax, so the new method is named `update:_and:_` where the _ represent holes for the arguments which are named `newx` and `newy` in the definition. The method may be viewed as having the name `update:and:` and being supplied with a pair of values. The body of a method is an expression describing what happens when the method is invoked and what the return value of the method is. An expression e may be one of the following: a sequence of expressions $e.e$; a message $e\ i_1 : e_1 \ldots i_n : e_n$; a return value $\uparrow e$; a block $[i|e]$; a literal object (*e.g.* a number or a character); an identifier i; or an update $i \leftarrow e$.

When describing a programming language, it is important to explain when a given identifier may be referenced, when it becomes associated with a value and how long the association lasts. Class names are introduced when the class is defined. The name is immediately associated with the class and may be referenced anywhere in the program from that point on, unless it is overridden by another class definition with the same name or is temporarily hidden by an association for the same identifier which has a shorter lifetime.

Instance variable names are introduced in class definitions. They may be referenced within the body of any of the methods which the class defines. If a class `c` is used as a super-class in the definition of a class `c'` then all of the instance variable identifiers defined by `c` may also be referenced in the bodies of the methods defined by `c'`. For example the definition of `3dpoint` uses `point` as a super-class and defines a method `reset` which updates all of the instance variables to `0`:

```
point subclass: 3dpoint
    instanceVariableNames: 'z'
    reset
        x ← 0.
        y ← 0.
        z ← 0
```

Notice that the identifiers `x` and `y` are referenced within the body of the method named `reset` even though they are declared in the body of the class definition for `point`.

Instance variable names are associated with fresh program locations (initialised to some null value) when a `new` message is sent to the class in which they are declared or a sub-class of that in which they are declared. The result of instantiating a class by sending a `new` message is an object which is referred to as an instance of the class. The instance is a program value which may be passed to methods as arguments, returned from methods as results and stored in program locations. The associations between the instance variable names and the program locations which occur when a class is instantiated, prevail as long as the instance which is produced is accessible. For example, if the expression `3dpoint new` is performed then the result is an instance, o, of `3dpoint`. The object o is associated with (and is often thought of as *containing*) three new associations between the instance variable names `x`, `y` and `z` and three fresh program locations. The associations between `x`, `y` and `z` and their respective program locations will last as long as the object o can be referenced since they will be required when o is sent a `reset` message for example.

Although the instance variable names are associated with locations, references to the identifiers provide access to the locations only in limited circumstances. Usually when an instance variable name is legally referenced (*i.e.* within an appropriate method body) the reference produces the contents of the location which is associated with the name. The only exception is in an update expression $i \leftarrow e$ where the identifier i refers to the location whose contents are updated, by side effect, to be the value of the expression e. This rule applies to all other identifier types which are associated with locations, such as formal parameters and locals described below.

Method names are introduced by class definitions and are immediately associated with methods. When a class is instantiated, the resulting object is associated with (and is often thought of as *containing*) the method names and their values which are defined by the class and its super-classes. The associations between method names and methods prevail as long as the class in which they are defined, a sub-class of that in which they are defined or an instance of either is accessible.

Method names are referenced with respect to the object which contains them. The reference is performed using a send expression. If `o` is an object, $i_1 : \_\, i_2 : \_\ \ldots i_n : \_$ is a method and $o_1, o_2, \ldots, o_n$ are objects then:

$$o\ i_1 : o_1\ i_2 : o_2\ \ \ldots i_n : o_n$$

passes a message to o by referencing the method named $i_1 : \_\, i_2 : \_\ \ldots i_n : \_$ which is contained in o, associating each value $o_1, o_2, \ldots, o_n$ with the formal parameters of the method and then performing the method body.

The formal parameters of methods are identifiers which are introduced as part of the method definition. They are associated with values when the method is activated as part of passing a message. The identifiers may be referenced only in the text of the body of the method in which they are introduced. The association between the formal parameter and its value prevails for the duration of the method activation, *i.e.* when the method returns a value, the associations cease to exist. The values which formal parameters are associated with are fresh program locations containing the actual parameters which were passed in the message. The rules describing instance variable location reference and update which are described above also apply to formal parameters.

Methods may have local identifiers which are declared just inside the method body. In the following redefinition of `point`, the method named `swap` has a local called `temp`:

```
object subclass: point
    instanceVariableNames: 'x y'
    swap
      | temp |
      temp ← x.
      x ← y.
      y ← x
```

Local identifiers behave very like formal parameters except that they are associated with fresh locations containing null values instead of fresh locations containing the actual parameter values.

In addition to instance variable names, a class may define class variable names. For example the following redefinition of the class `point` introduces a new class variable named `swaps`:

```
object subclass: point
    instanceVariableNames: 'x y'
    classVariableNames: 'swaps'
    swap
      | temp |
      temp ← x.
      x ← y.
      y ← temp.
      swaps ← swaps + 1
```

The purpose of `swaps` is to keep track of the number of executions of the method named `swap`, no matter which instance of `point` or instance of any sub-class of `point` performs the swap.

A class variable name is associated with a value when it is introduced and the association prevails so long as the class which defines it, any of its sub-classes or any of their instances are accessible. The value associated with a class variable name is a fresh program location containing a null value. Class variable names may be referenced within the bodies of any method which is defined by the same class or any of its sub-classes.

In addition to instance variables and class variables, a class may define that certain pool variables are scoped over its methods and the methods of its sub-classes. A pool is a collection of associations between pool variable names and program locations. The pool is created once and may be shared between any collection of classes, unlike class variables which are created once and are shared between only those classes which inherit from the defining class. A pool variable name may be referenced in the body of a method which is defined in any class which declares the use of the pool or inherits from a class which uses the pool. For example consider a pool which records the number of times variables are swapped anywhere in the program. The number of swaps is held in a variable named `swapCount` in the pool named `stats`. The following two class definitions use the pool named `stats` to record when they swap co-ordinates and bank accounts. When the identifier `swapCount` is referenced in either of the methods named `swap` and `swapAccounts`, it will be the same program location which is updated in both cases:

```
object subclass: point             object subclass: bankAccount
   instanceVariableNames: 'x y'       instancevariableNames: 'depAcc curAcc'
   pools: 'stats'                     pools: 'stats'
   swap                               swapAccounts
      | temp |                           | temp |
      temp ← x.                          temp ← depAcc.
      x ← y.                             depAcc ← curtAcc.
      y ← x.                             curAcc ← temp.
      swapCount ← swapCount + 1          swapCount ← swapCount + 1
```

A pool variable name is associated with a program location once. The association persists as long as there is a possibility that the pool variable will be referenced, *i.e.* so long as a method body over which the pool variable name is scoped can be invoked.

The difference between pool, class and instance variable names arises due to the slight differences in the relations between the names and the program fragments within which they may be legally referenced. A difference also arises as to the lifetime of the associations: the lifetime of instance variable associations depends upon the lifetime of a single object, the lifetime of class variable associations depends upon the lifetime of a given class, its sub-classes and their instances and the lifetime of pool variable associations depends upon the lifetime of collections of classes, their sub-classes and instances.

A method body is performed in response to sending a message to an object o. The object o is associated with a collection of method names, one of which corresponds to the message name, the formal parameters and locals are associated with suitably initialised program locations and the method body is performed. When an identifier i is referenced as part of the method body, the state of a Smalltalk program will contain many packages of associations between names and values; some or all of the packages will contain associations for i. These packages are: the pools p_1 which contain associations which are shared between many parts of the program; the class variables p_2 which are defined by the class containing the method definition and all of its super-classes; the instance variables p_3 which are associated with the object o; the formal parameters p_4 whose associations were created when the method was invoked; and, the local variables p_5 whose associations were created when the method was invoked. A reference to identifier i is resolved by searching for an associated value in the packages in the order: p_5, p_4, p_3, p_2, p_1.

In addition to the classes of identifier which are described above, Smalltalk makes the identifier `self` available for reference in each method body. The value of the identifier `self` is the object to which the message was sent which invoked the currently executing method body. The scope of the association between the identifier `self` and the object lasts for the duration of the execution of the method body. For example, if the `point` class is redefined as follows:

```
object subclass: point
     instanceVariableNames: 'x y'
     update: newx and: newy
          x ← newx.
          y ← newy
     reset
        self update: 0 and: 0
```

when the method named `reset` is invoked for a particular instance of `point`, the body of `reset` will send the instance another message which will reset both its instance variables to

0.

The earliest point at which the association between the identifier `self` and an object o can be established is when o is created. In this way when `3dpoint` inherits from `point`, a reference to `self` in a method defined by `point` will refer to instances of `3dpoint` when the method is invoked with respect to an instance of `3dpoint` and will refer to an instance of `point` when the method is invoked with respect to an instance of `point`.

Since the association between `self` and an object o cannot be established until the object comes into existence, the association could be performed once for all the possible method invocations relating to o and then all the methods will share the same association. Alternatively, the last moment at which such an association may be established is when the method body is entered. This would imply that the association is performed once for every method invocation of o. These are two extremes of a whole spectrum of approaches to achieving self reference. Differences between the two approaches will become evident if the identifier `self` is bound to a program location. If the location is updated and is shared between all method invocations for o then it will be possible to change the value which is used for self reference in all subsequent method invocations. If the location is updated and the association is established with a new location for each invocation then the update will only have an effect for the duration of the current method invocation. Smalltalk associates the identifier `self` with an object just before each method invocation since each method is defined once in a class and then shared between all the instances of the class.

Smalltalk makes the identifier `super` available for reference in each method body. The value of `super` allows methods which are defined by a super-class to be invoked by a sub-class. This is useful when the sub-class defines a method with the same name as a definition in the super-class. For example, consider the following redefinition of the class `point`:

```
object subclass:  point
    instanceVariableNames:  'x y'
    update:  newx and:  newy
        x ← newx.
        y ← newy
```

and the following redefinition of the class `3dpoint`:

```
point subclass:  3dpoint
    instanceVariableNames:  'z'
    update:  newx and:  newy and:  newz
      z ← newz.
      super update:  newx and:  newy
```

When an instance of `3dpoint` is sent an `update` message, the definition for the method in `3dpoint` will shadow the definition in `point`. When the method body is performed, the `z` instance variable is updated and then an `update` message is sent to `super`. At this point, the value of `super` is an object which contains all the methods of `point` but none of those from `3dpoint`. In this respect, `super` is like `self` except that it refers to different parts of an object rather than the whole thing and its value depends upon where it is referenced.

The earliest point at which the identifier `super` can be associated with an object, is when the object is created. At this point, multiple associations between `super` and parts of the object can be constructed and associated with the correct methods. Using this strategy a number of methods originating from the same class definition can share the same association.

Whether they share or not can be detected by updating the location which is associated with `super`.

Alternatively, the latest point at which `super` can be associated with a value is when a method body is invoked in response to a message. In order to perform the correct association there must be a mechanism for determining which methods should be associated with `super` given a particular method invocation.

The duration of the association between the identifier `super` and its value will depend upon the strategy which is used to establish the association. If `super` is associated at the earliest opportunity then the association prevails as long as the object does. On the other hand, if `super` is associated at the latest opportunity then the association prevails for the duration of the method body in which it may be referenced. Smalltalk associates the identifier `super` with an object at the latest opportunity for the same reason as the association of `self` given above.

The execution of a Smalltalk method body may terminate in one of three ways: the body exits naturally by evaluating all the component expressions and returning the value of the last expression which is performed; or, the body exits by performing a return expression $\uparrow e$ which performs e and then returns its value as the result of the execution occurrence of the textually enclosing method; or, the body exits because a non-local return has been performed as a result of sending a message to a Smalltalk *block*.

In order to describe Smalltalk blocks we firstly describe a simple model of control flow for method execution. A method m is invoked, within the body of the currently executing method m', by sending a message to an object which contains m. The execution of the method m' is temporarily halted, producing a control point p' to which the result of the execution of m will return. If the body of m becomes exhausted then the value of the last expression evaluated is returned to p' where execution continues. If a return expression $\uparrow e$ is performed as part of the text of m then the expression e is performed and (providing no return or non-local return has occurred during e) its value is returned to p'.

As part of the text of m, a block expression $[i|e]$ may be performed, producing a Smalltalk block. The block is associated with the current point of control to which the block invocation will return, *i.e.* p'. A block is a program value which may be passed as an argument, returned as a result and stored in a location. At any time while the control point p' is active (*i.e.* during the current execution occurrence of m), the block may be invoked by sending it a message containing a value for the block identifier i. When this occurs a new control point p is created to receive the outcome of performing e. The performance of e behaves like a method invocation except that the result of performing a return expression within the text of e will return to the captured control point p' instead of p. When this occurs all intermediate control points are discarded.

Once a value has been returned to a control point p or p has been discarded due to a non-local return, any blocks which are associated with p become obsolete and may not be used. This has implications for the implementation of control points since they are allocated and de-allocated in a stack-like manner. A block has a dynamic scope for the duration of the invocation of the method which creates it and need only survive as long as there is a possibility of reference during this duration.

As an example of the use of Smalltalk blocks for control flow, consider defining a division method for integers which is used in resizing the length and width of a rectangle. The class `integer` is defined as follows[1]:

```
object subclass: integer
    add: ...
    sub: ...
    mul: ...
    div: otherInt exception: handler
      (otherInt isZero) ifTrue: handler ifFalse: someBlock
```

An integer object has methods for adding, subtracting, multiplying and dividing itself with respect to another integer. We are interested in the division method since it can go wrong when an integer is requested to divide itself by zero. The `div:` method is supplied with a divisor which is another integer and a handler which is a block. If the divisor is zero then an exception is raised by invoking the handler otherwise the division is performed via the object `someBlock` (whose details we omit). The `rectangle` class is defined below:

```
object class: rectangle
    instanceVariableNames: 'length width'
    resize: factor
      width ← width div: factor :exception [ _ | ↑ fail ].
      length ← length div: factor :exception [ _ | ↑ fail ].
      ↑ success
```

The class defines two integer valued instance variables and a method named `resize:` which will divide the `length` and `width` of the rectangle by the given integer `factor`. If the resizing is successful then the method returns the global value `success` and if the resizing fails then the method returns `fail`.

The method `resize:` uses the `div:` method of the class `integer` to perform the resizing with respect to the `length` and `width`. Each call of `div:` supplies the divisor and an exception handler. The exception handlers which are supplied by `resize:` are both blocks which will return the value `fail`. If either of these blocks are invoked during the `div:` method activation then control will return the value `fail` to the caller of `resize:`, causing the intervening calculations to be discarded. If zero is ever supplied as `factor` then the division of `width` by zero will activate the handler and immediately return `fail` from `resize:` discarding the rest of the method body. Notice that the exception handling performed by the second call of `div:` above is superfluous since any error will be caught by the first call; however, the calling protocol of `div:` requires an exception handler to be supplied.

2.2 Eiffel

The Eiffel programming language emerged in the mid 1980's and is aimed at supporting Software Engineering principles using an object-oriented framework. The language is strongly typed and provides facilities for specifying pre-conditions, post-conditions, invariants and exception handlers for methods. It is beyond the scope of this thesis to analyse the methodological aspirations of Eiffel which explain the origins of many of the language features. We focus on the main evaluation mechanisms which are particular to the object-oriented nature of Eiffel. An Eiffel program is a collection of class definitions, for example:

[1]Note that `this font` is used throughout this chapter for program text and that *this font* will be used to indicate incomplete fragments and pseudocode within program text.

```
class Point
     feature
        x :  Integer;
        y :  Integer;
        update(newx :  Integer, newy :  Integer) is
           x := newx;
           y := newy;
end
```

which defines a class named `Point` with two attribute features named `x` and `y` and a procedure feature (or method) named `update`. The body of a method is a statement s which is one of the following: a sequence of statements $s; s$; a side effect $i := e$, a conditional `if` e `then` s `else` s or a method invocation $e.i(e_1, e_2, \ldots, e_n)$. An expression e is one of the following: a constant such as a number or character; an identifier i; a creation i.`Create` expression or a method invocation $e.i(e_1, e_2, \ldots, i_n)$.

The name of the class, *e.g.* `Point` above, may be referenced within any method in the program as part of a type expression. A type expression declares the type of an attribute, method argument, method result or method local variable; type expressions are also used in *inheritance clauses* as described below. When the name of a class is introduced, it is immediately associated with the class and the association prevails for the duration of the program.

A class defines *features* which may be either attribute features or methods. An attribute feature is declared as a name and a type expression (*e.g.* `x` and `y` above) and a method is declared as a name, arguments, an optional return type and a method body. All the feature names may be referenced within the bodies of methods from the same class. An attribute name is not associated with a value until the class is instantiated. A method name is immediately associated with a method when it is introduced.

An object is created by instantiating a class. If an identifier i is declared as having type T which is the name of a class then the statement i.`Create` will instantiate the class named T and associate the resulting object with the identifier i. When a class is instantiated, each of its attribute names is associated with a fresh program location containing a null value. The null value is associated with the declared type T of the attribute name so that when the expression i.`Create` is performed, the null value can be replaced with an instance of T.

The result of instantiating a class is an object. The object is associated with the attribute names of the class and their program locations. The object is also associated with the methods of the class. The object may be thought of as *containing* the attribute names, their associated locations, the method names and their associated methods.

The names of the attributes and methods which are associated with an object may be referenced anywhere in the program providing that the reference is made with respect to the containing object, for example `o.update(0,0)` or `o.x`. The associations between attribute names and method names prevail as long as there is the possibility of reference, in the case of attribute names this is as long as the containing object may be referenced and in the case of the method names this is as long as there is the possibility of execution (via an object or a class instantiation).

An attribute name contained in an object is associated with a program location. If the attribute name is referenced as the left hand operand in a side effect statement, $i := e$ or $o.i := e$, then the contents of the program location is updated. Otherwise, if the attribute name is referenced as part of an expression then the contents of the program location is returned.

A method name contained in an object is associated with a method. All references to methods involve method execution, *i.e.* methods of objects are not first class data values. A method execution involves supplying values for the formal parameters of the method and then executing the method body to produce a value. Each formal parameter of a method may be referenced within the body of the method. The formal parameter is associated with a fresh program location containing the supplied value when the method is invoked and the association prevails for the duration of the method invocation. A method body may declare locals, each of which is associated with a fresh program location containing a null value when the method is invoked and the association prevails for the duration of the method invocation.

The identifier `Current` is scoped over all method bodies. Its value, during a method invocation, is the object which contains the method which is currently executing. For example, if a method is invoked by `o.f(x)` then the value of any reference to `Current` for the duration of the method invocation and as part of the text of the method body for `f` will be the object `o`. If, during the course of performing the text of `f`, another method of `o` is invoked without reference to `o`, *i.e.* `g(y)`, then the value of `Current` remains unchanged. If, during the course of performing `f` or `g` the method of another object is invoked then the value of `Current` will change for the duration of that method invocation.

A class definition includes an *inheritance clause.* The inheritance clause defines those classes whose feature definitions are to be included into the class being defined. For example the class `3dPoint` inherits from `Point`:

```
class 3dPoint
    inherit Point
    feature
      z : Integer;
      reset is
        x := 0;
        y := 0;
        z := 0;
end
```

By naming the class `Point` in the inheritance clause of `3dPoint`, the attribute definitions `x` and `y`, and the method definition `reset` are included. This has the effect of allowing the names of the super-class features to be legally referenced within the methods of any sub-class, as seen in the definition of `reset` above where the inherited feature names `x` and `y` are referenced without further qualification. Where no inheritance clause is present, the default super-class is `Object`, which for the purposes of this description contains various system features which are inherited by all classes in the program.

Certain program identifiers are declared along with a type, *e.g.* attribute names, method names, method parameters, local variables. The type is an expression in terms of the names of classes which are defined in the program. Usually a type expression is just the name of a class. The type of an identifier defines how it may be used within the program. In the case of an attribute, *e.g.* `x : T`, the type defines the collection of objects which may be placed in the program location associated with `x`. In the case of a method, *e.g.* `f(i:T):T'`, the types define the collection of objects which may be passed to the method as arguments and returned by the method as results. Formal parameters and locals of methods are the same as attributes, *i.e.* the types define what the locations associated with these identifiers may contain.

Consider declaring an attribute `i : Point`. The attribute `i` will be associated with different program locations each of which must contain either null or an instance of the class `Point`. Also, any instance of a sub-class of `Point` may be thought of as being an instance of `Point` by forgetting the extra feature definitions. For example, an instance of the class `3dPoint` can be thought of as an instance of the class named `Point` by forgetting the `z` and `reset` features. Similarly, any instance of classes which inherit from `3dPoint` can be thought of as instances of `Point` by forgetting the extra features.

Declaring an attribute name (or formal parameter, or method return value, or local) to be of a particular class type, defines the feature names which may be used with respect to values associated with the name. A class is associated with a *signature* which is the interface of names and their respective (signature) types which it defines, including all the names which it inherits. Given a declaration `i : T`, the names of the values associated with `i` must conform to the signature of `T`. For example, given a declaration `i : Point`, event though `i` has a value which is an instance of `3dPoint`, only those names in the signature of `Point` may be used, *e.g.* `i.x` and `i.update(0,3)`.

Each class definition may reference any other class by name either by inheriting from it or by using it as a type in an identifier declaration. There may be no cycles of inheriting classes, but classes which are used as types may be self referential either directly or indirectly. Thus, an Eiffel program is a mutually recursive collection of class definitions with a restriction that the inheritance relationship forms a partial order.

The inheritance clause of a class definition may refer to more than one super-class. This is referred to as *multiple inheritance* and causes the feature definitions from all the named classes to be included into the sub-class definition. In the case of single inheritance, there is no issue of sharing with respect to the inherited attribute features because all attributes in a class, whether they are defined locally or inherited, must have disjoint names in Eiffel. The issue of sharing arises with respect to multiple inheritance because the same class may be included more than once into a class definition. Consider the following diagram:

$$\begin{tikzcd} & A & \\ B \arrow[ur] & & C \arrow[ul] \\ & D \arrow[ul] \arrow[ur] & \end{tikzcd}$$

Class A is defined with a feature named f. Classes B and C both inherit from A and so include the definition of the feature f. Class D inherits from both B and C and therefore inherits from A *twice*. In general, a class may inherit from another class any number of times. A question arises as to whether the multiple occurrences of f at D are the same, *i.e.* there is really only one occurrence of the definition of f which is shared by all classes which inherit it; or whether the multiple occurrences of f at D are different, *i.e.* each time a class inherits a feature from another class, the definition of the feature is copied.

The difference in the two alternative possibilities arises when f is an attribute feature which is referenced by different methods which are defined by B and C respectively. If each method may access and update the attribute f independently, then the difference between a shared f or a copied f will occur when one method updates f and the other accesses it.

Eiffel defines that all occurrences of a repeatedly inherited feature will share unless the feature has been renamed. A feature may be explicitly renamed when it is inherited, for example in the following redefinition of `3dPoint`:

```
class 3dPoint
     inherit Point rename x as mycopyofx
     feature
        z :  Integer;
        reset is
           mycopyofx := 0;
           y := 0;
           z := 0;
end
```

Any class which inherits from both `Point` and `3dPoint` will define two features named `x` and `mycopyofx` respectively. These feature definitions will not share so that when the class is instantiated the two identifiers will be associated with different program locations. On the other hand, any direct instance of `3dPoint` will contain two identifiers named `x` and `mycopyofx` which will both be associated with the same program location. This can be seen when a direct instance of `3dPoint` uses a method defined by `Point` to update the value of `x` and the same instance uses a method defined by `3dPoint` to access the value (or vice versa).

A direct instance of `3dPoint` may be associated with an identifier `i` whose type is `3dPoint` or `Point`. The outcome of the expression `i.x` and the expression `i.mycopyofx` will be the same when the type of `i` is `Point` and `3dPoint` respectively. The expressions are illegal when the types are attributed the other way round.

In the absence of further information, all feature names may only be referenced in the methods of the class in which they are defined and its sub-classes. Eiffel provides a mechanism which allows the scope of such feature name references to be controlled. Each class definition includes an *export clause* containing a statement about the scope of each feature name reference `o.f`. For example, the following redefinition of the `Point` class:

```
class Point
     export update{Object};
     feature
        x :  Integer;
        y :  Integer;
        update(newx :  Integer, newy :  Integer) is
           x := newx;
           y := newy;
end
```

contains an export clause which declares that the feature identifier `update` may be referenced with respect to an object, *e.g.* `o.update(x,y)`, when `o` is an instance of `Point` or any sub-class of `Point` and providing the reference takes place in the method of any instance of `Object` or a sub-class of `Object`. Since all classes are sub-classes of `Object`, this allows references to `update` with respect to an instance of `Point` anywhere in the program. Each feature name may be exported to as many different classes as required, for example

```
export update { Graphics, Geometry, ...}
```

The export clause in the definition of `Point` does not export the features named `x` and `y`. This means that although these features may be referenced without qualification, *i.e.* as `x` and `y`, within the methods of `Point` and its sub-classes, they may not be referenced elsewhere. Consider the following redefinition of the class `Point`:

```
class Point
    export update L;
    feature
      x : Integer;
      y : Integer;
      update(newx : Integer, newy : Integer) is
        x := newx;
        y := newy;
      updateOther(o : Point) is o.update(x,y)
end
```

where `L` is a list of classes to which `update` is exported. For the statement `o.update(x,y)` to be legal, the class `Point`, or a super-class, must be present in the list `L` of classes to which `update` is exported.

A *generic class* [51] is parameterised with respect to a number of classes which are used in type expressions within the class body. For example the following generic class defines a range of objects which can contain values of any type:

```
class Cell[Contents]
    v : Contents;
    update(newv : Contents) is v := newv;
```

The identifier `Contents` may be referenced in a type expression anywhere within the class definition body. It is used in the generic `Cell` class as the type of the attribute named `v` and as the type of the formal parameter of the method named `update`.

Generic classes cannot be instantiated without first supplying values for all of the generic type parameters. Values are supplied by applying the generic class to actual type parameters within a type expression. For example, if the identifier `i` is declared

```
i : Cell[Point]
```

then the expression `i.Create` will instantiate the following class:

```
class PointCell
    v : Point;
    update(newv : Point) is v := newv;
```

The type application `Cell[Point]` has the effect of performing the class definition `Cell` in a context where the identifier `Contents` is associated with the class named `Point`. This association prevails for the duration of the class definition. The result is a class which, unlike non-generic classes within the program, is not associated with a program identifier.

Eiffel provides an exception handling mechanism. A control point is established on entry to each method and an exception may be raised during the execution of the method body. When an exception is raised, control jumps to the most recently established control point. The control point is discarded and if the method associated with the control point defines a *rescue clause* which is a program phrase then this is executed. The performance of a rescue clause may have one of two outcomes: the clause body performs a `retry` statement whereupon the method containing the rescue clause is re-invoked; or the clause becomes exhausted and an exception is raised. Raising an exception from a rescue clause does not cause the same rescue clause to be re-invoked because the associated control point has been discarded. Note that re-invoking a method via a `retry` will re-establish a control point just inside the method. The exception handling mechanism is demonstrated by the following example:

```
class Integer
    add ...
    sub ...
    mul ...
    div(divisor :  Integer):Integer is
       if divisor.isZero
       then raise
       else doDivision
end
```

The class `Integer` supports integer addition, subtraction, multiplication and division. If the divisor supplied to the `div` method is ever zero then an exception is raised. The exception will be handled by the most recently activated method with a `rescue` clause. Consider the following definition of `Rectangle` which uses `Integer`:

```
class Rectangle
    feature
       length, width :  Integer;
       resize(factor :  Integer):Outcome is
          do
             length := length.div(factor);
             width := width.div(factor);
          rescue
             if nothing can be done
             then raise
             else factor := getNewFactor; retry
end
```

The method `resize` uses `div` from `Integer` to modify the `length` and `width` attributes. Since `resize` defines a `rescue` clause, if `div` ever raises an exception then it will be handled within the call of `rescue`. The `rescue` clause will check to see if anything can be done to correct the situation. If nothing can be done then an exception is raised and the next most recent `rescue` clause will be invoked. If it is decided that corrective action can be taken then the value of `factor` is updated and the `retry` statement is performed causing the `resize` method to be re-activated with a new value for `factor`.

2.3 C++

The language C++ emerged in the mid 1980's as an object-oriented extension to the programming language C. Since C++ contains C as a sub-language and C is not object-oriented, C++ contains many features which are not found in most object-oriented languages, such as explicit type casting and access to raw memory. It is beyond the scope of this thesis to deal with the complete C++ language, since this would entail an analysis of non-object-oriented features inherited from C. We focus on the main features of C++ which determine its object-orientation.

A program in C++ is a collection of class, operation and data definitions. A program evaluates by performing the definitions in order and evaluating the predefined operation named `main`. A class definition has the following form:

```
class Point {
    int x;
    int y;
    update(int newx, int newy) {
      x = newx;
      y = newy;
    }
}
```

A class defines named *members* which are either attributes, such as `x` and `y` or methods, such as `update`. All classes are named, such as `Point` above. The scope of the name of a class depends upon where the definition occurs. If the definition occurs at the top level of the program then the name may be referenced anywhere from that point on in the program. C++ allows class definitions to be nested within a program context such as a function definition or another class definition. In this case the scope of the class name is defined by the extent of the enclosing textual context, limiting references to the class to the body of the function definition, for example.

The body of a method or function definition is a statement s. The following are a simplified list of C++ statement categories: an assignment $e = e$; a sequence $s; s$, a declaration $i : t$; a function call $e.i(e, \ldots, e)$; or a block $\{s\}$. An expression e is one of the following: an identifier i; a constant k; or a data member reference $e.i$. A type t is the name of a built-in type such as `int` or the name of a class.

Each attribute name defined by a class is immediately associated with a type signature, such as `int` for both `x` and `y` above. Each method name defined by a class is immediately associated with a method which defines formal parameters and a body. All of the names defined by a class are scoped over the method bodies, for example `x` and `y` are referenced within the body of `update`.

A class is instantiated to produce an object which is referred to as an instance of the class. As part of the instantiation process, each attribute name is associated with a fresh program location which contains an instance of the class declared for the attribute[2]. The resulting object is associated with the attribute names, the program locations, the method names and

[2]Notice that we are ignoring the difference between a pointer to a value and the value itself. This is a very complex area since C and hence C++ allow access to raw memory. Chapter 3 provides memory locations and allows copying operators to be defined so that both assignment by reference and assignment by value can be represented.

the methods. The object can be thought of as containing these values. The associations between attribute names and program locations prevail as long as there is the possibility of reference with respect to the object which contains them. The associations between the method names and the methods prevail as long as there is the possibility of reference via a containing object or an instantiation of a defining class.

The member names which are introduced in a class definition may be declared `public`, `private` or `protected`. The declarations control the scope of names when they are referenced with respect to an instance of the class, *i.e.* `o.i`. Names which are declared `public` may be referenced anywhere in the program. Names which are declared `private` may not be referenced as `o.i` outside of the defining class. Names which are declared `protected` are like privately declared names except when inherited (see below). For example the following redefinition of the class `Point`:

```
class Point {
    private:
      int x;
      int y;
    public:
      update(int newx, int newy) {
        x = newx;
        y = newy;
      }
}
```

makes the attribute names unavailable outside of the class body, however the method `update` may be executed with respect to an instance of `Point` anywhere in the program.

The identifier `this` is scoped over all method bodies. When the method is invoked using an expression `o.f(x)`, the value of `this` is the object `o`. When a method body invokes another method contained by the same object, the value of `this` remains unchanged. The duration of the association between the identifier `this` and an object is the entire method activation.

Classes in C++ may be nested and there are complex rules describing which identifiers become scoped over a nested class body. For example, the parameters and local variables of operations (and methods) may not be referenced within a nested class body. The reason for this is that there is no support for creating *closures i.e.* an association between an operation and a collection of values for the identifiers which are freely referenced in the operation body.

A C++ method need not be textually nested within the class definition to which it belongs. The class definition can contain only the name and type signature of the method and then the method can be defined elsewhere in the program providing that it is in the scope of the name of the class to which it belongs.

Operations in C++ may be declared to be friends of one or more classes. Such a declaration extends the scope of the member names which are declared by the class as `private` and `protected` to include the body of the operation. A class grants friendship by declaring the operation to be a friend in the class body – an operation cannot demand friendship. For example, if the class named `Point` is redefined as follows:

```
class Point {
     private:
        int x;
        int y;
        friend sneaky_updater(Point *p);
     public:
        update(int newx, int newy) {
           x = newx;
           y = newy;
        }
}
```

then the operation `sneaky_updater` is granted access to the names `x` and `y`:

```
sneaky_updater(Point *p) {
            p->x = 10;
            p->y = 20;
               }
```

A C++ class definition may name one or more classes to inherit from, for example:

```
class 3dPoint:  private Point {
     private:
        int z;
     public:
        3dupdate(int newx, int newy, int newz) {
           update(newx,newy);
           z = newz;
        }
}
```

the class named `3dPoint` inherits from `Point` and includes some or all of the member definitions from `Point`. The particular member definitions which are included depends upon the keyword which is used to inherit the class. The keywords which are available are `private` (as shown in the example) and `public`. When a class is privately inherited the definitions of public and protected members are included as private members of the resulting class. When a class is publicly inherited the definitions of public and protected members become public and protected members of the resulting class respectively. When a base class is protected then both the public and the protected members are included as protected members of the resulting class.

A class may multiply inherit from many classes and when it does there is the possibility that a class may be included twice. When this occurs, C++ defines that each time a member is included into a new definition via inheritance, it is copied. There will be as many distinct occurrences of a member as the number of times its defining class is inherited. This is analogous to the discussion on repeated inheritance in §2.2.

When there are multiple occurrences of a member with the same name (or when different members arise with the same name) C++ allows identifier references to be qualified with the name of the defining class. For example, If classes `A` and `B` inherit from `Point` and class `C` inherits from both `A` and `B` then the name `x` will be associated with two different memory locations in instances of `C` and similarly for the name `y`. Methods of `C` may distinguish between the attributes of `A`, by referring to them as `A::x` and `A::x`, and the attributes of `B`, by referring to them as `B::x` and `B::y`.

A class may define that all repeatedly inherited members share by declaring that a class is inherited using the keyword `virtual`. All members which have been inherited from the same source via `virtual` are shared in the resulting class. Using this mechanism all occurrences of an attribute which are repeatedly inherited may be defined to give rise to the same memory location in instances of a class. If class `C` from the previous paragraph inherits classes `A` and `B` using the keyword `virtual` then the occurrences of the member named `x` which is repeatedly inherited from `Point` will share and consequently when `C` is instantiated only one memory location will be associated with all occurrences of the name `x` (and similarly for `y`) in the resulting object.

C++ provides an exception handling mechanism which is analogous to that of ML [34]. Handlers are established by executing a `try` statement and prevail for the dynamic scope of the body. Each handler is a program statement associated with an identifier/type pair. An exception is raised by evaluating a `throw` statement whose argument is an expression which produces the value to throw. The exception is handled by finding the most recently established handler which is still in scope and whose type declaration matches the value which is thrown. Given such a handler, the identifier is bound to the thrown value and the body of the handler is executed.

For example, the class `Integer` provides a method for integer division which checks for division by zero and throws the dividend to the most recently established handler:

```
class Integer {
    add
    sub
    mul
    Integer div(Integer divisor) {
      if divisor.isZero
      then throw(this)
      else doDivision
    }
}
```

The class `Rectangle` uses the class `Integer` to implement to `length` and `width` attributes:

```
class Rectangle {
    Integer length, width;
    Outcome resize(Integer factor) {
      try {
        length = length.div(factor);
        width = width.div(factor);
      }
      catch(Integer i) { IntegerRelatedHandler }
      catch(Object o) { otherHandlers }
    }
}
```

The `resize` method places two exception handlers in scope before attempting to divide the two attributes by `factor`. If an exception is raised by throwing an instance of `Integer` then the first handler will be executed, otherwise the second handler will be executed – assuming that all values are instances of the ultimate super-class `Object`.

2.4 BETA

The language BETA was developed at the University of Linkoping as an extension to SIMULA-67. BETA represents an interesting variation from many object-oriented programming languages and has achieved a synthesis of features such as classes, procedures, methods and objects through its use of a single abstraction which is referred to as a *pattern*. In addition to providing object-oriented programming facilities, BETA provides concurrency primitives which are outside the scope of this thesis, see chapter 5 for a discussion of this point.

A BETA program is a collection of named pattern definitions followed by an expression. Each of the names are associated with the corresponding patterns and mutual recursion is allowed. A pattern has the following form:

$$\begin{array}{ll} (\# & d;d;\ldots;d \\ & enter\ s \\ & do\ e \\ & exit\ s\ \#) \end{array}$$

where `d` stands for an identifier declaration of the form $i : p$ where i is an identifier and p is a pattern or an identifier, `s` is an input/output specification and `e` is the body of the pattern which is a program expression. For the purposes of this work, an input/output specification is either an identifier or a tuple of input/output specifications. The body of a pattern is an expression e, which for the purposes of this thesis will be restricted to: sequences of expressions $e; e$; reference assignments $e[] \to e[]$; value assignments $e \to e$; identifier references i; remote identifier references $e.i$; new expressions `new` e; and constants such as numbers and characters.

When a pattern *definition* is executed the result is a pattern *value* which may then be instantiated and executed. Instantiating a pattern involves associating each of the names in the declaration part with instances of the corresponding patterns and executing a pattern involves associating values with the identifiers which are declared in the `enter` part, performing the pattern body and then producing the values of the identifiers in the `exit` part. In this way, when they are instantiated, patterns correspond to the conventional notion of a class producing an object. When they are executed, patterns correspond to the conventional notion of a method, function or procedure. In many ways, patterns correspond to closures in functional programming languages which allow part of the closure environment to be accessed from the outside.

The pattern `Point` defines a template for creating two dimensional point objects with a single method which allows the values of the dimensions to be updated:

```
Point :  (#  X, Y: @ integer;
             Update:   (#  Newx, Newy:  @ integer;
                           enter(Newx,Newy)
                           do  X + Newx -> X;
                               Y + Newy -> Y
                       #)
         #)
```

The identifiers which are declared in a pattern, such as `X` and `Newy` in the `Point` pattern, are scoped over all other patterns at the same level. A declaration may be one of the following: *inline* which is written `i: @p` and means that when the containing pattern is instantiated, `i` is associated with a new instance of pattern `p`; *reference* which is written `i: ↑p` and means that when the containing pattern is instantiated, `i` is associated with a fresh program location which may contain instances of the pattern `p`; or simply a pattern `i: p` which means that when the containing pattern is instantiated, `i` is associated with the pattern `p` (and can be used in `new` expressions).

When a pattern is instantiated, either explicitly using `new` or as part of the instantiation of a surrounding pattern, each of the names in the declaration part of the pattern which are either inline or reference declarations are associated with instances of their pattern types. For example when `new Point` is performed, `X` and `Y` are associated with instances of `integer`. The associations between the names and the instances prevails as long as their is the possibility of reference. Since the names are scoped over the bodies of all other patterns at the same level and these patterns may be instantiated and the results returned, the associations between the names of an object and the values may prevail long after the object itself has become inaccessible.

A pattern may be defined by combining a new pattern with a previously defined prefix pattern as in `3dPoint`:

```
3dPoint:  Point  (#  Z: integer;
                     Reset (# do 0 -> X; 0 -> Y; 0 -> Z #)
                 #)
```

The result of defining a pattern with a prefix is that the declarations of the prefix are appended to the declarations of the pattern. In the example above, the declarations of `X`, `Y` and `Update` are included into `3dPoint`. In this way the identifiers `X`, `Y`, `Z`, `Update` and `Reset` are all scoped over the body of the pattern `Reset`.

The identifier `this` is scoped over the bodies of all patterns. When a pattern is instantiated, the identifier `this` is bound to the resulting object for the scope of all the sub-patterns. The association between the identifier and an object persists as long as there is the possibility of reference. The issue of shadowing occurs with respect to the identifier `this` because patterns may be nested. This can lead to a problem, for example:

P: (# P': (# do ...this... #) ... do ...this... #)

which describes a pattern `P` which contains a sub-pattern `P'`. Given an instance of `P`, `new P[] -> X[]`, a reference to `this` in the executable part of `X` will refer to `X`. However, in an instance of `P'`, for example `new X.P'[] -> Y[]`, a reference to `this` will refer to `Y`. The identifier `this` always refers to the innermost object when objects are nested. In order to allow surrounding objects to be referenced from the innermost, BETA allows the identifier `this` to be qualified with the name of a pattern, for example `this(P)` or `this(P')`. When the identifier is qualified, the value is the object which surrounds the currently executing object and which was instantiated from the named pattern.

When a pattern is defined, it may specify a collection of declarations, an enter part, an executable part and an exit part. When a pattern is defined using a prefix pattern, the result will merge the declarations, enter, executable and exit parts as follows: the declarations are appended, the enter parts are appended, the resulting executable part is the prefix executable (with an important change – see below) and the exit parts are appended. Executable parts are merged so that the result is the prefix executable except that the identifier `inner` is bound to the executable part of the suffix pattern. Under normal circumstances, referencing the identifier `inner` within the executable part of an object will have no effect unless it was defined as the prefix of a pattern, in which case it causes the executable part of the suffix pattern to be performed.

This mechanism is the exact opposite of that described for `super` in Smalltalk (see §2.1). As such it allows operations to be defined which declare themselves to be extensible by performing the main activity and invoking `inner` whenever they allow themselves to be extended in different ways. For example, consider the redefinition of `Update` as defined in `Point`:

```
Update:  (#  Newx, Newy:  integer
             enter (Newx,Newy)
             do  Newx -> X;
                 Newy -> Y;
                 inner
         #)
```

the redefinition performs the same task as before but now calls `inner` at the end which allows any pattern which uses `Update` as a prefix to extend the basic behaviour:

```
3dUpdate:  Update (# Newz:  integer enter Newz do Newz -> Z #)
```

When prefix and suffix patterns are merged to produce a new pattern, the names which are declared by both patterns may overlap. When the occurrences of the same name both refer to patterns, BETA defines the result to contain a single occurrence of the name. The value associated with the name is a pattern constructed by merging the corresponding values in the prefix and suffix patterns. For example, the following definitions of `Point` and `3dPoint` patterns:

```
Point:  (#  X, Y: @ integer;
            Update:   (#  Newx, Newy:  @ integer;
                          enter(Newx,Newy)
                          do  X + Newx -> X;
                              Y + Newy -> Y;
                              inner
                      #)
        #)

3dPoint:Point  (#  Z: @integer;
                   Update:(# enter Newz do Newz -> Z #)
               #)
```

Given an instance `p:@3dPoint`, when the method `Update` is invoked, for example:

```
(10,20,30)->p.Update
```

the executable part of `Update` defined by pattern `Point` will be performed first. The attributes `X` and `Y` are updated and then `inner` is performed. During the execution of the body of `Update` defined by `Point`, the identifier `inner` is bound to the definition of `Update` defined by the suffix pattern of `3dPoint`. Referencing `inner` causes the value to be performed, updating the attribute `Z`.

Since BETA allows arbitrary nesting of patterns, merging the values of identifiers as prefix and suffix patterns can occur through many different levels of the same pattern definition. In the example given above, the nested pattern `Update` is used to implement a method and contains no patterns itself. For more complex applications, patterns may involve multiple levels of nesting. The ability to merge nested patterns is a programming feature which is unique to BETA, it supports some very sophisticated programming techniques, *e.g. virtual classes* [50], and its omission from all other object-oriented programming languages is directly linked to their limited support for block structured programming, *i.e.* in no other mainstream object-oriented language can classes (for example) be nested so that identifier scoping behaves as described by the Algol family of languages, see [49] for more details.

Nesting of patterns gives rise to a problem with the identifier `inner` which is the same as that described for the identifier `this`. The identifier `inner` always refers to the executable part of the innermost nested object except when it is explicitly qualified as in `inner(P)` in which case it refers to the executable part of the object which surrounds the current object and was instantiated from the named pattern `P`.

BETA provides a simple exception handling mechanism which can be extended using the object-oriented features of the language. The pattern named `exception` is provided by the system and has an executable part which causes a message to be displayed and execution of the program conditionally halted. An instance of `exception` contains associations for the names `stop`, `msg` and `continue` and these may be used in sub-patterns of `exception` in order to construct various exception handlers. Execution of an instance of the pattern `stop` will cause the program to halt. The name `msg` is associated with a text object and is used to display messages such as:

```
'This message is to be displayed' -> msg.putline
```

The name `continue` is associated with boolean values and may be updated by extensions to `exception`. After performing all of the body of an instance of `exception`, the program will be halted when the value of `continue` is `false`.

The following pattern describes integers which support a method `Div` which causes an exception to be raised when the divisor is zero:

```
Int: (# V:@integer;
        DivException:  Exception(# do   'Division by zero' -> msg.putline;
                                        inner
                                #)
        Add:...
        Sub:...
        Mul:...
        Div:  (# V':@integer;
                 enter V'
                 do  (if V' = 0 then
                          DivException else
                          V/V' -> V
                     if)
              #)
     #)
```

The exception handler `DivException` is a sub-pattern of `Exception` and extends the execution part so that it prints a message describing the nature of the exception. The handler is defined separately from the method `Div` so that sub-patterns of `Int` can extend its behaviour (via `inner`) of the exception to suit different circumstances. The following pattern describes rectangles:

```
Rectangle: (# Width:@Int(# DivException:(# do false -> continue#)#);
              Length:@Int (# DivException: (# do true -> continue
                                              exit Length
                                           #)
                          #);
              Resize:  (# Factor:  @integer;
                          enter Factor
                          do  Factor -> Width.Div;
                              Factor -> Length.Div;
                       #)
           #)
```

The attributes `Width` and `Length` are instances of a sub-pattern of `Int`. `Width` extends the exception handler of `Int` so that if `Width` is ever divided by zero, the program will halt. `Length` extends the exception handler of `Int` so that if `Length` is ever divided by zero, the program continues and returns `Length` unchanged.

2.5 Self

The language Self emerged in the late 1980's from Stanford University in the United States. This language is based on the idea of *prototypes* and does not provide a separate language feature corresponding to the *class* in other object-oriented languages such as those described in the previous sections. Objects are either defined as "one-off" data values or they come into existence by *cloning* or copying parts of other objects. An object contains updateable program locations and the language provides facilities for controlling the cloning process so that the result of cloning an object is a new object which shares certain program locations and has copied the contents of others into fresh program locations.

Computations are performed in Self by sending messages to objects. This is similar to message passing in other object-oriented programming languages since the message contains a name and optionally a collection of arguments, however the language implements methods as objects which are themselves cloned on activation.

Inheritance is implemented in Self by having a special type of attribute (referred to as a *parent*) which serves as a target for all messages which the object cannot otherwise handle. The attributes may be updateable program locations, supporting *dynamic inheritance* and an object may have more than one parent attribute, supporting multiple inheritance. This is often referred to as *delegation* [68] [44].

Unlike class-based object-oriented programming languages, Self provides the programmer with the facilities for building objects of arbitrary structural complexity. This leads to problems in determining the outcome of delivering a particular message to an object which is an arbitrary cyclic graph. Self proposes a number of novel solutions to this problem.

Self objects are collections of associations between identifiers and values. A value can be atomic, *e.g.* a number, a character or `nil`, another object or a method. Some methods, such as that which performs basic cloning and slot updating ($\leftarrow$) are built-in, other methods are just special types of object containing slots with predetermined names.

An object is created as a prototype by defining its slots, for example

```
(| x .  y |)
```

creates an object with four associations between `x` and `y` and the default slot value `nil` and the corresponding updating methods named `x:` and `y:` whose values are the slot updater method $\leftarrow$. When such an object definition is performed, fresh program locations are created for each of the slots, initialised to `nil` and associated with the slot names. Secondly, new identifiers are created for the updater methods by appending `:` to each name. These are each associated with the updater method and the corresponding program location.

The following object definition defines a prototypical point which has two attributes named `x` and `y` and provides a single method which is the built-in primitive which returns a copy of the receiver:

```
point = (| x .  y .  copy = _Clone |)
```

a new object can be created by cloning the prototype:

```
p1 = point copy
```

The object `p1` is created by taking each of the identifier associations in `point` and copying each of the program locations. An update to a program location in `p1`:

```
p1 x:  100
```

will not have affected the locations associated in `point`. The object `point` corresponds to the classes named `Point` in languages such as Smalltalk and Eiffel. The definition of `point` given above contains only one method named `copy` which is necessary to allow the object to be cloned. Other methods can be defined using the syntax `(| locals | body )` where `locals` are a list of slots which act as the formal parameters and local variables of the method, and `body` is a program expression.

A program expression e is either an identifier i, a sequence of program expressions $e.e$, a constant such as a number or a character, a message without parameters $e\ i$, or a message with parameters $e\ i:\ e\ i:\ e\ \ldots\ i:\ e$ where the concatenation of each of the identifiers corresponds to a method name of the value of the first expression and each subsequent expression is an actual parameter. For example, a method object which updates the `x` and `y` attributes of a `point` object is:

```
(| newx .  newy |
 x:  newx.
 y:  newy
)
```

If we want `point` to behave in the same way as the point classes defined in previous sections, then all points will have the same collection of methods. It is not possible, for example, in Eiffel to add methods to an object after it has been created. Furthermore, any changes to the collection of methods for point objects (*i.e.* an edit to the class) in languages such as Smalltalk and Eiffel will affect all the instances of that class.

Class-like behaviour is expressed in Self by creating a *traits* object containing all of the methods which are shared by a group of objects. The prototype object then inherits from the traits object by defining it as the value of a parent slot. Any messages which are sent to the prototype, or any object which is cloned from it, will be handled in the traits object via the parent slot:

```
pointTraits =  (| copy = _Clone.
                  updatex:y:  = (| newx .  newy |
                                 x:newx.
                                 y:newy
                                )
               |)

protoPoint = (| x .  y .  parent* = pointTraits |)

p1 = protoPoint copy

p2 = protoPoint copy

p1 updatex:10 y:20

p2 updatex:30 y:40
```

The objects `p1` and `p2` are copied from the prototypical point object. When they are sent the update message the method is found by searching the particular object for the named method. When this fails, the message is forwarded to the object in the parent slot (named `parent`) where the lookup continues. In both cases the value of the parent slot is the object `pointTraits`. This contains a method named `updatex:y:` which is activated and will update the `x` and `y` slots of the corresponding objects.

Self exhibits a high degree of coherence by implementing methods as objects. Methods correspond closely to Lisp *closures* as is described in [71], we give a brief account as follows. A method object is a collection of slots and values:

<parent>*	o
self*	nil
arg1	nil
arg2	nil
...	
argn	nil
<code>	*e*

When a method object is created, there will be a current collection of associations between identifiers and values. This corresponds to the current lexical environment in Lisp-like languages. The method object contains a slot named `parent` which will contain the lexical environment which was current when the object was created. This slot is a parent slot and is used to resolve references to free identifiers in the method body. The method object also contains a slot named `<code>` whose value is the method body. All other slots of the method object are initialised when the method is invoked in response to sending a message to an object containing the method as a slot value.

A message is sent to an object `o` which leads to the above method object being invoked. The invocation proceeds as follows. The method object is cloned and the `self` slot of the clone is set to the object `o`. This allows the body of the method to access the receiver even though the method may be a slot value of more than one object. `self` is a parent slot which means that any messages sent to the cloned method object which cannot be handled are forwarded to the receiver. Each of the message actual parameters are set as the values of the cloned method argument slots. Finally, the body of the method is performed with a current environment set to be the cloned method object, *i.e.* references to any identifiers during the performance of the body are resolved by sending a message to the method clone.

The way that `self` gets bound during Self method activation differs from class-based languages such as Smalltalk and Eiffel in the following way. In class-based languages, when an object is created it cannot be extended with further attributes and methods. This allows for the possibility that the value of the identifier `self` (or whatever is used for self reference) is bound *early* for each object. In Self, an object may be arbitrarily extended by becoming part of another object via a parent slot, so for each object `self` must be bound as *late* as possible when a method of the object is activated.

Self uses parent slots to implement inheritance:

```
3dPointTraits = (| parent* = pointTraits.
                   copy = (|| (parent copy) proto:  ((self proto) copy) ).
                   updatex:y:z:  = (| newx .  newy .  newz |
                                    z:  newz.
                                    parent updatex:  x y:  y
                                   )
                |)

proto3dPoint = (| proto** = protoPoint.
                  traits* = 3dPointTraits.
                  z
               |)
```

Like two-dimensional points above, three-dimensional points are represented as a traits object containing the shared behaviour and a prototype object defining the data slots. In both cases the objects contain parent slots which refer to the corresponding object from the two-dimensional point definition.

The method `copy` is defined in `3dPointTraits` to use the inherited definition of `copy` from `pointTraits`. This will copy the slots of `proto3dPoint` but not the inherited slots from `protoPoint` so the inherited slots are explicitly copied by `(self proto) copy`.

The object `proto3dPoint` gives an example of multiple inheritance in Self. The slots `proto` and `traits` are both parent slots. One of the problems which multiple inheritance raises is how to resolve the case where two or more of the parents of an object define a(different) slot with the same name. Self provides a facility which allows the programmer to choose an ordering. Each parent slot can be prioritized with respect to the other parent slots of the same object; when resolving a slot reference, the parents are visited in priority order. Parent priority is indicated by the number of asterisks after the slot name, the less asterisks the higher the priority. The object `proto3dPoint` has two parent slots where `traits` has higher priority than `proto`. This means that when `proto3dPoint` is sent a `copy` message, the method defined by `3dPointTraits` is activated rather than the method defined by `pointTraits`.

As described in [12] it is sometimes the case that the behaviour of an instance of a data type can be divided into several different modes of behaviour or implementation, with the state of an instance determining the particular mode. For example, boxed polygons are drawn using straight lines and smoothed polygons are drawn using splines. Self allows an object to update the contents of a parent slot. This allows the object to inherit different behaviour or different implementations of the same behaviour, at different times during its lifetime. For example, the implementation below:

```
      polygonTraits = (| copy = return a copy of the receiver |)

      boxedPolygonTraits =  (| parent* = polygonTraits.
                               draw = draw boxed polygon
                            |)

      smoothPolygonTraits =  (| parent* = polygonTraits.
                                draw = draw smooth polygon
                             |)

      protoPolygon = (| parent* boxedPolygonTraits.  vertices |)

      p = (protoPolygon copy) vertices:  some list of vertices

(1)   p draw

      p parent:  smoothPolygonTraits

(2)   p draw
```

describes two possible behaviours for polygons. The initial polygon prototype `protoPolygon` has the behaviour of boxed polygons. The object `p` is created by copying the prototype polygon object and sets the `vertices` slot to be some initial list of vertices. When the `draw` message is sent to `p` at `(1)`, the method defined by `boxedPolygonTraits` is activated. The parent slot of `p` is updated to contain the alternative implementation of `draw` which is defined by `smoothPolygonTraits`. When `draw` is sent to `p` at `(2)` the implementation will have changed and the drawing method is handled by `smoothPolygonTraits`.

The *dynamic inheritance* mechanism described above is unusual in class-based object-oriented languages. One reason for this is that many class-based languages are statically typed (such as Eiffel) and freedom to change the interface and implementation of an object makes the static analysis of the program very difficult. Another reason why class-based languages rarely support dynamic inheritance is that the mechanisms by which an object inherits a particular slot/attribute/method are usually hidden behind the language facilities which define classes and there are no facilities for constructing objects from first principles.

As described in [13], Self incorporates a unique feature which is used to control an irregularity which occurs as a direct result of providing the programmer with the ability to create objects of arbitrary structural complexity. An object is constructed of a collection of objects which are connected using the parent slot mechanism. When an object `o` contains object `o'` as the value of a parent slot, then `o'` is an ancestor of `o`, all of the ancestors of `o'` are also ancestors of `o`; correspondingly, `o` and all of its descendants are descendants of `o'`.

When a method sends a message to `self`, it expects to find the matching slot in one of the object's ancestors or descendants. This allows the sending method to view the invoked method as a more general abstraction (ancestor) or a refinement (descendant). However, if a descendant of the object containing the sending method contains unordered multiple inheritance and the multiple inheritance involves an otherwise unrelated object, then the message could become ambiguous. For example:

```
o1 = (| m = implementation of basic m.   n = (|| self m ) |)

o2 = (| m = (|| extension of m).   parent* = o1 |)

o3 = (| m = (|| completely unrelated m) |)

o = (| parent1* = o2.   parent2* = o3 |)
```

If the message `m` is sent to the object `o1` then the basic implementation is performed. The object `o2` extends `o1` and provides a new and better implementation of `m`. When `n` is sent to `o2` the new implementation of `m` is performed, even though the send was performed from `o1`. The object `o3` implements a completely unrelated collection of features from both `o1` and `o2` and calls one of its methods `m`. The object `o` needs to exhibit the features both of `o2` and of `o3` so it multiply inherits these objects.

The construction of the object `o` has an unfortunate effect on the behaviour of the method `n`. When `o` is sent a message `n`, the message `m` is sent to `self`, *i.e.* to `o`. Since the two parent slots of `o` are of equal priority, it is ambiguous as to whether the method named `m` from `o2` should be invoked or the method named `m` from `o3`.

From the construction of the object `o` it seems likely that the intention was to perform the method named `m` from `o2` rather than that at `o3`. This is because the object `o2` is initially defined as a unit and the modularity of the unit is severely altered by composing it with a completely unrelated object. Self defines the *sender path tiebreaker rule* to resolve the ambiguity arising in such situations: if two slots with the same name are defined in equal-priority parents of the receiver, but only one of the parents is an ancestor or descendant of the object containing the method that is sending the message, then that parent's slot takes precedence over the other parent's slot.

Using the sender path tiebreaker rule, the result of sending `n` to `o` is the invocation of `m` at `o2` since this is a descendant of the sending method from `o1`.

As seen from the example given above, methods can be overridden in objects when their names clash with a parent's method name, for example the method named `m` defined by `o2` overrides the definition of `m` from `o1`. Like Smalltalk, Self provides a mechanism for "continuing the lookup" from a method, which gives access to the method which is overridden. For example:

```
o1 = (| m = implementation of basic m |)

o2 = (| m = (|| some extra computation.  resend.m )
```

The object `o2` overrides the definition of `m` from `o1`. The body of `m` in `o2` performs some extra computation and then performs the basic implementation of `m` by performing a `resend` of `m`.

The resend mechanism will continue the message lookup from the object which contains the implementation of the currently executing method. If this object contains multiple parents, then all of these parents will be traversed in the lookup process. Under the circumstances that only one of the parents makes sense for a resend, Self provides the *directed resend* mechanism where the search continues with only one of the parent objects contained in the slot which is named as part of the `resend` statement.

2.6 Analysis

The previous sections in this chapter have given an overview of a collection of representative object-oriented programming languages. This section analyses these overviews in order to draw conclusions about general object-oriented features. Our claim is that the language which is defined in chapter 3 can support a wide variety of these features.

The aim of the analysis is to identify the object-oriented features which are broadly common to all the analysed languages. To a greater extent, the overviews have been performed with this task in mind and have omitted language features which are uncommon. However we believe that the features which are omitted, for example Smalltalk metaclasses and Eiffel pre- and post-conditions, are not typical of most languages which claim to be object-oriented and therefore can be legitimately excluded from the analysis. We claim to have been representative, within the scope of the thesis defined in chapter 1, in the choice of languages which have been described in the previous sections.

The analysis is performed by addressing a number of areas which are common to the languages. In each case the languages are compared and contrasted, a collection of features is identified which must be supported by any general framework, and a statement is made about how the language features differ from those provided by a simple λ-calculus. Such differences will lead to the design of language extensions for a λ-calculus which is defined in chapter 3. The common areas are: objects, classes, identifier scopes, sharing and updating, methods and message passing and exception handling.

The following starting point is used in order to suggest general object-oriented extensions to a λ-calculus. Expressions in the calculus are one of: identifiers, single argument functions and applications. The language has a call-by-value execution strategy [60]. Standard language extensions are assumed as required, for example tuples, arithmetic values and if-then-else expressions.

2.6.1 Objects

All of the analysed languages exhibit objects which are structured collections of associations between program identifiers and values. The single operation which can be applied to an object is that of message passing. Most languages seem to distinguish between two forms of message passing: that which produces the value associated with an attribute and that which activates a method.

Languages differ as to whether all the identifiers bound by an object are available from the outside via message passing. For example, Smalltalk provides only the method names, Self provides all of the names and C++ & Eiffel allow the programmer to declare private names which are available only within certain sections of program code.

Self is unique in providing basic object construction operations. One reason why languages such as Eiffel do not provide access to such operations is that the type systems are based around classes which provide a restricted collection of object construction operations via the instantiation mechanisms. To provide unrestricted access could make the type systems very difficult to enforce.

All object-oriented languages allow objects to refer to themselves via an identifier which is used for self-reference. This seems one of the key features of object-oriented programming languages. There are differences between languages as to when the self-reference identifier is bound or could be bound. For example in Self, the identifier must be bound at the point when an object's operation is activated since the language provides operations which extend objects. Alternatively, Eiffel allows the identifier for self-reference to be bound at the point where the object is created, since the object cannot change during its lifetime.

Self provides an interesting variation of self-reference, where the referent of self-reference differs depending upon where the reference takes place in the object. This facility allows an object to be viewed differently depending on where the the viewer is within the object.

With respect to object behaviour, a general object-oriented framework must support features for: the association of names and values; the ability to reference a value through a name and to activate an operation through a name; the ability to hide some of the associations as private to the object; the ability to create self-referential associations between a name and an object; the ability to extend an object once it has been created; the ability to seal an object during its lifetime so that it cannot be extended further; the ability to view an object differently depending upon where the viewer is.

In order to support objects, a λ-calculus must provide mechanisms for associating keys and values. The keys should not rely upon positional information within an object, this excludes the possibility of representing objects as tuples and keys as indices. An operation must be provided which produces the value associated with a key in a given object. Such an operation can be used to support attribute reference and message passing since operation invocation can be viewed as producing the method and then activating it.

At a simple level, sets of labelled values could be used to represent objects where the lookup operator simply selects the value with the appropriate label. Unfortunately, some language features rely upon objects having more structure than simple sets (for example shadowing rules and the sender path tiebreaker rule) which can only be used to represent objects after the structure had been "flattened out".

It is desirable to provide operators which allow objects to be constructed from first principles out of their atomic components. Such operators will allow objects to be merged together whilst remembering enough structure to uphold shadowing rules *etc.*

The identifier scoping rules of the λ-calculus are often used to limit the scope of identifiers so that they cannot be referenced from outside a program fragment. To a limited extent, these techniques can be used to allow objects to have private and public parts. These techniques do not explain the mechanisms of Eiffel, for example, where identifiers can be exported to particular program fragments by naming the classes to which the identifier is exported. Such facilities, in addition to inheritance scoping which is described below, would suggest that a λ-calculus is required to provide facilities which allow identifier scoping rules to be programmed to some extent.

Many λ-calculi allow recursive definition and cyclic data values to be constructed using the combinator $\mathbf{Y}$. This is a good candidate for providing closed self referential objects. Open self referential objects must be supplied with the value for self reference when a message is sent, in which case the object can be sent as an extra argument to the method.

2.6.2 Classes

Most object-oriented languages are class based and classes tend to have a number of identifiable roles:

- Packages of definitions which can be composed with other classes to construct larger packages of definitions.

- Abstractions over objects where an object is created by supplying all of the missing values. There appears to be almost universal rejection of partial instantiation which supplies only some of the values. Also, most languages do not allow abstractions over classes, which would imply classes as first class data values.

- Programming language types which occur in the signatures of functions and attributes. The relationships between classes which arises from their combination during definition produces a corresponding collection of relationships between the classes when they are used as types. This produces so-called sub-type or inclusion polymorphism whereby an object of type c may be viewed as being of type c' when both c and c' are classes and c' is a super-class of c.

- Classes are associated with scopes for the names which are defined in the class itself and any sub-class. For example, the attribute names defined by a class c may be referenced in the bodies of any sub-class of c.

The composition of classes during the production of a new class is usually referred to as *inheritance*. Languages differ with respect to the number of classes which may be composed to produce a new class. Single inheritance involves composing a single existing class with an extension – the extension can be thought of as another class. Multiple inheritance involves composing multiple existing classes with an extension.

Languages differ with respect to whether or not a class must be named. Smalltalk and Eiffel force classes to be named (although Smalltalk classes could in theory be unnamed) whereas BETA allows anonymous patterns. Eiffel is characteristic of languages which are forced to invent arbitrary rules in order to distinguish between the use of a class in an expression and the definition of a class. For example, class definitions cannot be used in type expressions except where the class is being constructed by applying a generic class to a type argument. Languages which do not insist on naming values in programs, greatly increase the coherence of the expression rules, for example the rules for definition of a pattern and usage of a pattern in BETA are the same.

Languages differ with respect to whether or not a class may be nested. Eiffel prevents class nesting whereas BETA and C++ allow class nesting. There are a number of problems associated with class nesting which arise from the identifier scoping rules which occur in object-oriented languages. Firstly, the context in which a nested class is created becomes important, since the same definition can be performed more than once. This problem can be solved by treating a class like a function closure in Lisp – when it is created, a class must be associated with the current lexical environment so that these can be passed on to its instances when they are created. Secondly, nested classes can cause problems with respect to self-reference. These problems can be solved using techniques similar to those for BETA in §2.4.

There are a number of class definition issues which involve possible ambiguities. For example, when a class repeatedly inherits from another class and when a class inherits a value for the same name twice. Eiffel deals with this issue by mandating one of the multiple possibilities for each ambiguous situation. Self allows the programmer a degree of control by ordering multiple inheritance. None of the languages surveyed provided the programmer with full control over situations where more than one possibility could arise.

Most strongly typed object-oriented languages implement sub-type polymorphisms. Program identifiers which are associated with program values are given a type and may be associated with any value of the given type or a related type. Types are related by inheritance so two objects may be associated with the same program identifier when they are both instances of the type declared for the identifier or some type which inherits from it. It is usually the case, that the type of an identifier defines the signature of names for its values. So, even though an object may be an instance of a sub-class, whilst it is bound to an identifier whose type is a super-class, only the names in the signature of the super-class may be referenced.

Languages which are strongly typed and which provide classes also often provide classes which are parameterised with respect to one or more type variables; these are often referred to as generic classes. Wherever classes are used as program values, such as in the declaration of identifier types or in instantiation expressions, a generic class may be applied to one or more classes to produce a new class.

With respect to class behaviour, a general object-oriented framework must provide: abstractions over objects, allowing initial values to be passed to the class; instantiation which turns the class into an object in response to a collection of initial values for the attributes; extension of a single class with more definitions to support single inheritance; combination of multiple classes with more definitions to support multiple inheritance; the issues of repeated inheritance and shadowing order should be defined by the programmer; operators which modify classes by renaming and removing definitions; the use of classes as types in identifier declarations and the subsequent restriction of object interfaces bound to the identifier; the parameterisation of classes to support generic classes; classes should be first class data values; nesting of classes should be allowed without any unreasonable restrictions; BETA style merging of nested classes with the same name should be optionally supported.

A basic λ-calculus can support many of the required features of classes simply using functions which are used in suitable ways. For example, if the calculus can support objects then abstraction over objects is provided by wrapping functions around them. Functions are first class and may be nested. Functions which are parameterised with respect to other (suitably behaved) functions can be said to be generic classes. This representation does not easily work for classes as types because the functions are not immediately associated with the signature of the class which is used to restrict objects. To overcome this problem, a class is represented as a pair containing the function and the signature as a collection of associations between names and other classes.

2.6.3 Identifier Scopes

Identifier scoping describes the fragments of program text over which an identifier may be legally referenced after it is introduced. Traditional languages such as Pascal [76] and the Algols enforced *textual scoping* where an identifier introduced via a declaration construct and is scoped over the rest of the textually enclosing block. Some other languages including Common Lisp allow *dynamic scoping* where an identifier may be introduced and references are legal to it at any time during the evaluation of the declaration construct.

A programming construct which declares a new identifier may be performed once or multiple times during the evaluation of a program. A question arises as to whether subsequent declarations of an identifier will produce the same identifier or a different one.

Declaration constructs can usually be nested and the same identifier may be declared in a context which already contains that identifier. The traditional mechanism for dealing with this case is to allow the inner declaration to *shadow* the outer declaration for the scope of the inner program.

Object-oriented languages do not conform to the traditional scoping rules of textual and dynamic. The identifiers which are declared in a class are scoped over the methods which are defined by the class. The methods may be defined in the class or elsewhere in the program as is the case in C++. The identifiers are also scoped over all the sub-classes of a class. These are program fragments which are not textually nested within a class definition and are not performed during the evaluation of a class definition.

This suggests a new type of identifier scoping: *inheritance scoping* which is governed by inheritance relationships between classes. There are a number of variations on this theme, for example in C++ the keywords **public** and **private** can be used to control inheritance scoping.

Shadowing in object-oriented languages is handled differently in each language. For example Eiffel does not provide shadowing as such, it provides mechanisms for redefining: *i.e.* deleting and defining. Smalltalk allows methods to be shadowed and the shadowing method can refer to the shadowed method via `super`. BETA allows methods to be shadowed and the shadowing works the other way around.

When a declaration of an attribute is performed in most object-oriented languages, a fresh program location is allocated. Declarations are performed when classes are instantiated and therefore different instances of the same class contain the same attribute identifiers but associated with different program locations. The exceptions to this rule are the static declarations of C++ and the class variables of Smalltalk which are evaluated once and produce the same identifier and value on each subsequent evaluation.

In common with programming languages which offer *modules*, such as Modula-2, some object-oriented languages allow identifier scoping to be controlled via *import* and *export* mechanisms. Eiffel is an example of this type of language. Although this mechanism does interact with inheritance scoping, it is viewed as a feature which is characteristic of module-oriented languages and is not within the scope of the thesis.

With respect to identifier scoping, a general framework must provide features for: inheritance scoping, fine grain control over the shadowing of identifiers including `super` and `inner`, the ability to control the lifetime of associations such as class variables.

The λ-calculus naturally supports textual scoping of identifiers due to the β-reduction rule. Operational semantics for the λ-calculus usually involve a component which is called an *environment* which contains associations between the identifiers which are currently in scope and their values. Textual scoping is implemented by associating the current environment with functions when they are created, forming *closures*; by replacing the current environment with a closure's environment when it is applied; and by saving and restoring the environment on function call and return. Other scoping rules can be supported by suitable modification to the evaluation strategy.

Such an operational framework is ideal for analyzing the identifier scoping rules of a programming language since the identifier bindings are made explicit and rules can be added with control how the identifier bindings are modified when program constructs are performed. The most general approach to investigating new scoping mechanisms is to provide a language with basic primitives for affecting the underlying environment and how it is used. In this way different scoping mechanisms may be constructed by combining the primitives in various ways.

2.6.4 Sharing and Updating

All object-oriented languages implement object attributes as program locations which may be updated by side-effect. Objects are often designed as state-transition machines where state transitions occur as a side-effect of sending an object a message. The state of an object is described by the current contents of the program locations which are named by the object's attributes.

Languages differ as to whether objects provide external access to their state or whether the state is encapsulated. Smalltalk enforces encapsulation whereas Eiffel allows the programmer to define whether an attribute is encapsulated or not.

Although all object-oriented languages provide objects with state, this feature is not significantly different to the state which is provided by non object-oriented imperative languages such as C and Pascal.

The semantics of basic λ-calculi do not involve state. There are two major approaches to introduce state into programs written in a λ-notation. The first is to explicitly represent state as a data value which is *threaded* through the program and is the approach used in denotational semantics. The second is to add an extra component to an operational semantics which is a state or *heap*. The first approach preserves referential transparency since the state is *reified* at the expense of having to pass the state as an argument to all functions and return it as a result from all functions. The second approach hides the state behind the scenes and provides primitive operations for accessing it only when required at the expense of losing referential transparency in programs. A relatively recent technique which involves the use of *monads* [72] combines the benefits of both approaches. This approach has not been used in this thesis but could prove useful in order to integrate object-oriented and lazy functional programming paradigms.

2.6.5 Methods and Message Passing

All object-oriented languages provide object operations or *methods*. A method is activated by *sending the object a message*. A method has a number of formal parameters and a body. When the method is activated the formal parameters are bound to the actual parameters and the body is performed.

Method activation is similar to procedure activation. The differences include provision for inheritance scoping and self reference within the method. Few object-oriented languages provide methods as first class data values and usually require that methods are defined as part of object/class definition.

Some object-oriented languages treat methods as objects. The local variables of the method become attributes of the method-object and the method-object handles a small collection of messages which relate to its activation.

A framework for object-oriented languages must provide procedures and functions which are used as methods. The methods should be first class data values and should allow access to their local variables.

Operational semantics for λ-calculi often implement functions as *closures* which are data values. Closures are a good candidate for implementing methods but will require primitive operators for accessing their internals if they are to be treated as objects.

2.6.6 Exceptions

Many languages provide features for non-local goto and handling exceptions. Such features are not characteristic of object-oriented languages but languages such as Smalltalk, Eiffel and C++ provide object-oriented variations on the theme of exceptions.

A framework which support object-oriented features must provide a flexible mechanism for constructing exception handlers and non-local goto. Operational semantics for λ-calculi do not usually provide such control mechanisms, but can easily be provided with such a facility using the J operator (see chapter 3).

2.7 Features

Figures 2.1, 2.2 and 2.3 show the features which have been identified as being characteristic of object-oriented programming languages. The features which are shown in figures 2.1 and 2.2 are viewed as being more fundamental to the object-oriented model of programming then the features shown in figure 2.3. Consequently these are the features which are analysed in some detail in chapter 4. We revisit the features in figure 2.3 in chapter 5 and give an overview of how they can be implemented using the primitives from chapter 3.

Chapter 3 defines a simple functional programming language called the Λ-calculus which contains primitive mechanisms for supporting object-oriented languages. Each feature will be analysed in chapter 4 by giving its specification and an implementation using the Λ-calculus.

The list of features is far from exhaustive, for example sophisticated type systems and multi-methods [55] are not addressed, but it is representative of the type of facilities which are offered by object-oriented languages. Given implementations for these features in the Λ-calculus the examples should be sufficient to form a basis for the definition of related features which are not covered in the thesis.

Feature number	Description
1	The framework must support simple objects as collections of labelled values.
2	The framework must allow the values which are contained in objects to be program locations.
3	The framework must allow objects to be incrementally extended with extra collections of labelled values.
4	Where values in objects are functions, they must be able to reference other values by name and without further qualification.
5	An object may have private names which refer to values internal to the object.
6	The framework must provide facilities to construct a wide variety of object self-reference mechanisms.
7	The objects which is used for self reference can depend upon the position in the object at which the self reference occurs.
8	Parts of an object may be associated with labels and self reference may use the labels to refer to various parts of 'self'.
9	An object can inherit attributes and methods from other objects and the inheritance can be changed during the execution of the program.
10	The names in an object can be updated during the execution of a program.
11	Classes can be defined in terms of attributes and methods. Instantiation of classes produce objects by supplying initial values for the attributes.

Figure 2.1: A List of Object-Oriented Features: Part I of III

Feature number	Description
12	Attributes can be defined when a class is created (class variables) and are shared between all of its instances.
13	Classes are first class values which may be passed as arguments, returned as results and stored in program locations.
14	Classes may be arbitrarily nested.
15	Classes inherit attribute and method definitions from a single existing class.
16	Identifiers which are referenced in method bodies may be bound *late* taking advantage of redefinitions in sub-classes.
17	Sub-class methods shadow super-class methods. The shadowing method can reference the super-class method via the identifier *super*.
18	Super-class methods shadow sub-class methods. The shadowing method can reference the shadowed method via the identifier *inner*.
19	Inheritance supports both run-super and run-inner. The programmer can choose which is the most appropriate.
20	Classes may inherit attributes and methods from more than one super-class.
21	Where a class is inherited more than once due to multiple inheritance, attributes are not copied.
22	Functions may be defined using inclusion polymorphism. The parameter of the function is typed and objects of the type and any of its sub-types may legally be supplied as actual parameters.
23	Facilities for exception handling and non-local gotos are supported and may be integrated with other object-oriented features.

Figure 2.2: A List of Object-Oriented Features: Part II of III

Feature number	Description
24	A wide variety of object-oriented program structure is supported.
25	Programmer control over sharing and copying with respect to repeated inheritance.
26	Names in classes and objects can be renamed and are automatically renamed due to inclusion polymorphism.
27	Classes may be parameterized with respect to all types of data value including other classes.
28	The basic features of the language are controlled by (meta-)class definitions.
29	Definitions in a class may belong to different categories which determine whether they are inherited.
30	Shadowed definitions may be referenced using explicit pathnames involving class names in inheritance paths.
31	Classes may be field values in objects and BETA-style class composition is supported.
32	Messages may be resent in methods to named antecedents and descendants.
33	Export and import mechanisms for class definitions are supported and integrated with other object-oriented features.

Figure 2.3: A List of Object-Oriented Features: Part III of III

Chapter 3

The Λ-Calculus

3.1 Programming Language Definition

The aim of this thesis is to present a programming language which can be used as a framework within which to construct object-oriented language features. The language must be defined precisely so that it clearly expresses the meaning of each object-oriented feature. There are two major components to the definition of a programming language: is *syntax* and its *semantics*.

The syntax describes how programs in the language are constructed and may be *concrete* in which case the syntax describes how the program will look on the screen or printout, or *abstract* in which case the syntax describes the data which will be used as input to a compiler or an interpreter for the language.

The semantics describes the meaning of a program. Different meanings can be attributed to the same program depending on the intended use of the meaning. For example, the meaning of a given program may be defined as the outcome of performing it or the sequence of steps which are taken when performing it. In [56] the following categories of programming language semantics are defined:

Operational Semantics The meaning of a programming language construct is specified by the computation it induces when it is executed on a machine. In particular it describes *how* the effect of computation is produced.

Denotational Semantics Meanings are modelled by mathematical objects that represent the effect of computing the constructs. Thus *only* the effect is of interest, not how it is obtained.

Axiomatic Semantics Specific properties of the effect of executing the constructs are expressed as *assertions*. Thus there may be aspects of the executions which are ignored.

Each category of semantics is interesting in its own right and could be applied to object-oriented programming languages. We have chosen operational semantics to represent the meaning of the framework language since this is general purpose and we are interested in object-oriented programming features and how they work. Within the category of operational semantics, [56] identifies two extremes:

Natural Semantics Sometimes referred to as *big-step* semantics, this aims to define the relationship between each program and its outcome.

Structural Operational Semantics Sometimes referred to as *small-step* semantics, this aims to describe computation as a sequence of small individual steps and in terms of the changes which occur at each step.

We have used a mixture of small-step and big-step semantics to define the meaning of the framework language and of object-oriented programming language features. It is useful to describe the small steps taken by a computing engine in order to give a feel for how the machine might be implemented, whereas it becomes tedious to spell out all the individual steps for complex features (for which there are many different implementation strategies).

Operational semantics gives a description (at some level of abstraction) of how the programming language executes. It is not usually sufficient to give an interpreter for the language in some existing base language, unless the base language has a precise semantics. For example, if the framework language was defined as an interpreter written in C, then questions could arise as to the actual meaning of framework language constructs since C does not have a precise semantics. Operational semantics are usually defined in terms of simple state transition systems which can be precisely defined without recourse to complex evaluation mechanisms. This is the approach which we have taken in the thesis.

The choice of a particular framework language is motivated by the requirements that it must be simple, general purpose and support the features which are found in many programming languages. We also require that it must allow the easy identification and implementation of programming features which are characteristic of object-oriented programming languages.

Landin [47] identified λ-notation as a general purpose tool for programming language analysis. It has a simple syntax and can readily be given an operational semantics in terms of a state transition system (the SECD machine [46]). In particular he identified a number of ways in which the basic λ-calculus can be extended with new features. The choice of features which are added in order to construct a family of related programming languages capture the characteristics of the family.

We use the λ-calculus as the basis of the framework language. The basic λ-calculus is extended with features which are characteristic of object-oriented programming languages. We have termed the extended language the Λ-calculus (for want of a better name). The Λ-calculus is defined in terms of its syntax and semantics (the Λ-machine). The features which are characteristic for object-oriented programming are identified as extensions to the basic language. The Λ-calculus is then used to construct characteristic object-oriented programming features. Where appropriate, the new features are packaged up into new calculation rules which are added to the basic Λ-calculus semantics.

3.2 Components of an Operational Semantics

The Λ-calculus is given an operational semantics by defining a state transition machine. In order to define the language, definitions for the following must be given: concrete syntax; abstract syntax; the states of the machine; and, a transition relation between states.

The concrete syntax of a programming language is usually defined using Backus-Naur Form (BNF) [65]. BNF is given in terms of a collection of mutually recursive rules for specifying how the programming language will appear on the screen or the printed page.

The abstract syntax of a programming language defines how the syntax can be represented as a data value without extra information required by the concrete syntax necessary to distinguish between multiple possible representations. Abstract syntax is viewed as being the result of a parse whereas concrete syntax is viewed as the input to the parse.

We will not use any notation to distinguish between concrete and abstract syntax. Thus the closure $<i, e, b>$ is ambiguous: is b a meta-syntactic variable whose value is a program expression or is the body of the closure the program identifier b? In practice no confusion should arise and it will be obvious from the context which case is intended.

The states of a machine are defined to be n-tuples. Each component of the n-tuple is a *machine component*. A sub-set of all possible machine states are *initial states* which contain entire programs and the context information necessary to evaluate the program. A sub-set of all possible machine states are *terminal states* which contain the outcomes of performing programs. States which are neither initial or terminal represent partially evaluated programs.

Each machine state must be defined in terms of the data values which occur in the particular n-tuple. These will include the following:

- Program constructs which are currently being performed or which are part of other data values in the state.
- Values which are the results of performing programs or sub-programs. Some values may be temporary in the sense that they are the results of performing sub-programs and will be consumed during subsequent computation. Other values may occur as part of the final outcome of the computation. Values which result from performing some program expression are referred to as *denotable* and may occur as part of the program outcome.
- Values which may not form part of the outcome of performing a program *i.e. non-denotable* values. These values are usually part of the execution mechanism of the machine, for example machine components.

Given a machine state, the transition relation defines the set of subsequent states. For deterministic programming languages there is at most one subsequent state and the transition relation is a function. A calculation is performed by constructing an initial state and continually applying the transition function until a terminal state is reached. The entire program is part of the initial state and the outcome of the program is part of the terminal state.

For the Λ-machine, the transition function is $\longmapsto$. It is defined using a collection of separate definitions which consist of *patterns*. A pattern p is the same as a data value except that variables may occur anywhere that a value may occur. A *substitution* θ is a collection of associations between variables and values; a substitution is applied to a pattern $\theta(p)$ to yield a value which is the result of replacing each variable in the pattern with the corresponding value in the substitution.

Each definition which contributes to $\longmapsto$ is called a *rule* and has one of the following forms: either $p_1 \longmapsto p_2\ R$ or:

$$\frac{\begin{array}{l} p_{11} \longmapsto p_{12} \\ p_{21} \longmapsto p_{22} \\ \vdots \\ p_{n1} \longmapsto p_{n2} \end{array}}{p \longmapsto p'}\ R$$

The first form is a rule such that for every state σ_1 such that there is a substitution $\theta(p_1) = \sigma_1$ then $\longmapsto (\sigma_1) = \theta(p_2)$. The second form is a rule which means that if there is a substitution

θ such that $\longmapsto (\theta(p_{i1})) = \theta(p_{i2})$ for $i = 1, \ldots, n$ then $\longmapsto (\theta(p)) = \theta(p')$. The labels R are used to refer to the rules.

3.3 The Basic Λ-Calculus

3.3.1 Syntax

The concrete syntax of the basic Λ-calculus is defined below:

$$E ::= K \mid I \mid \lambda I.E \mid EE \mid (E, \ldots, E) \mid \textbf{if } E \textbf{ then } E \textbf{ else } E$$

$$T ::= \textbf{let } I = E \mid E \mid TT$$

where upper case letters denote syntactic categories, bold case denotes keywords of the language and other characters denote syntactic delimiters. E is the syntactic category of program expressions. K is the category of constants such as numbers, characters and character strings. The latter are delimited by quote characters as in '*thisisastring*'. I is the category of program identifiers which are sequences of alpha-numeric and underscore characters. T is the category of program transactions.

A basic program expression is one of the following: a constant such as 10 or 'y'; an identifier; a λ-function such as $\lambda i.i$ which is the identity function; an application such as $(\lambda i.i)10$ where the first sub-expression is referred to as the *operator* and the second sub-expression is referred to as the *operand*; an n-tuple $(e_1, \ldots, e_n)$ where the sub-expressions e_i $i = 1, n$ are referred to as components of the n-tuple and e_j $1 \leq j \leq n$ is the j^{th} component; or a conditional expression such as **if** $p(x)$ **then** $f(x)$ **else** $g(x)$. Individual sub-expression may be parenthesized in order to clarify the meaning of an expression.

The Λ-calculus provides the programmer with a primitive interactive environment. An entire programming session or *transactions* T is a sequence of interleaved definitions of the form **let** $I = E$ and expressions E. The former transaction introduces a new named value and the latter evaluates the expression in the current context of named values.

3.3.2 Computational Values

The operational semantics of the Λ-calculus is defined in terms of a state transition machine whose states consist of structured collections of computational values. This section defines the computational values which can occur during Λ-calculations; each type of value is defined in terms of constructors which produce values of the data type and operations over the data type.

The boolean values *true* and *false* occur in Λ-calculations. The standard propositional operators: and (infix &); or (infix |); and not (prefix $\neg$) are used to combine boolean values.

Sequences of values occur as computational values, most notably as components of the Λ-machine states. A sequence is either: [] when it is the empty sequence; or, $v : l$ which is the sequence constructed from a value v and a sequence l. The value v is referred to as the *head* of the sequence and l is referred to as the *tail* of the sequence. The notation $[v_1, v_2, \ldots, v_n]$ is shorthand for $v_1 : (v_2 : (\ldots : (v_n : []) \ldots))$. A list homomorphism [35] is constructed using the operator *lh*, if $\otimes$ is a binary operator, f is a unary operator and v is a value then:

$$lh(\otimes)(f)(v)([v_1, v_2, \ldots, v_n]) = f(v_1) \otimes f(v_2) \otimes \ldots \otimes f(v_n) \otimes v$$

Lists are appended using the infix binary operator $+\!\!+$, for example:

$$[v_1, \ldots, v_n] +\!\!+ [w_1, \ldots, w_m] = [v_1, \ldots, v_n, w_1, \ldots, w_m]$$

An *environment* is a structured collection of associations between values. An environment is either: $\{\}$ which is the null environment and contains no associations; $v_1 \mapsto v_2$ which associates the *key* v_1 with the *value* v_2; or $e_1 \oplus e_2$ where e_1 and e_2 are both environments and $\oplus$ is an associative constructor with left and right identity $\{\}$. The set of keys which are contained in an environment are referred to as the *domain* of the environment and is produced using the operator *dom*. The set of values which are contained in an environment are referred to as the *range* of the environment and are produced using the operator *ran*.

Given an environment e and a key k, a value which is associated with k by e is produced using the infix operator $\bullet$. The value ϵ represents failure to find a key in an environment. This operator is defined by case analysis on the structure of the environment e:

$$\{\} \bullet e = \epsilon$$

$$(v \mapsto v') \bullet k = \begin{cases} v' & \text{when } v = k \\ \epsilon & \text{otherwise} \end{cases}$$

$$(e_1 \oplus e_2) \bullet k = \begin{cases} e_1 \bullet k & \text{when } e_2 \bullet k = \epsilon \\ e_2 \bullet k & \text{otherwise} \end{cases}$$

The result of looking the key k up in the environment e is either the rightmost value associated with k in e or ϵ if e does not associate k with any value. An environment may associate a given key with more than one value. The rightmost value is returned by the lookup operator $\bullet$ and environments in the Λ-calculus are said to "shadow on the right". The distinguished value ϵ is returned when an environment does not associate a given key with any value. The e component of the Λ-machine contains an environment which associates identifiers (strings) with values. The binary infix operator ? is used to combine values: the value $v_1?v_2$ is v_1 when $v_2 = \epsilon$ and is v_2 otherwise.

The operator *eh* constructs environment homomorphisms. If $\otimes$ and $\odot$ are binary operators, v is a value and $e = (i_1 \mapsto v_1) \oplus (i_2 \mapsto v_2) \oplus \ldots \oplus (i_n \mapsto v_n)$ then:

$$eh(\otimes)(\odot)(v)(e) = (i_1 \odot v_1) \otimes (i_2 \odot v_2) \otimes \ldots \otimes (i_n \odot v_n) \otimes v$$

The postfix operator $[v_1/v_2]$ is applied to an environment and produces a new environment in which all occurrences of the key v_2 has been replaced with key v_1. Given a set of keys S and an environment e, the expression e/S denotes an environment which is the same as e except that all the keys in S have been removed; the expression $e\backslash S$ denotes an environment which is the same as e except that all the keys which are not in S have been removed. Environment difference is performed by infix binary $-$ and is defined as follows:

$$(e_1 - e_2) \bullet v = \begin{cases} e_1 \bullet v & \text{when } e_1 \bullet v \neq e_2 \bullet v \\ \epsilon & \text{otherwise} \end{cases}$$

Tuples of values are denoted $(v_1, v_2, \ldots, v_n)$ where the *length* of the tuple is n. A single tuple may be denoted $\tilde{v}$, its length is produced using the operator $\#$ and each individual tuple component is denoted $\tilde{v}_i$ $1 \leq i \leq \#(\tilde{v})$. The infix binary operator : (also defined for sequences) is used to construct tuples:

$$v_0 : (v_1, \ldots, v_n) = (v_0, v_1, \ldots, v_n)$$

The infix binary operator $+\!\!+$ (also used for sequences) is used to append tuples:

$$(v_1, \ldots, v_n) +\!\!+ (w_1, \ldots, w_m) = (v_1, \ldots, v_n, w_1, \ldots, w_m)$$

A *function closure* is created when a λ-expression is performed and is denoted $<i, e, b>$ where i is an identifier, e is an environment (referred to as the *closure environment*) and b is an expression (referred to as the *closure body*).

A *program point* is a function closure which is associated with a machine state. It is denoted $<f, d>$ where f is a function closure and d is a dump.

A *machine address* or *program location* is a value which is denoted α. A *heap* is an environment which associates addresses with arbitrary values.

The Λ-machine is driven by a machine component which is a sequence of program expressions and *machine instructions*. A machine instruction is one of the following: an *apply* instruction @; a *test* instruction $[e_1, e_2]$ where e_1 and e_2 are both program expressions; a *tuple* instruction $[n]$ where n is a positive integer; or a *define* instruction $<i>$ where i is an identifier (string).

In the calculation rules we will often want to represent patterns of the following types:

$$(i_1 \mapsto v_1) \oplus (i_2 \mapsto v_2) \oplus \ldots \oplus (i_n \mapsto v_n)$$

$$(i_1 \mapsto <j_1, e, b_1>) \oplus (i_2 \mapsto <j_2, e, b_2>) \oplus (i_n \mapsto <j_n, e, b_n>)$$

i.e. arbitrary sized environments mapping different names to values or closures. These patterns introduce considerable notational clutter in a rule and this is reduced by the following notational conventions. The binding constructor $(\mapsto)$ is overloaded so that it can construct arbitrary sized environments when both its arguments are tuples of equal length:

$$\tilde{i} \mapsto \tilde{v} = (\tilde{i}_1 \mapsto \tilde{v}_1) \oplus (\tilde{i}_2 \mapsto \tilde{v}_2) \oplus \ldots \oplus (\tilde{i}_n \mapsto \tilde{v}_n)$$

The closure constructor $<\_, \_, \_>$ is overloaded so that it can construct a tuple of closures when any of its arguments are tuples. When more than one of its arguments are tuples they all must be of the same length:

$$<\tilde{i}, e, \tilde{b}> = (<\tilde{j}_1, e, \tilde{b}_1>, <\tilde{j}_2, e, \tilde{b}_2>, \ldots, <\tilde{j}_n, e, \tilde{b}_n>)$$

Sets may also be constructed using tuple notation: $\{\tilde{v}\} = \{\tilde{v}_1, \ldots, \tilde{v}_n\}$. The operator ? is overloaded as follows: $?(\tilde{v}) = \tilde{v}_1 ? \ldots ? \tilde{v}_n$. The infix lookup operator $\bullet$ is overloaded as follows: $e \bullet \tilde{i} = (e \bullet \tilde{i}_1, \ldots, e \bullet \tilde{i}_n)$.

3.3.3 The Λ-Machine

The Λ-machine is composed of a collection of machine states and a transition function. The states of the machine have the general form (s, e, c, d, h) where: s is a sequence of denotable values referred to as the *stack*; e is an environment; c is a sequence of program expressions and machine instructions referred to as the *control*; d is either a 4-tuple (s, e, c, d) or the value $()$ and is referred to as the *dump*; and h is a heap.

The initial states of the machine have the following form $([], e, t, (), \{\})$ where e is an environment which associates program identifiers with values; e is the initial environment and defines all *built in* identifiers and their values. The initial control t is a sequence of transactions being all the interactions between the programmer and the Λ-calculus environment for an entire programming session.

The terminal states of the machine have the following form $(v : \_, \_, [], (), \_)$ where v is the outcome of the entire session; $\_$ denotes values which may be anything.

Evaluation by machine proceeds by performing the entire expression at the head of the control and placing the outcome at the head of the stack. Where the outcome of an expression depends upon the outcomes of some or all of its sub-expressions, these are performed to produce sub-outcomes on the stack and these are then consumed in order to produce the overall outcome.

Each category of program expression has an evaluation rule which describes how it is handled when it occurs at the head of the control. Some expression types are handled immediately (for example constants) by consuming the head of the control and "pushing" a new value at the head of the stack. Other expression types require their sub-expressions to be evaluated before they can proceed (for example **if**-expressions) in which case the expression is "popped" from the control, a machine instruction is pushed and then the sub-expression(s) is (are) pushed onto the control. When a machine instruction occurs at the head of the control, a sequence of sub-outcomes will have been pushed at the head of the stack. The evaluation rule for the machine instruction defines how the sub-outcomes are consumed and an overall outcome is produced at the head of the stack.

The stack component is used to hold the outcomes of program expressions. The environment component is used to hold the values of current program identifiers which are currently *in scope*, *i.e.* which may be referenced by the expression which is currently being performed.

The control component is used to contain a sequence of definitions and expressions which are to be performed. The head of the control is the next expression to be performed. Where sub-expressions must be performed before the outcome of an expression can be produced, the expression is *flattened* at the head of the control by removing it and pushing a machine instruction before pushing the sub-expressions.

The current machine components define a *context* for the evaluation of the expression at the head of the control. Certain instructions (for example @) require that the current context be temporarily suspended and replaced with a new context. The dump component is used to hold the suspended contexts. When the control of the current context is exhausted, the most recently suspended context is restored from the dump.

The heap component is used to associate program locations and their contents. Program locations are sometimes referred to as *variables* in imperative languages. The Λ-calculus transition function is defined as follows:

$$(s, e, k : c, d, h) \longmapsto (k : s, e, c, d, h) \qquad R_1$$

$$(s, e, i : c, d, h) \longmapsto ((e \bullet i) : s, e, c, d, h) \qquad R_2$$

$$(s, e, \lambda i.b : c, d, h) \longmapsto (<i, e, b> : s, e, c, d, h) \qquad R_3$$

$$(s, e, e_1 e_2 : c, d, h) \longmapsto (s, e, e_2 : e_1 : @ : c, d, h) \qquad R_4$$

$$(<i, e', b> : v : s, e, @ : c, d, h) \longmapsto ([], e' \oplus (i \mapsto v), [b], (s, e, c, d), h) \qquad R_5$$

$$(v : \_, \_, [], (s, e, c, d), h) \longmapsto (v : s, e, c, d, h) \qquad R_6$$

$$(s, e, (e_1, \ldots, e_n) : c, d, h) \longmapsto (s, e, e_1 : \ldots : e_n : [n] : c, d, h) \qquad R_7$$

$$(v_n : \ldots : v_1 : s, e, [n] : c, d, h) \longmapsto ((v_1, \ldots, v_n) : s, e, c, d, h) \qquad R_8$$

$$(s, e, \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_2 : c, d, h) \longmapsto (s, e, e_1 : [e_2, e_3] : c, d, h) \qquad R_9$$

$$(\mathit{true} : s, e, [e_1, \_] : c, d, h) \longmapsto (s, e, e_1 : c, d, h) \qquad R_{10}$$

$$(\mathit{false} : s, e, [\_, e_2] : c, d, h) \longmapsto (s, e, e_2 : c, d, h) \qquad R_{11}$$

$$(s, e, \textbf{let } i = x : c, d, h) \longmapsto (s, e, x : <i> : c, d, h) \qquad R_{12}$$

$$(v : s, e, <i> : c, d, h) \longmapsto (s, e \oplus (i \mapsto v), c, d, h) \qquad R_{13}$$

The calculation rules are described as follows. Rule R_1 shows the evaluation of a constant k. Rule R_2 shows the evaluation of an identifier i which is looked up in the current environment e. Rule R_3 shows the evaluation of a λ-function, a function closure is pushed onto the stack containing the function argument i, the current environment e and the function body b.

Rule R_4 shows the start of the evaluation of an application expression. The expression is popped from the control, the machine instruction @ is pushed onto the control followed by the operator and operand expressions. Rule R_5 shows the start of an application. The machine instruction @ is at the head of the control, a closure is at the head of the stack above a value v. The closure is the result of performing the operator expression and the value is the result of performing the operand expression. The closure is applied by constructing a new machine context (*i.e.* machine state) in which the closure environment is extended with a binding between the formal parameter i and the actual parameter v, and the control is the body of the closure. The current context is saved on the dump. Rule R_6 shows what happens when the current control is exhausted. The result of performing a program expression (the body of a function closure) is at the head of the stack and the previously saved context is the dump. The machine state becomes the saved context and the value is "returned" by popping from the *before* state and pushing it onto the stack of the *after* state.

Rule R_7 shows the start of the evaluation of a tuple-expression. In order to construct a tuple, each of the component values must be produced by performing the sub-expressions. The tuple expression is popped from the control and the machine instruction $[n]$ is pushed onto the control before each of the sub-expressions. The integer n is the length of the tuple. Rule R_8 shows the construction of a tuple. Each of the tuple's component values are head on the stack and the machine instruction $[n]$ is at the head of the control. The machine instruction is popped from the control, the n values are popped from the stack and the tuple $(v_1, \ldots, v_n)$ is pushed onto the stack.

Rule R_9 shows the start of the evaluation of an **if**-expression. The outcome of the expression depends upon the outcome of the predicate expression e_1. If this produces *true* then the *consequent* expression e_2 is performed otherwise the *alternative* expression e_3 is performed. The **if**-expression is popped from the control and the machine instruction $[e_2, e_3]$ is pushed onto the control before the predicate expression e_1. Rule R_{10} describes the case where the predicate expression produces *true*: the machine instruction at the head of the stack is replaced by the consequent expression and *true* is popped from the stack. Rule R_{11} describes the case where the predicate expression produces *false*: the instruction at the head of the control is replaced by the alternative expression and *false* is popped from the head of the stack.

Rule R_{12} describes the start of a top level definition. The head of the control is popped and the machine instruction $<i>$ is pushed onto the control before the program expression e. Rule R_{13} defines the addition of a top level definition to the current environment: e is extended with a binding between the identifier i and the value v.

The Λ-machine executes by performing repeated application of the transition function $\longmapsto$. The function $\longmapsto^n$ represents n applications of $\longmapsto$. The transition function can be extended to its transitive closure $\longmapsto^*$ where $\sigma_1 \longmapsto^* \sigma_2$ if and only if $\sigma_2 = \longmapsto^n (\sigma_1)$ for some integer n. Given a program expression $x \in E$, its *outcome* with respect to the *context* (s, e, c, d, h) is v, written $x \Rightarrow v$, when $(s, e, x : c, d, h) \longmapsto^* (v : s, e, c, d, h)$.

3.3.4 Extending the Λ-machine

There are three major ways of extending a λ-calculus which is given an operational semantics using a state transition machine:

1. Add *built-in* operators and values. Such an extension involves defining an algebra in terms of its carrier and operators and adding the entire algebra to the machine via the initial environment. Such an algebra of values is likely to have a collection of basic constant values which are added to the syntactic category K, a collection of constructor operators which create new values from existing ones, and a collection of accessor operators which retrieve components of constructed values. The accessors and constructors are added to the initial environment. Such an extension does not affect the basic evaluation rules of the language and can be incorporated using the existing rules for identifier lookup and operator application.

2. Add new evaluation rules. Such an extension may be necessary because the outcome of performing a new syntactic expression cannot be easily described using operator/operand combination.

3. Define syntactic *sugar*. New program expressions can be defined in terms of a translation to combinations of other types of expression whose operational semantics is described using the existing transition rules. This is often implemented in languages by having a pre-processing or macro-expansion phase before compilation or interpretation.

There is often a trade-off between the three types of extension, since a given extension may be achieved in more than one way. The trade-off may include concerns such as clarity of exposition, efficiency and simplicity of the semantic rules. The following paragraphs give examples of each type of extension as included in the Λ-calculus.

The Λ-calculus provides built-in integers with all usual operators such as $+$ and $>$. Sets are also built-in: $\emptyset$ is the empty set; $\{\_\}$ constructs singleton sets; and $\cup$ is infix set union. The infix predicate $\in$ is used to test for set membership and infix $-$ is used for set difference. Strings are built-in and are treated as atomic. An inexhaustible supply of atomic *tags* is built-in to the machine. Each tag is distinct and may be compared with another tag using binary infix $=$.

Builtin operators are a distinct category of computational value and have application rules which are different from that for closure application. If f is a value of type *built-in operator*, and v is a value in the domain of f and the value $f(v)$ is defined by a predefined algebra (such as integers and arithmetic operators) then the calculation rule is:

$$(f : v : s, e, @ : c, d, h) \longmapsto (f(v) : s, e, c, d, h) \;\; R_{14}$$

As shown in R_{14}, the application of such a built-in operator behaves like the application of a closure except that the outcome is known directly from the identity of the operator and the operand.

Operators which test tuple-hood and access the components of tuples are built-in to the Λ-calculus. The predicate *istuple* is true of an n-tuple. The operators *1st*, *2nd*, ... access the components of a tuple.

Other types of built-in operator behave differently to closure application because they affect components of the machine other than the stack and the control in the manner defined by R_{14}. Typically each operator or new program expression type of this form will require a new calculation rule; whereas R_{14} covers enriching the Λ-calculus with values and operators from many independent algebras.

An example of a built-in operator which requires a new calculation rule is *halt*. This operator is applied to a value and will immediately cause the current calculation to abort, producing a terminal machine state which contains the supplied value at the head of the stack:

$$(halt : v : \_, @ : \_, \_, \_) \longmapsto ([v], \{\}, [], (), \{\}) \;\; R_{15}$$

Programming in the basic Λ-calculus is made difficult because the language does not provide any high level programming constructs. One way to add such constructs to the language is to define them as syntactic sugar, *i.e.* for each new type of construct define a translation rule which transforms instances of the new construct into expressions for which evaluation is well defined. Evaluation is well defined for an expression when there is a sequence of state transitions which will evaluate it. Evaluation is also well defined for an expression when it can be translated, by applying a translation rule, to an expression for which evaluation is well defined. We describe four types of sugar used in Λ-calculus programs: infix operators; currying; λ-expressions with patterns as formal parameters; and **let**-expressions.

Infix operators are usually binary operators, *e.g.* $1+2$, $x \in S$. The basic Λ-calculus syntax is extended to allow infix binary operators: $E\,O\,E$ where O is a suitable syntactic category for binary operator identifiers. The translation rule for infix application translates each instance of this new expression category, $e_1\,o\,e_2$ to $o(e_1, e_2)$ whose evaluation is well defined.

An operator is often characterised by the number of arguments which it expects, for example $+$ expects two arguments. A *curried* operator may be applied to less arguments than it expects [28]; the result is an operator which can be supplied with the remaining arguments at some later date. For example, $+$ may be applied to 1 to yield the operator which is supplied with an integer and adds 1 to it. The syntax of the basic Λ-calculus is extended to allow curried λ-functions to be expressed as follows: $\lambda i_1 i_2 \ldots i_n.e$. The translation rule for curried functions inserts the necessary λ's: $\lambda i_1.\lambda i_2.\ldots\lambda i_n.e$. Application expressions associate to the left so that the concrete syntax $fghv$ produces the following abstract syntax tree $((f(g))(h))(v)$. Where a function expects a tuple of values it may be partially applied using the following notational convention. The application $f(x,\_)$ where occurrences of $\_$ represent the missing arguments, represents a function $\lambda y.f(x,y)$. The use of $\_$ in such an application means that we do not have to introduce any unnecessary identifiers.

In programs, functions are often applied to structured values which are then decomposed within the body of the function. The amount of program code necessary to do this can be greatly reduced by providing a simple pattern mechanism. The formal parameter of a λ-function may be a pattern which describes the structure of the actual parameter and contains identifiers which are bound from the corresponding components of the actual parameter value when the function is applied. For example, the function which swaps the order of a pair of values can be written $\lambda(x,y).(y,x)$. The syntactic category P of patterns is:

$$P ::= K \mid I \mid CP \mid (P,\ldots,P)$$

where C is the syntactic category of data constructors (such as : for lists). In order to desugar a Λ-function which contains a pattern in the formal parameter position, it is necessary to decide what to do when the function is applied to a value which does not match the pattern. We assume that the function returns a special value denoted *fail* which is built-in to the initial environment. The translation rules for patterns are as follows:

Sugar	Desugared
$\lambda k.e$	$\lambda i.$**if** $i = k$ **then** e **else** *fail*
$\lambda i.e$	$\lambda i.e$
$\lambda cp.e$	$\lambda i.$**if** $isc(i)$ **then** $(\lambda p.e)(stripc(i))$ **else** *fail*
$\lambda(p_1,\ldots,p_n).e$	$\lambda i.$ **if** $istuple(i)\&\#(i) = n$ **then** $(\lambda p_1.\ldots.\lambda p_n.e)(1st(i))\ldots(nth(i))$ **else** *fail*
$\lambda p_1 p_2 \ldots p_n.e$	$\lambda p_1.\lambda p_2.\ldots\lambda p_n.e$

where i is an identifier introduced by the translation rules and which is guaranteed to be unique; *isc* is a predicate which tests whether or not its argument is constructed using the constructor c; *stripc* is an operator which uncovers the argument to the constructor c given a data value constructed using c.

Functions may be *overloaded* using the infix binary operator $|$. Given two functions f_1 and f_2, the operator $f_1 \mid f_2$ behaves as follows:

$$(f_1 \mid f_2)(x) = \begin{cases} f_1(x) & \text{when } f_2(x) = \mathit{fail} \\ f_2(x) & \text{otherwise} \end{cases}$$

Patterns may be used in closures, for example $<(),e,b>$ and $<x:l,e,b>$. This notation is unsugared as shown for λ-functions above. For more details about implementing pattern matching in functional programming languages see [59].

In order to refer to a value more than once, it is necessary to apply a λ-function to it so that the value is associated with an identifier. The identifier can then be referred to as many times as is necessary in the body of the function. This programming feature occurs often enough for a new syntactic constructed to be invented for its expression. A **let**-expression has the following form: **let** $b_1 \ldots b_n$ **in** e where each b_i is a *binding* and e is an expression referred to as the body of the **let**-expression. Each binding associates one or more identifiers with values for the scope of the evaluation of the body of the **let**-expression. Each identifier bound by the **let**-expression is associated with its corresponding value in parallel with all other identifiers in the **let**-expression; so the value of each identifier cannot be dependent on the value of any other identifier.

A simple binding has the form $p = e$ where p is a pattern and e is an expression. The intended meaning of the expression **let** $p = e_1$ **in** e_2 is that e_1 is performed producing a value v which is matched against pattern p causing identifiers to be bound to components of v during the performance of the expression e_2. We assume that the value v always matches pattern p, *i.e.* the value *fail* is never produced. A function binding has the form $ip_1p_2 \ldots p_n = e$ which is equivalent to the simple binding $i = \lambda p_1 p_2 \ldots p_n.e$.

A top level definition introduces a new binding. A top level definition may establish a new value or a new function. A top level value binding is of the form: **let** $p = e$. A top level function definition introduces bindings b of the form $i\ p_1 \ldots p_n = e$ and may use the $|$ operator to define the function by cases: **let** $b_1 \mid b_2 \mid \ldots \mid b_n$ where the initial identifier in each binding b_i must be the same.

A **let**-expression may have single or multiple bindings. A **let**-expression with a single binding: **let** $p = e_1$ **in** e_2 is desugared as follows: $(\lambda p.e_2)e_1$ and a **let**-expression with multiple bindings: **let** $p_1 = e_1 \ \ldots \ p_n = e_n$ **in** e is desugared as follows:

$$(\lambda(p_1, \ldots, p_n).e)(e_1, \ldots, e_n)$$

Since a **let**-expression with multiple bindings performs them in parallel, such expressions must be nested to achieve sequential bindings. The following style is adopted for sequential bindings:

let b_1 **in**
let b_2 **in**
...
let b_n
in e

which allows identifiers bound by binding b_i to be referenced in the value part of the binding b_j such that $j > i$.

3.4 Recursive Values

Cyclic or recursive values occur naturally in programs, for example a function or procedure which calls itself or a linked list which contains a pointer to itself. It is necessary for the Λ-calculus to support recursive values since object-oriented programming languages support cycles in the form of classes which refer to themselves by name, objects which refer to themselves as *self* (or some other distinguished identifier) and methods which call themselves directly.

Systems based on λ-notation often support recursive values by using a family of operators referred collectively as Y [28]. The operator Y is applied to a function f and finds the fixed point of f. The fixed point of a function is that value which, when supplied to the function, causes the function to behave as if it were identity. For example, if f is a function and v is its fixed point then $f(v) = v$. The operator Y finds the fixed point so: $v = Y(f)$ or equivalently: $f(Y(f)) = Y(f)$.

The operator Y is useful when creating recursive values because the function f may be a λ-function whose formal parameter allows the fixed point to be referenced within the body of f. Consider the definition of the factorial function:

$$\textbf{let}\ \mathit{Fact}\ \mathit{fact}\ n = \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n * (\mathit{fact}(n-1))$$

The function *Fact* is parameterized with respect to a function *fact* and an integer n. The value of *fact* is intended to be the factorial function. The result of applying Y to the function *Fact* is the fixed point, *i.e.* the value which is supplied as the value of *fact* will be the same as the value returned by $Y(\mathit{Fact})$.

Another example of constructing fixed points using Y is the following: $Y\lambda t.(1,t)$ which creates a binary tree whose left branches all contain the integer 1 and whose right branches lead to itself.

As mentioned above, the operator Y represents a family of definitions, each of which is a fixed point operator. The following two definitions are examples of fixed point operators using pure λ-calculus notation:

$$\textbf{let}\ Y = \lambda g.(\lambda x.g(\lambda y.(x(x))(y)))\lambda x.g\lambda y.(x(x))(y)$$

$$\textbf{let}\ Y = \lambda g.(\lambda x.g(x(x)))\lambda x.g(x(x))$$

The various definitions for Y differ with respect to the class of functions which they find fixed points for, *i.e.* some are more universal fixed point operators than others. Rather than provide recursive definitions within the Λ-calculus using a definition such as those above, we define a built-in operator and an evaluation rule which succinctly describes the construction of fixed points:

$$\frac{(s, e' \oplus (i \mapsto v), [b], (s,e,c,d), h) \longmapsto (v : s, e' \oplus (i \mapsto v), [\,], (s,e,c,d), h')}{(Y : <i, e', b> : s, e, @ : c, d, h) \longmapsto (v : s, e, c, d, h')}\ R_{16}$$

The calculation rule R_{16} describes the construction of fixed points by defining that the result of applying Y to any closure is a result v which is produced when the closure is applied to v.

A question arises as to when it is possible to find the fixed point of a function and whether or not the fixed point is unique. These questions are beyond the scope of this thesis and are covered at length in many sources describing functional programming and programming language semantics such as [69].

It is useful to define syntactic sugar in order to establish recursive definitions. A **letrec**-expression is a construct for establishing recursive bindings for data and functions for the scope of the evaluation of an expression. The format of a **letrec**-expression is: **letrec** $b_1 \ldots b_n$ **in** e where b_i are bindings as described for **let**-expressions in §3.3.4 and e is the body of the **letrec**-expression.

The operational semantics of such an expression is that all of the bindings are established using mutual recursion for the scope of the body. This is achieved by desugaring a **letrec**-expression as follows. Let the following be the result of desugaring each of the bindings to produce simple bindings:

$$\textbf{letrec } i_1 = e_1 \dots i_n = e_n \textbf{ in } e$$

which is translated to produce:

$$\textbf{let } (i_1, \dots, i_n) = Y\lambda(i_1, \dots, i_n).(e_1, \dots, e_n) \textbf{ in } e$$

the degenerate case of such an expression is

$$\textbf{let } i = Y\lambda i.e \textbf{ in } e'$$

Two examples of **letrec**-expressions are:

$$\textbf{letrec } \mathit{fact}(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * (\mathit{fact}(n-1))$$

$$\textbf{letrec } t = (1, t)$$

One way to think about the construction of recursive values is by *tying knots* which is performed by an SECD machine as follows. When Y is applied to a function, a fresh program location is allocated. The function is applied to the program location which initially contains a special value. Two outcomes are possible: either the function returns a value in which case the program location is updated to contain the returned value; or, the function attempts to perform an operation which requires the contents of the program location. We can tell that such an operation is illegal since the location contains a special value; under these circumstances the program diverges.

Top level definitions introduce recursive definitions. The rule for top level definition evaluation is modified as follows:

$$\frac{(s, e \oplus (i \mapsto v), x : c, d, h_1) \longmapsto (v : s, e \oplus (i \mapsto v), c, d, h_2)}{(s, e, (\textbf{let } i = x) : c, d, h_1) \longmapsto (s, e \oplus (i \mapsto v), c, d, h_2)} \; R_{12}$$

3.5 Environment Reification

The Algol family of programming languages have rules dealing with the association between the binding occurrence and the usage occurrence of identifiers which follows that of the λ-calculus, *i.e.* β-reduction. The object-oriented family of programming languages has introduced new rules which dictate where an identifier is introduced and where it may be referenced within the program text. These rules are broadly gathered together under the term "inheritance".

Chapter 2 gives many examples of object-oriented programming scoping rules and identifies a new type of scoping rule referred to as *inheritance* scoping. Inheritance scoping allows attribute and method identifiers which are defined in super-classes to be legally referenced within all their sub-classes. Inheritance scoping differs from textual scoping because classes are not textually nested and differs from dynamic scoping because the classes are not active in any sense when the identifier references take place.

The scope of identifiers is controlled by the mechanisms which affect the environment component of the Λ-machine. Each time an identifier is referenced, its value is found by looking the identifier up in the current environment. Environments are associated with function closures and are saved and restored on function call and return via the dump component.

We claim that any new identifier scoping rules can be based upon access to the environment component of the machine. In order to support such access, the Λ-calculus is extended to allow programmer control over environments. The programmer is able to construct environments, get at the current machine environment interrogate environments and replace the current environment with a new one.

Builtin operations and values are supplied in order to create and manipulate environments. These operators are: $\{\}$ the empty environment; infix binary $\mapsto$; infix binary $\oplus$; infix binary $\bullet$; lookup failure ϵ; infix binary environment difference $-$; environment domain *dom*; environment range *ran*; infix binary / and \; postfix $[\_/\_]$; environment homomorphisms *eh*.

Access to environments within the machine is provided by the built-in operators R pronounced *reify* and I pronounced *install*:

$$(R : <\_, e', \_> : s, e, @ : c, d, h) \longmapsto (e' : s, e, c, d, h) \ R_{17}$$

$$(I : (e', <i, \_, b>) : s, e, @ : c, d, h) \longmapsto (<i, e', b> : s, e, c, d, h) \ R_{18}$$

Rule R_{17} defines the reify operator which is applied to a function closure and returns the closure environment. Rule R_{18} defines the install operator which is applied to a pair (e, f) containing an environment e and a function closure f. The result is a new function closure whose argument and body are those of f and whose closure environment is e.

The install and reify operators provide flexible access to the environments which control identifier scopes in the Λ-machine. In order to capture the current machine environment the following expression is performed: $R\lambda().()$, the λ-expression is evaluated to produce a function closure which contains the current environment; the reify operator accesses and returns the current environment. In order to update the current environment to be e for the scope of the evaluation of a particular expression b the following expression is performed: $I(e, \lambda().b)()$ which creates a function closure, replaces its environment with e and activates it.

Recursive function definition allows a function to access its own environment:

$$\textbf{let } \mathit{doibind}(i) = i \in \mathit{dom}(R(\mathit{doibind}))$$

The function *doibind* is applied to a name i and returns *true* when i was in scope when *doibind* was created. Recursive definitions also allow a function to access that part of the environment which was created when the function was applied, for example:

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

The function *myenv* is applied to a triple and returns an environment in which the identifiers x, y and z are bound to the components of the triple. This is achieved using environment reification: when the function is applied, the closure environment for *myenv* is extended with bindings for x, y and z and is installed as the current machine environment. The body of *myenv* reifies the current environment which contains the bindings for x, y and z and reifies the closure environment of *myenv*. The latter is subtracted from the former which produces an environment which just contains the bindings which were established when the function closure was applied. For example:

$$myenv(1, 2, 3) \Rightarrow (\text{`}x\text{'} \mapsto 1) \oplus (\text{`}y\text{'} \mapsto 2) \oplus (\text{`}z\text{'} \mapsto 3)$$

$$myenv(3, 2, 1) \Rightarrow (\text{`}x\text{'} \mapsto 3) \oplus (\text{`}y\text{'} \mapsto 2) \oplus (\text{`}z\text{'} \mapsto 1)$$

Inheritance scoping relies on being able to transfer attribute bindings from a super-class to a sub-class. The creation of attribute bindings by a super-class is very like the behaviour of *myenv* above. A sub-class is a function which contains free references to the inherited attribute identifiers, for example the function *scale* below is applied to a number n and scales the sum of x, y and z by n:

$$\textbf{let } scale(n) = n * (x + y + z)$$

Rather than supply the values of x, y and z as arguments, they can be shared with *myenv* as follows:

$$I(R(scale) \oplus (myenv(1, 2, 3)), scale)(4) \Rightarrow 24$$

where the closure environment of *scale* is extended with the environment which is returned by $myenv(1, 2, 3)$ and the new closure is applied to 4 producing $4 * (1 + 2 + 3) = 24$.

Programming languages often provide modules which are packages of bindings from names to values very like environments. Such languages will usually allow the modules to be *opened* for the textual scope of some program phrase. Languages may allow modules to be parameterized by other modules and provide a calculus of module operations. All of these facilities can be expressed using reified environments in the Λ-calculus.

3.6 Side Effects

Objects in object-oriented programming languages usually have *state i.e.* the outcome of sending an object a message depends upon the computational history of the object. This is not true of *pure* functional languages which have no state and have a property referred to as *referential transparency* where any expression can be replaced by any equivalent expression and the order of evaluation of sub-expressions is not important.

The state of an object is usually hidden within the object as a collection of attributes. Each attribute is a program location which may be updated by side effect. The attributes are usually private to the object and may only be updated by the object's methods.

Some object-oriented languages, for example Smalltalk, provide other types of state such as *class variables* and global variables. These program locations are not private to any object and may be referenced by many different parts of the program.

The Λ-calculus is not a pure functional programming language since it provides first class program locations which may be updated by side effect. Unlike most imperative programming languages the Λ-calculus provides very flexible access to program locations since they are implemented as first class data values which can be passed as arguments to functions, returned as results, bound to identifiers and even stored in locations. The locations are essentially the same as those provided by the functional programming language ML [34].

Program locations are created and manipulated using a collection of built-in primitive operators which are described in the following paragraphs.

A program location is created and initialised using the operator *loc*. The expression *loc*(10) creates and returns a new program location each time it is performed and initialises the location with 10.

The current value contained in a program location is produced by the operator !. For example:

$$\textbf{let } l = loc(10)$$

$$!l \Rightarrow 10$$

A program location is updated by side effect using the built-in binary infix operator :=. When a program location is passed as an argument to a function or returned as a result, it is never implicitly copied. Therefore, the location may exist at many points within the current state of an evaluating program. When the := operator is used to update one occurrence of the program location, all occurrences of the location within the machine are updated simultaneously. For example:

$$\textbf{let } l_1 = loc(10)$$

$$\textbf{let } l_2 = l_1$$

$$l_1 := 20 \Rightarrow 20$$

$$!l_1 \Rightarrow 20$$

$$!l_2 \Rightarrow 20$$

A location l may be explicitly copied by creating a new location and initialising it with the contents of the original location: $loc(!l)$. This example shows *shallow* copying where the program location is copied but any locations contained in the value $!l$ are not copied. Alternatively *deep* copying will also copy all of the locations contained in $!l$. This can be supported in the Λ-calculus by providing built-in data type predicates, accessors and constructors for all the possible computational values and using these to construct deep copying operations for each data type.

A built-in predicate, *isloc*, is provided for testing whether or not a data value is a program location.

The order of Λ-evaluation is important due to the inclusion of program locations. In addition, some program expressions are evaluated for their side effect rather than for their value. The syntactic construct $E; E$ is provided for specifying the order of expression evaluation.

Program locations are implemented in the Λ-machine using a heap which is an environment associating *addresses* α with the contents of the program location. The heap is not saved on the dump when a function closure is applied and is not restored on return from the application. Any program locations which are constructed have indefinite scope and exist as long as there is the possibility of reference. This is in contrast with block structured imperative languages which automatically allocate program locations on entry to a block and reclaim the storage when the block is exited.

The calculation rules describing program locations are as follows:

$$(loc : v : s, e, @ : c, d, h) \longmapsto (\alpha : s, e, c, d, h \oplus (\alpha \mapsto v)) \text{ when } \alpha \notin dom(h) \qquad R_{19}$$

$$(! : \alpha : s, e, @ : c, d, h) \longmapsto ((h \bullet \alpha) : s, e, c, d, h) \qquad R_{20}$$

$$((:=) : (\alpha, v) : s, e, @ : c, d, h) \longmapsto (v : s, e, c, d, h \oplus (\alpha \mapsto v)) \qquad R_{21}$$

$$(isloc : v : s, e, c, d, h) \longmapsto \begin{cases} (true : s, e, c, d, h) & \text{when } v \in dom(h) \\ (false : s, e, c, d, h) & \text{otherwise} \end{cases} \qquad R_{22}$$

$$(s, e, (e_1; e_2) : c, d, h) \longmapsto (s, e, e_1 : e_2 : c, d, h) \qquad R_{23}$$

Rule R_{19} describes the creation of new program locations. A new address α is coined and the heap is extended with a binding from the new address to the initial value v. The result left at the top of the stack is the address α.

Rule R_{20} describes looking up the contents of a program location in the heap. Rule R_{21} describes location update. An update causes the current binding for the address α to be shadowed in the heap.

Rule R_{22} describes testing for a program location. A value is a program location when it is an address in the domain of the heap. Finally, rule R_{23} describes sequencing of expressions.

3.7 Program Points

The flow of control in sequential object-oriented programming languages is usually determined by the message passing construct. A message is sent to an object, the caller suspends until the receiver handles the message and returns a value. When the sender receives the result of the message it resumes the suspended calculation.

The message passing model of computation is similar to function call and return, except that the particular function which is activated depends upon an object which receives the message.

Object-oriented languages typically provide a control mechanism which is used to handle exceptional conditions such as errors. The exception mechanism provides a way to jump back over several layers of suspended computations discarding them in the process. Typically, a condition is detected which means that the current calculation cannot continue; all the suspended calculations which are waiting for the result of the current calculation must be discarded and the control must jump back to a position where the condition can be safely handled.

Such a mechanism is provided by the Λ-calculus in the form of *program points*. A program point contains two items of data: a function closure and a machine state. When the program point is applied, it behaves like a closure except that the machine state to which it will return is that in the program point rather than the current machine state. Any suspended calculations which are held in the dump component of the current machine state are discarded.

A program point is a new computational value and is represented as $<f, d>$ where f is a function closure and d is a dump. A program point is constructed using the built-in operator J [48] which is similar to the call/cc mechanism in Scheme [16] and is defined by the following calculation rule:

$$(J : f : s, e, @ : c, d, h) \longmapsto (<f, d> : s, e, c, d, h) \;\; R_{24}$$

which captures the current dump component of the machine in a program point. When a program point is applied, the current machine state is discarded and the dump from the program point is used:

$$(<<i, e, b>, d> : v : \_, \_, @ : \_, \_, h) \longmapsto ([], e \oplus (i \mapsto v), [b], d, h) \;\; R_{25}$$

As an example of how program points can be used to handle exceptions, consider the a division routine *div* which takes three arguments n, m and x. If m is not 0 then the result of *div* is n/m otherwise x is applied to n. The argument x is an exception handler and is used to abort the division calculation. The handler is supplied by the caller of *div* who must decide in advance what is to be done when this exception occurs:

$$\mathbf{let}\ \mathit{div}(n, m, x) = \mathbf{if}\ m = 0\ \mathbf{then}\ x(n)\ \mathbf{else}\ n/m$$

The operator *stdev* computes the standard deviation of a list of numbers. It takes two arguments: a list of numbers and the average of the list of numbers. The following expression:

$$\mathit{stdev}(l, \mathit{sum}(l)/\#l)$$

computes the standard deviation of non-empty sequences of numbers l. Unfortunately, we may not be able to guarantee that l is non-empty. One way of handling the case where l is empty is to use the *div* operator and to supply a suitable exception handler:

$$\mathit{stdev}(l, \mathit{div}(\mathit{sum}(l), \#l, J\lambda\_. - 1))$$

The supplied exception handler will return -1 when the standard deviation cannot be computed. To see that this works, consider the calculation which occurs, let $p = <<\_, e, -1>, d>$ and $b = \mathbf{if}\ m = 0\ \mathbf{then}\ x(n)\ \mathbf{else}\ n/m$ and $d = (s', e', c', d')$:

$$\begin{array}{l}
(s, e, l : l : \mathit{sum} : @ : l : \# : @ : (\lambda_. - 1) : J : @ : [3] : \mathit{div} : @ : [2] : \mathit{stdev} : @ : c, d, h) \longmapsto^{*} \\
\quad (\mathit{div} : (0, 0, p) : l : s, e, @ : [2] : \mathit{stdev} : @ : c, d, h) \longmapsto^{*} \\
\quad\quad ([], e \oplus (\text{‘}n\text{’} \mapsto 0) \oplus (\text{‘}m\text{’} \mapsto 0) \oplus (\text{‘}x\text{’} \mapsto p), [b], (l : s, e, [2] : \mathit{stdev} : @ : c, d, h) \longmapsto^{*} \\
\quad\quad\quad ([p, 0], e \oplus (\text{‘}m\text{’} \mapsto 0) \oplus (\text{‘}n\text{’} \mapsto 0) \oplus (\text{‘}x\text{’} \mapsto p), [@], (l : s, e, [2] : \mathit{stdev} : @ : c, d, h) \longmapsto \\
\quad\quad\quad\quad ([], e, [-1], d, h) \longmapsto^{*} \\
\quad\quad\quad\quad\quad (-1 : s', e', c', d', h)
\end{array}$$

The calculation shows that when the program program point p is applied, the current dump is discarded and the dump d is restored from p. The length m is found to be 0 so the exception handler x is applied. When this occurs, the effect is to return -1 by jumping over the pending calculation which involves applying *stdev*.

Chapter 4

Object-Oriented Features

This chapter shows how each of the object-oriented language features which are identified in chapter 2 can be implemented in the Λ-calculus. In each case, a feature is specified in terms of simple syntactic constructs (*i.e.* how the feature is expressed) and their outcomes when evaluated (*i.e.* how the feature behaves). A version of this chapter appears as [14].

An implementation for each feature is presented in terms of the Λ-calculus. The implementation represents the object-oriented data values and calculations in terms of the features available in the Λ-calculus. The implementation is shown to be correct with respect to the specification by defining conditions on the behaviour of the implementation.

The chapter is organised in a series of sections. Each section analyses a different object-oriented feature and earlier sections address simpler features than later sections. The features which are analysed are listed in the tables 2.1 and 2.2. Each section is intended to stand alone and little attempt has been made to merge different features or to present later features as an accumulation of earlier features. In this respect, the chapter could be viewed as an object-oriented language 'cook-book' where new language features can be created by merging implementations from different sections.

Many of the object-oriented language features exist in present languages. A number of the features are new and are not currently available in programming languages. It should be stressed that the Λ-calculus is not intended to be used for engineering production software and no effort has been made to analyse the efficiency of the implementations. This is defended on the basis that we are attempting to identify abstract mechanisms underlying object-oriented features and that there may be many efficient execution mechanisms which are consistent with each abstract description. The general format of each section is as follows:

Specification The object-oriented language feature is specified by inventing suitable expression syntax and showing the outcomes of performing program executions involving aspects of the feature. This allows the feature to be specified in terms of its format and its general behaviour without stating any mechanisms by which the behaviour is achieved. Note that we are using the term 'specification' in a limited context here; each specification is a particular instance of the desired feature and is at a lower level of abstraction than a description of the feature in a logic based specification language for example.

Implementation The feature is implemented in terms of the Λ-calculus. This usually involves providing a collection of operators. The expression syntax which is used in the specification of the feature can be viewed as sugar (see [27] for an analysis of the expressive powers of programming languages). When the specification is desugared, the result is a Λ-calculus expression which uses the operators. The operators use the basic features of the Λ-calculus and are viewed as characterising the object-oriented feature.

Correctness The implementation is correct with respect to the specification when its execution is consistent with respect to the specified behaviour for the feature. The correctness criteria are defined with respect to specific but characteristic instances of the programming feature.

Calculation Rules In order to show that the implementation is correct with respect to the specification, new Λ-calculus calculation rules can be defined. The rules use syntactic constructs and computational values which are specifically designed to express calculations involving the particular object-oriented feature. Each new syntactic construct and computational value can be viewed as sugar for Λ-calculus constructs; the calculation rules in terms of the new constructs are 'macro rules' which expand out to a series of applications of lower level rules.

As an example of this technique, suppose that we identify all closures of the form $<i, e, i>$ with the new computational value *id*. A new calculation rule is defined:

$$(id : v : s, e, @ : c, d, h) \longmapsto (v : s, e, c, d, h)$$

which is expanded, due to the definition of *id* and the existing calculation rules, to

$$\dfrac{\dfrac{\dfrac{e' \oplus (i \mapsto v) \bullet i = v}{([], e' \oplus (i \mapsto v), [i], (s, e, c, d), h) \longmapsto ([v], e' \oplus (i \mapsto v), [], (s, e, c, d), h)}}{(<i, e', i> : v : s, e, @ : c, d, h) \longmapsto (v : s, e, c, d, h)}}{(id : v : s, e, @ : c, d, h) \longmapsto (v : s, e, c, d, h)}$$

The main calculation rules which are necessary to show that the implementation is correct with respect to the specification is given in each section. The proof of correctness which uses the rules to show that the implementation produces the specified outcomes is not shown. In all cases the proof is constructed by mechanically instantiating the rules for the general cases of each feature, as shown for *id* above.

4.1 Feature 1: Simple Object Representation

The simplest form of object representation is a *record* or environment which associates atomic labels with values. Once it is created, such an object is a first class data value. The labels are referred to as *fields* or *attributes* of the object. The field values are referenced indirectly using the associated label. A specification for a simple object feature is as follows:

$$\textbf{let } o = \{x = 10; y = 20\} \quad S_1$$

$$\begin{array}{ll} \mathit{ref}(o, \text{`}x\text{'}) \Rightarrow 10 & S_2 \\ \mathit{ref}(o, \text{`}y\text{'}) \Rightarrow 20 & S_3 \\ \mathit{ref}(o, \text{`}z\text{'}) \Rightarrow \epsilon & S_4 \end{array}$$

Line S_1 defines an object o which has two fields named x and y with field values 10 and 20 respectively. Field names are assumed to be distinct; this restriction is lifted in later features. Lines S_2 and S_3 show the operator *ref* being used to reference the field values and line S_4 shows that the distinguished value ϵ is returned when the object does not contain a given field.

The Λ-calculus provides first class environments which may be used to implement the specified object feature. The implementation using the Λ-calculus is as follows:

$$\begin{array}{ll} \textbf{let } o = (\text{‘}x\text{’} \mapsto 10) \oplus (\text{‘}y\text{’} \mapsto 20) & I_1 \\ \\ \textbf{let } \mathit{ref} = (\bullet) & I_2 \end{array}$$

Line I_1 shows that an object is implemented as an environment. Note that the field names are not first class values in the definition S_1. This reflects usual practice in object-oriented programming languages whereas the Λ-calculus provides a great deal of flexibility by allowing names in environment construction to be arbitrary program expressions.

Field reference is implemented as the environment lookup operator $\bullet$. It is easy to see that the implementation is correct with respect to the specification when the computational value associated with o after the definition in line I_1 is:

$$o = (\text{‘}x\text{’} \mapsto 10) \oplus (\text{‘}y\text{’} \mapsto 20)$$

The ideal evaluation mechanism for object definitions such as S_1 is almost identical to the evaluation produced by the implementation. The only difference is that the specification does not allow expressions for the field names. The following evaluation rule describes object creation, an object expression is represented as $o(\tilde{i}, \tilde{e})$ where $\tilde{i}$ are the distinct field names and $\tilde{e}$ are the field value expressions:

$$\frac{\begin{array}{c}(s, e, \tilde{e}_i : c, d, h_{i-1}) \longmapsto (\tilde{v}_i : s, e, c, d, h_i) \;\; i = 1, n \\ \#(\tilde{i}) = \#(\tilde{e}) = \#(\tilde{v}) = n\end{array}}{(s, e, o(\tilde{i}, \tilde{e}) : c, d, h_0) \longmapsto (\tilde{i} \mapsto \tilde{v} : s, e, c, d, h_n)} \; R_{26}$$

4.2 Feature 2: Attribute Locations

Feature 2 requires the framework to allow parts of objects to be program locations which can be updated by side effect. The following is a simple specification for a two component object where both components refer to program locations:

$$\begin{array}{ll} \textbf{let } o = \{x = 10; y = 20\} & S_1 \\ \\ \mathit{ref}(o, \text{‘}x\text{’}) \Rightarrow 10 & S_2 \\ \mathit{ref}(o, \text{‘}y\text{’}) \Rightarrow 20 & S_3 \\ \mathit{set}(o, \text{‘}x\text{’}, 30) \Rightarrow 30 & S_4 \\ \mathit{ref}(o, \text{‘}x\text{’}) \Rightarrow 30 & S_5 \\ \mathit{ref}(o, \text{‘}y\text{’}) \Rightarrow 20 & S_6 \end{array}$$

Line S_1 shows the definition of an object o with two fields x and y. Both fields are updateable program locations and the initial contents of x is 10 and of y is 20. Lines S_2 and S_3 show that the operator *ref* is used to access the contents of the program locations which are associated with object field names. Line S_4 shows that the operator *set* is used to update the contents of a program location associated with a field name. Lines S_5 and S_6 show that the update affects the named field but no other fields of the object.

The Λ-calculus provides program locations as first class values and therefore supports a very flexible model for implementing sharing and updating mechanisms. The following is an implementation of the specification:

$$\textbf{let } o = (\text{`}x\text{'} \mapsto loc(10)) \oplus (\text{`}y\text{'} \mapsto loc(20)) \quad I_1$$

$$\textbf{let } ref(o,i) = !(o \bullet i) \quad I_2$$

$$\textbf{let } set(o,i,v) = o \bullet i := v \quad I_3$$

Line I_1 shows that object construction is implemented as an environment where field values are explicitly constructed as new program locations. Line I_2 implements the field reference operator which must look the name of the field up in the environment and then access the contents of the program location. Line I_3 implements the field update operator which must look the program location up in an object before destructively updating its contents.

In order for this implementation to be correct, the outcome of defining o must be the object o defined below and the outcomes of using the operators must agree with the specification:

$$o = (\text{`}x\text{'} \mapsto \alpha_1) \oplus (\text{`}y\text{'} \mapsto \alpha_2) \quad V_1$$

$$h_1 = h \oplus (\alpha_1 \mapsto 10) \oplus (\alpha_2 \mapsto 20) \quad V_2$$

$$h_2 = h_1 \oplus (\alpha_1 \mapsto 30) \quad V_3$$

Line V_1 shows the outcome of defining the object o in line I_1. Each field name is associated with a different machine address. Line V_2 shows the heap component of the Λ-machine after performing line I_1 where h is the heap before the definition and h_1 is the heap after. The heap h_1 associates each address with the corresponding initial field value. Line V_3 shows the heap after performing the update shown in line S_4, the contents of program location associated with field name 'x' is updated to contain 30.

In order to show that the implementation meets the specification, a number of calculation rules are defined below. The syntactic expression $o(\tilde{i},\tilde{e})$ refers to an object definition with field names $\tilde{i}$ associated with initial value expressions $\tilde{e}$:

$$\frac{\begin{array}{c}(s,e,\tilde{e}_i : c,d,h_{i-1}) \longmapsto (\tilde{v}_i : s,e,c,d,h_i) \;\; i = 1,n \\ \#(\tilde{\alpha}) = \#(\tilde{e}) = \#(\tilde{i}) = n\end{array}}{(s,e,o(\tilde{i},\tilde{e}) : c,d,h_0) \longmapsto (\tilde{i} \mapsto \tilde{\alpha} : s,e,c,d,h_n \oplus (\tilde{\alpha} \mapsto \tilde{v}))} \; R_{27}$$

Rule R_{27} describes the calculation which occurs when an object definition is performed. Each of the field value expressions $\tilde{e}$ is performed which produces each of the initial field values $\tilde{v}$. A corresponding number of location addresses $\tilde{\alpha}$ are allocated and initialized with the field values and the object produced is an environment mapping field names to machine addresses.

$$(ref : (o,i) : s,e,@ : c,d,h) \longmapsto (h \bullet (o \bullet i) : s,e,c,d,h) \;\; R_{28}$$

Rule R_{28} describes the application of the operator *ref*. The result of the application is the contents of the location bound to the field name i in object o.

$$(set : (o, i, v) : s, e, @ : c, d, h) \longmapsto (v : s, e, c, d, h \oplus ((o \bullet i) \mapsto v)) \; R_{29}$$

Rule R_{29} describes the application of the operator *set*. The result of the application is the new field value v and the field location is updated to contain v by side effect.

The program locations which are provided by the Λ-calculus can be used to support a wide variety of sharing mechanisms in objects. For example different fields of the same object can refer to the same program location or different objects can refer to the same program location through different field names.

4.3 Feature 3: Object Concatenation

Feature 3 states that the framework must support the concatenation of objects and some form of shadowing. Shadowing occurs when two objects are merged into one and when the same field name is contained in both objects. There is then a question as to which of the two possible fields is contained in the composite object. A specification for a simple version of shadowing is:

$$
\begin{array}{ll}
\textbf{let } o_1 = \{w = 10; x = 20; y = 30\} & S_1 \\[1em]
\textbf{let } o_2 = \{x = 40; y = 50; z = 60\} & S_2 \\[1em]
\textbf{let } o_3 = merge(o_1, o_2) & S_3 \\[1em]
ref(o_3, \text{`}w\text{'}) \Rightarrow 10 & S_4 \\
ref(o_3, \text{`}x\text{'}) \Rightarrow 40 & S_5 \\
ref(o_3, \text{`}y\text{'}) \Rightarrow 50 & S_6 \\
ref(o_3, \text{`}z\text{'}) \Rightarrow 60 & S_7 \\
set(o_3, \text{`}x\text{'}, 70) \Rightarrow 70 & S_8 \\
ref(o_3, \text{`}x\text{'}) \Rightarrow 70 & S_9 \\
ref(o_1, \text{`}x\text{'}) \Rightarrow 20 & S_{10} \\
ref(o_2, \text{`}x\text{'}) \Rightarrow 70 & S_{11}
\end{array}
$$

Lines S_1 and S_2 define two objects which have fields named x and y in common. Line S_3 defines an object o_3 which is the result of merging objects o_1 and o_2. We require that *merge* produces an object which contains all of the field names in its operands. All of the field values of the new object will share with the field values of the operands. Where the field names of the operands overlap a simple shadowing rule is that the composite object contains the field from the second operand. Lines S_4 – S_7 show the results of referencing fields in the merged object. Line S_8 sets the x field of the merged object and lines S_9 – S_{11} show that the field which is affected by update comes from the object o_2 and not o_1.

The Λ-calculus already provides this form of object merging and shadowing through the environment concatenation operator $\oplus$. An implementation is as follows:

$$\textbf{let } o_1 = (\text{`}w\text{'} \mapsto loc(10)) \oplus (\text{`}x\text{'} \mapsto loc(20) \oplus (\text{`}y\text{'} \mapsto loc(30)) \qquad I_1$$

$$\textbf{let } o_2 = (\text{`}x\text{'} \mapsto loc(40)) \oplus (\text{`}y\text{'} \mapsto loc(50)) \oplus (\text{`}z\text{'} \mapsto loc(60)) \qquad I_2$$

$$\textbf{let } merge = (\oplus) \qquad I_3$$

$$\textbf{let } ref(o,i) = !(o \bullet i) \qquad I_4$$

$$\textbf{let } set(o,i,v) = (o \bullet i) := v \qquad I_5$$

Lines I_1 and I_2 implement objects o_1 and o_2 as environments. Line I_3 implements object merging using environment concatenation and line I_4 implements field reference using environment lookup. Line I_5 shows that update is implemented by looking the location up in the environment and then updating the contents.

The implementation is correct with respect to the specification when the objects have the following values and the outcomes of the field references and updates are as specified:

$$o_1 = (\text{`}w\text{'} \mapsto \alpha_1) \oplus (\text{`}x\text{'} \mapsto \alpha_2) \oplus (\text{`}y\text{'} \mapsto \alpha_3) \qquad V_1$$

$$o_2 = (\text{`}x\text{'} \mapsto \alpha_4) \oplus (\text{`}y\text{'} \mapsto \alpha_5) \oplus (\text{`}z\text{'} \mapsto \alpha_6) \qquad V_2$$

$$o_3 = o_1 \oplus o_2 \qquad V_3$$

$$\begin{aligned} h_1 = \;& h \oplus (\alpha_1 \mapsto 10) \oplus (\alpha_2 \mapsto 20) \oplus (\alpha_3 \mapsto 30) \oplus \\ & (\alpha_4 \mapsto 40) \oplus (\alpha_5 \mapsto 50) \oplus (\alpha_6 \mapsto 60) \end{aligned} \qquad V_4$$

$$h_2 = h_1 \oplus (\alpha_4 \mapsto 70) \qquad V_5$$

Lines V_1 – V_3 show the outcomes of performing the object definitions. Line V_4 shows the heap after performing the definitions where h is the heap beforehand. Line V_5 shows the heap after performing the field update.

Calculation rules can be devised for object creation, field access and field update. These are as defined in §4.2.

Few object-oriented languages provide fine grain programmer control over the shadowing priority which occurs when objects are concatenated. Suppose that we require a mechanism which allows objects to be merged and where name conflicts arise the programmer must resolve the conflict:

$$
\begin{array}{ll}
\textbf{let } o_1 = \{w = 10; x = 20; y = 30\} & S_1 \\[1em]
\textbf{let } o_2 = \{x = 40; y = 50; z = 60\} & S_2 \\[1em]
\textbf{let } o_3 = \mathit{merge}(o_1, o_2, \{\text{‘}x\text{’}\}, \{\text{‘}y\text{’}\}) & S_3 \\[1em]
\mathit{ref}(o_3, \text{‘}w\text{’}) \Rightarrow 10 & S_4 \\
\mathit{ref}(o_3, \text{‘}x\text{’}) \Rightarrow 20 & S_5 \\
\mathit{ref}(o_3, \text{‘}y\text{’}) \Rightarrow 50 & S_6 \\
\mathit{ref}(o_3, \text{‘}z\text{’}) \Rightarrow 60 & S_7 \\
\mathit{set}(o_3, \text{‘}x\text{’}, 70) \Rightarrow 70 & S_8 \\
\mathit{ref}(o_3, \text{‘}x\text{’}) \Rightarrow 70 & S_9 \\
\mathit{ref}(o_1, \text{‘}x\text{’}) \Rightarrow 70 & S_{10} \\
\mathit{ref}(o_2, \text{‘}x\text{’}) \Rightarrow 40 & S_{11}
\end{array}
$$

The new specification extends the *merge* operator with two arguments which are sets of fields names. Given a merge: $\mathit{merge}(o_1, o_2, P_1, P_2)$ both P_1 and P_2 are sets of names which are fields of both o_1 and o_2. The operator allows the programmer to define whether the fields values in the result are taken from o_1 or o_2 when the field names occur in both. When the name occurs in P_1 the field value will be taken from o_1 and when it occurs in P_2 then the value will be taken from o_2. The sets P_1 and P_2 should have no names in common.

Lines $S_1 - S_2$ define two objects o_1 and o_2. Line S_3 merges these objects to produce o_3 where field x is to be taken from o_1 and field y is to be taken from o_2. Lines $S_4 - S_{11}$ show that the fields have been merged as required.

This can be implemented in the Λ-calculus since operators are provided for transforming environments. Such an operator is defined in the Λ-calculus as follows:

$$
\begin{array}{l}
\textbf{let } \mathit{merge}(o_1, o_2, P_1, P_2) = \\
\quad \begin{array}{ll}
\textbf{let} & o_3 = o_1 \backslash ((\mathit{dom}(o_1)) - (\mathit{dom}(o_2))) \\
& o_4 = o_2 \backslash ((\mathit{dom}(o_2)) - (\mathit{dom}(o_1))) \\
& o_5 = o_1 \backslash P_1 \\
& o_6 = o_2 \backslash P_2 \\
\textbf{in} & o_3 \oplus o_4 \oplus o_5 \oplus o_6
\end{array}
\end{array}
$$

The implementation is correct when the following outcomes are produced:

$$
\begin{array}{ll}
o_1 = (\text{‘}w\text{’} \mapsto \alpha_1) \oplus (\text{‘}x\text{’} \mapsto \alpha_2) \oplus (\text{‘}y\text{’} \mapsto \alpha_3) & V_1 \\[1em]
o_2 = (\text{‘}x\text{’} \mapsto \alpha_4) \oplus (\text{‘}y\text{’} \mapsto \alpha_5) \oplus (\text{‘}z\text{’} \mapsto \alpha_6) & V_2 \\[1em]
o_3 = (\text{‘}w\text{’} \mapsto \alpha_1) \oplus (\text{‘}x\text{’} \mapsto \alpha_2) \oplus (\text{‘}y\text{’} \mapsto \alpha_5) \oplus (\text{‘}z\text{’} \mapsto \alpha_6) & V_3 \\[1em]
h_1 = h \oplus (\alpha_1 \mapsto 10) \oplus (\alpha_2 \mapsto 20) \oplus (\alpha_3 \mapsto 30) \oplus & V_4 \\
\qquad (\alpha_4 \mapsto 40) \oplus (\alpha_5 \mapsto 50) \oplus (\alpha_6 \mapsto 60) & \\[1em]
h_2 = h_1 \oplus (\alpha_2 \mapsto 70) & V_5
\end{array}
$$

The calculation rule for *merge* is as follows:

$$\frac{\begin{array}{c} o_5 = o_1 \backslash P_1 \\ o_6 = o_2 \backslash P_2 \\ o_3 = o_1 \backslash ((dom(o_1)) - (dom(o_2))) \\ o_4 = o_2 \backslash ((dom(o_2)) - (dom(o_1))) \end{array}}{(merge : (o_1, o_2, P_1, P_2) : s, e, @ : c, d, h) \longmapsto (o_3 \oplus o_4 \oplus o_5 \oplus o_6 : s, e, c, d, h)} \; R_{30}$$

The calculation rules for *ref* and *set* are R_{28} and R_{29}.

4.4 Feature 4: Unqualified References

Feature 4 requires the framework to support unqualified references in objects. Such a feature allows objects to contain functions as field values and allows all the field names to be scoped over the bodies of each function. When a function is a field value it will be referred to as a *method* of the object which contains it. Other field values are distinguished from methods by referring to them as *attributes* of the object which contains them. A specification for such a feature is as follows:

$$\textbf{let } p(f, v_1, v_2) = \textbf{if } v_1 = v_2 \textbf{ then } 0 \textbf{ else } 1 + (f(v_1 - 1)) \quad S_1$$

$$\textbf{let } o = \{i = 10; j(v) = p(k, v, !i); k(v) = p(j, v, !i)\} \quad S_2$$

$$\begin{array}{ll} send(o, `j', 20) \Rightarrow 10 & S_3 \\ send(o, `k', 20) \Rightarrow 10 & S_4 \\ set(o, `i', 20) \Rightarrow 20 & S_5 \\ send(o, `j', 30) \Rightarrow 10 & S_6 \\ send(o, `k', 30) \Rightarrow 10 & S_7 \end{array}$$

Line S_1 defines an operator p which is subsequently used to count the number of function calls which it takes to decrement v_1 until it is equal to v_2.

Line S_2 defines an object o with three fields named i, j and k. Attribute field i is a program location whose initial contents is 10. The two methods j and k are mutually recursive and both reference the attribute i. When either of the methods are invoked, the operator p is passed the other method and the current value of the argument v and the attribute i.

The program is intended to illustrate that field names may be referenced without further qualification in an object's methods. Outside of an object, methods are invoked by sending the object a *message*. A message contains three components: an object which is the *target* of the message, a method name and a value. A message is delivered to its target by invoking the named method. The method is a function which is applied to the message value. Message passing is performed by the operator *send*. The arguments of *send* are (o, i, v) where o is an object, i is a method field name and v is a value which will be supplied as an argument to the method.

Lines S_3 and S_4 show that the mutually recursive methods behave as required. Line S_5 updates the value of the attribute i and lines S_6 and S_7 show that the methods always access the current value of attributes.

The Λ-calculus achieves mutually recursive bindings using the built-in operator Y. The mutually recursive field bindings are established using a **letrec**-expression and then the environment which has been created is reified and returned as the new object. The implementation is as follows:

$$
\begin{array}{ll}
\textbf{let } o = & I_1 \\
\quad \textbf{letrec } x() = & \\
\quad\quad \textbf{letrec } i = loc(10) & \\
\quad\quad\quad\quad j(v) = p(k, v, !i) & \\
\quad\quad\quad\quad k(v) = p(j, v, !i) & \\
\quad\quad \textbf{in } R(\lambda().()) - R(x) & \\
\quad \textbf{in } x() & \\
& \\
\textbf{let } ref(o, i) = !(o \bullet i) & I_2 \\
& \\
\textbf{let } set(o, i, v) = (o \bullet i) := v & I_3 \\
& \\
\textbf{let } send(o, i, v) = o \bullet i(v) & I_4
\end{array}
$$

Line I_1 defines the object o which is an environment constructed using a **letrec**-expression. The outer **letrec**-expression is used to allow the current environment at that point to be subtracted from the environment containing the field definitions. The identifier x is introduced in order to refer to the closure environment.

The inner **letrec**-expression is used to establish the mutually recursive field definitions. Lines I_2 and I_3 define the field access and update operators. Line I_4 defines the message passing operator.

The implementation is correct with respect to the specification when the object definition for o produces the following outcome:

$$o = (\text{`}i\text{'} \mapsto \alpha) \oplus (\text{`}j\text{'} \mapsto <\text{`}v\text{'}, e \oplus o, p(k, v, !i)>) \oplus (\text{`}k\text{'} \mapsto <\text{`}v\text{'}, e \oplus o, p(j, v, !i)>) \quad V_1$$

$$h_1 = h \oplus (\alpha \mapsto 10) \quad V_2$$

$$h_2 = h_1 \oplus (\alpha \mapsto 20) \quad V_3$$

Line V_1 shows that the object o contains two closures whose environments are $e \oplus o$ where e was the environment which was current when the object definition was performed. Line V_2 shows the heap h_1 which is current after the object definition, h is the heap before the object definition. Line V_3 is the heap after the attribute is set.

Calculation rules can be defined to support unqualified references. These are shown below and use the following representation $o(\tilde{i}, \tilde{e}, \tilde{j}, \tilde{k}, \tilde{b})$ to denote an object definition expression where $\tilde{i}$ are the attribute field names, $\tilde{e}$ are the attribute field expressions, $\tilde{j}$ are the method field names, $\tilde{k}$ are the method parameter names and $\tilde{b}$ are the method bodies. For example the following is the object definition shown in lines S_2/I_1:

$$o(\text{`}i\text{'}, 10, (\text{`}j\text{'}, \text{`}k\text{'}), (\text{`}v\text{'}, \text{`}v\text{'}), (p(k, v, !i), p(j, v, !i)))$$

The calculation rules for object definition and message passing are as follows:

$$\frac{\begin{array}{c}(s, e, \tilde{e}_i : c, d, h_{i-1}) \longmapsto (\tilde{v}_i : s, e, c, d, h_i) \\ \#(\tilde{i}) = \#(\tilde{e}) = \#(\tilde{v}) = \#(\tilde{\alpha}) = n \\ x = (\tilde{i} \mapsto \tilde{\alpha}) \oplus (\tilde{j} \mapsto <\tilde{k}, e \oplus x, \tilde{b}>)\end{array}}{(s, e, o(\tilde{i}, \tilde{e}, \tilde{j}, \tilde{k}, \tilde{b}) : c, d, h_0) \longmapsto (x : s, e, c, d, h_n \oplus (\tilde{\alpha} \mapsto \tilde{v}))} \; R_{31}$$

Rule R_{31} describes the calculation which occurs when an object definition is performed. Each of the attribute expressions are performed and fresh program locations are allocated for them. Each of the methods are transformed into closures which contain the object x as part of their closure environment.

$$\frac{((o \bullet i) : v : s, e, @ : c, d, h) \longmapsto (v : s, e, c, d, h')}{(send : (o, i, v) : s, e, @ : c, d, h) \longmapsto (v : s, e, c, d, h')} \; R_{32}$$

Rule R_{32} shows that the simple message passing mechanism can be included in the Λ-calculus evaluation mechanism. The calculation rules for *ref* and *set* are R_{28} and R_{29}.

In general, many object-oriented features rely upon the message passing mechanism performing some interpretive tasks such as placing identifiers in scope or removing identifiers from scope. Rule R_{32} is the basis for building many different variations on the theme of message passing into the Λ-calculus.

4.5 Feature 5: Private Names

Feature 5 states that a framework must allow simple objects to hide part of their interfaces so that they are not available via outside reference. For example, if an object defines names i, j and k where i is a program location, j is a function which returns the current value of the program location and k allows the program location to be updated then the object may wish to hide the location i away so that other objects cannot have explicit reference to its internal representation. A specification for this is:

$$\textbf{let } o = \{i =_p 10; j() = !i; k(v) = i := v\} \quad S_1$$

$$\begin{array}{ll} \mathit{ref}(o, \text{`}i\text{'}) \Rightarrow \epsilon & S_2 \\ \mathit{send}(o, \text{`}j\text{'}, ()) \Rightarrow 10 & S_3 \\ \mathit{send}(o, \text{`}k\text{'}, 20) \Rightarrow 20 & S_4 \\ \mathit{send}(o, \text{`}j\text{'}, ()) \Rightarrow 20 & S_5 \end{array}$$

Line S_1 defines an object o with a private attribute i and two methods j and k. The private attribute is distinguished by the use of $=_p$ instead of $=$ in the definition. Private field names cannot be accessed using the external object operators *ref* and *send* although the private names can be accessed without qualification within the body of all methods.

Line S_2 shows that the private attribute i is not accessible outside the object. Lines S_3 – S_5 show that the methods can access and update private names.

The Λ-calculus can achieve the desired behaviour in more than one way. If the private part of the interface does not need to reference any of the public part then it may be defined in a surrounding scope, for example:

$$
\begin{array}{l}
\textbf{let } o = \\
\quad \textbf{let } i = loc(10) \\
\quad \textbf{in letrec } x() = \\
\qquad \textbf{letrec } \; j() = !i \\
\qquad\qquad\quad\;\; k(v) = i := v \\
\qquad \textbf{in } R\lambda().() - R(x) \\
\quad\; \textbf{in } x()
\end{array}
$$

where the binding for the identifier i is created outside the boundary of the closure which is bound to x. The definitions for the operators *ref*, *set* and *send* are given in §4.4. The implementation is correct with respect to the specification when the outcome of the object definition is o:

$$o = (\text{`}j\text{'} \mapsto <(), e \oplus o', !i>) \oplus (\text{`}k\text{'} \mapsto <v, e \oplus o', i := v>) \quad V_1$$

$$o' = (\text{`}i\text{'} \mapsto \alpha) \oplus o \quad V_2$$

$$h_1 = h \oplus (\alpha \mapsto 10) \quad V_3$$

$$h_2 = h_1 \oplus (\alpha \mapsto 20) \quad V_4$$

Line V_1 shows that the object o contains bindings for the fields named j and k only. Each of these fields are methods which have closure environments containing the environment o'. Line V_2 shows that o' contains the attribute i in addition to the methods j and k. Line V_3 shows the heap after the object definition and line V_4 shows the heap after the attribute i is updated by method k.

Object creation can be described by a calculation rule. An object definition expression is denoted by $o(\tilde{i}, \tilde{e}, \tilde{j}, \tilde{k}, \tilde{b}, S)$ where $\tilde{i}$ are attribute field names, $\tilde{e}$ are attribute expressions, $\tilde{j}$ are method field names, $\tilde{k}$ are method parameter names, $\tilde{b}$ are method bodies and S is a set of private field names.

$$\frac{\begin{array}{c}(s, e, \tilde{e}_i : c, d, h_{i-1}) \longmapsto (\tilde{v}_i : s, e, c, d, h_i) \; i = i, n \\ \#(\tilde{i}) = \#(\tilde{e}) = \#(\tilde{v}) = \#(\tilde{\alpha}) = n \\ x = (\tilde{i} \mapsto \tilde{\alpha}) \oplus (\tilde{j} \mapsto <\tilde{k}, e \oplus x, \tilde{b}>)\end{array}}{(s, e, o(\tilde{i}, \tilde{e}, \tilde{j}, \tilde{k}, \tilde{b}, S) : c, d, h_0) \longmapsto ((x/S) : s, e, c, d, h_n \oplus (\tilde{\alpha} \mapsto \tilde{v}))} \; R_{33}$$

In the implementation, the private names are bound in the environment which is eventually reified and subtracted from the environment containing the public names. This is expressed in the calculation rule R_{33} by removing the names in the set S from the environment x when it is returned as the outcome of the object definition.

The Λ-calculus provides built-in environment restriction operators. This provides the programmer with fine grain control over the names which occur in environments. An alternative way of achieving a private interface in an object is to explicitly remove the names from the object after it is created.

Suppose that the object o is defined again to contain the names i, j and k. Now suppose that all three are methods, that all three wish to refer to all the names in the interface of the object and that the name i is to be private to the object. The mechanism for private interfaces described above will not work because i would not be able to refer to either j or k. The following definition achieves the desired result:

$$\begin{array}{l}
\textbf{let } o = \\
\quad \textbf{letrec } x() = \\
\quad\quad \textbf{letrec } \begin{array}[t]{l} i = \ldots \\ j = \ldots \\ k = \ldots \end{array} \\
\quad\quad \textbf{in } (R\lambda().() - R(x))\backslash\{\text{`}j\text{'}, \text{`}k\text{'}\} \\
\quad \textbf{in } x()
\end{array}$$

The calculation rule for this expression is R_{33}.

4.6 Feature 6: Self Reference in Simple Objects

Feature 6 states that the framework must allow objects to reference themselves. Self reference can take a number of different forms, even with respect to simple objects. Firstly self reference may be open or closed. Open self reference means extensions to an object are taken into account by the self reference mechanism. Closed self reference means that self reference is fixed when an object is created and further extensions are not taken into account. A variant of open and closed self reference is *sealable* where the self reference is open until an object is sealed at which point the self reference is closed. When providing self reference, a language may dictate which name is to be used or allow the programmer to supply it. The Λ-calculus will be shown to support all of these features.

Suppose that a language feature is to be designed which provides self reference using the identifier *self*. The method invocation mechanism is to automatically provide a binding for *self* whenever a method is referenced in an object. A specification is as follows:

$$\begin{array}{ll}
\textbf{let } o_1 = \{i() = \mathit{self}\} & S_1 \\[1em]
\textbf{let } o_2 = \{j() = \mathit{self}\} & S_2 \\[1em]
\mathit{send}(o_1, \text{`}i\text{'}, ()) \Rightarrow o_1 & S_3 \\
\mathit{send}(o_2, \text{`}j\text{'}, ()) \Rightarrow o_2 & S_4 \\
\mathit{send}(\mathit{merge}(o_1, o_2), \text{`}i\text{'}, ()) \Rightarrow \mathit{merge}(o_1, o_2) & S_5 \\
\mathit{send}(\mathit{merge}(o_1, o_2), \text{`}j\text{'}, ()) \Rightarrow \mathit{merge}(o_1, o_2) & S_6
\end{array}$$

Lines S_1 and S_2 define two objects o_1 and o_2 which contain two identical methods called i and j which each return the value of the identifier *self*. A method is invoked by sending an object (the receiver) a message. During the activation of the body of the method the identifier *self* refers to the receiver of the message.

Lines S_3 and S_4 show that the value of *self* refers to o_1 and o_2 when they are sent messages i and j respectively. When the objects are merged in lines S_5 and S_6 the values of *self* in each method activation is updated accordingly.

One way of implementing open self reference in the Λ-calculus is to build the provision of the identifier *self* into the message delivery operator *send*. The implementation is as follows:

$$\textbf{let } o_1 = \text{`}i\text{'} \mapsto \lambda().self \qquad I_1$$

$$\textbf{let } o_2 = \text{`}j\text{'} \mapsto \lambda().self \qquad I_2$$

$$\textbf{let } send(o, i, v) = I(R(o \bullet i) \oplus (\text{`}self\text{'} \mapsto o), o \bullet i)(v) \qquad I_3$$

$$\textbf{let } merge = (\oplus) \qquad I_4$$

Lines I_1 and I_2 define the objects o_1 and o_2 as environments. The methods are λ-functions in which the identifier *self* is freely referenced. Line I_3 defines the operator *send* which extends the closure environment of the method with a binding for *self* using the operators I and R. This is made possible by the environment reification mechanism of the Λ-calculus. The implementation is correct when the objects o_1 and o_2 are defined as follows:

$$o_1 = \text{`}i\text{'} \mapsto <(), e, self>$$

$$o_2 = \text{`}j\text{'} \mapsto <(), e, self>$$

and passing the given messages have the specified outcomes using the following calculation rule:

$$\frac{\begin{array}{c} o \bullet i = <i_2, e_2, b> \\ (<i_2, e_2 \oplus (\text{`}self\text{'} \mapsto o), b> : v_1 : s, e_1, @ : c, d, h_1) \longmapsto (v_2 : s, e_1, c, d, h_2) \end{array}}{(send : (o, i_1, v_1) : s, e_1, @ : c, d, h_1) \longmapsto (v_2 : s, e_1, c, d, h_2)} \; R_{34}$$

Rule R_{34} shows the application of a message passing operator for open self reference. Self reference is achieved by the extension of the closure environment which implements the method named i_1 in object o. The operator *send* defines that the identifier used for self reference is *self*. By implementing the self reference feature using first class environments, the identifier which is used for self reference can be universally changed by using a different message delivery operator.

Rather than prescribe the identifier which is used for self reference, some languages let the programmer specify which identifier is to be used. In the following specifications, self reference is open, but the first component of an object declaration is the identifier to be used for self reference:

$$\textbf{let } o_1 = \{k; i() = k\} \qquad S_1$$

$$\textbf{let } o_2 = \{l; j() = l\} \qquad S_2$$

$$\begin{array}{ll} send(o_1, \text{`}i\text{'}, ()) \Rightarrow o_1 & S_3 \\ send(o_2, \text{`}j\text{'}, ()) \Rightarrow o_2 & S_4 \\ send(merge(o_1, o_2), \text{`}i\text{'}, ()) \Rightarrow merge(o_1, o_2) & S_5 \\ send(merge(o_1, o_2), \text{`}j\text{'}, ()) \Rightarrow merge(o_1, o_2) & S_6 \end{array}$$

Line S_1 defines an object o_1 which uses the identifier k for self reference. The scope of the identifier for self reference is the all of the methods defined by an object. Line S_2 defines an object o_2 which uses identifier l for self reference. Lines S_3 – S_6 show that the object definitions behave as expected.

The Λ-calculus implementation of this feature places the responsibility of naming the object which is used for self reference on the individual methods:

$$\begin{array}{ll} \textbf{let } o_1 = \text{`}i\text{'} \mapsto \lambda k.\lambda().k & I_1 \\[1em] \textbf{let } o_2 = \text{`}j\text{'} \mapsto \lambda l.\lambda().l & I_2 \\[1em] \textbf{let } \mathit{send}(o,i,v) = o \bullet i(o)(v) & I_3 \\[1em] \textbf{let } \mathit{merge} = (\oplus) & I_4 \end{array}$$

Lines I_1 and I_2 show that the identifier for self reference in each object definition is added as an extra parameter to the method definitions. The value of the identifier must be supplied when the message is delivered by the operator *send* which is implemented in line I_3. The operator *send* looks the method up in the object and then supplies the receiver of the message as the first argument to the method. Object merging is not complicated by this in line I_4.

A method can be represented as a data value $<i,j,e,b>$ where i is an identifier used for self reference, j is the method argument given in the definition, e is an environment and b is the method body. An object definition expression is represented as $o(i,\tilde{i},\tilde{e},\tilde{j},\tilde{k},\tilde{b})$ where i is an identifier used for self reference, $\tilde{i}$ are attribute names, $\tilde{e}$ are attribute expressions, $\tilde{j}$ are method names, $\tilde{k}$ are method arguments and $\tilde{b}$ are method bodies. The implementation is correct with respect to the specification when the outcomes of the object definitions are:

$$o_1 = \text{`}i\text{'} \mapsto <\text{`}k\text{'}, (), e, k>$$

$$o_2 = \text{`}j\text{'} \mapsto <\text{`}l\text{'}, (), e, l>$$

and the outcomes of the message passing expressions are as given in the specification. There are two calculation rules associated with this representation: object definition and message passing. These are defined below:

$$\frac{\begin{array}{c}(s,e,\tilde{e}_i : c,d,h_{i-1}) \longmapsto (\tilde{v}_i : s,e,c,d,h_i) \; i = 1,n \\ \#(\tilde{i}) = \#(\tilde{e}) = \#(\tilde{v}) = \#(\tilde{\alpha}) = n\end{array}}{(s,e,o(i,\tilde{i},\tilde{e},\tilde{j},\tilde{k},\tilde{b}) : s,d,h_0) \longmapsto ((\tilde{i} \mapsto \tilde{\alpha}) \oplus (\tilde{j} \mapsto <i,\tilde{k},e,\tilde{b}>) : s,e,c,d,h_n \oplus (\tilde{\alpha} \mapsto \tilde{v})} \; R_{35}$$

Rule R_{35} shows that the creation of an object allocates the program locations for the attributes and initialises them and translates the method definitions into method values as defined above. Message passing is performed by:

$$\frac{\begin{array}{c}(<i_3, e_2 \oplus (i_2 \mapsto o), b> : v_1 : s, e_1, @ : c, d, h_1) \longmapsto (v_2 : s, e_1, c, d, h_2) \\ o \bullet i_1 = <i_2, i_3, e_2, b>\end{array}}{(\mathit{send} : (o, i_1, v_1) : s, e_1, @ : c, d, h_1) \longmapsto (v_2 : s, e_1, c, d, h_2)} \; R_{36}$$

Rule R_{36} translates a method into a closure by extending the environment e_2 with a binding between the identifier for self reference i_2 and the receiver of the message o.

Closed self reference fixes the object when it is created. Extending the object has no effect on the value if the identifier which is used for self reference. Again, the identifier which is used for self reference may be supplied by the programming language or the programmer.

Here is a specification of closed self reference:

$$\textbf{let } o_1 = \{k; i() = k\} \qquad S_1$$

$$\textbf{let } o_2 = \{l; j() = j\} \qquad S_2$$

$$\begin{array}{ll} send(o_1, \text{`}i\text{'}, ()) \Rightarrow o_1 & S_3 \\ send(o_2, \text{`}j\text{'}, ()) \Rightarrow o_2 & S_4 \\ send(merge(o_1, o_2), \text{`}i\text{'}, ()) \Rightarrow o_1 & S_5 \\ send(merge(o_1, o_2), \text{`}j\text{'}, ()) \Rightarrow o_2 & S_6 \end{array}$$

Lines S_1 and S_2 define two objects o_1 and o_2 each each of which explicitly designates an identifier which will be used for self reference, k and l respectively, and defines a method which refers to this identifier, i and j respectively. Closed self reference requires that the referents of the identifiers k and l are fixed when the objects are defined.

Lines S_3 and S_4 show that the identifiers k and l refer to the required objects when the methods i and j are activated. Lines S_5 and S_6 show that, unlike open self reference, extending the objects has no effect: the identifiers k and l refer to o_1 and o_2 even if these objects are not the receivers of the message.

Closing self reference when an object is created can be viewed as creating a cyclic data structure where the object refers to itself through the identifier which is used for self reference. The Λ-calculus uses recursive definitions to create cyclic data values and closed self reference can be implemented as follows:

$$\textbf{let } o_1 = Y\lambda k.\text{`}i\text{'} \mapsto \lambda().k \qquad I_1$$

$$\textbf{let } o_2 = Y\lambda l.\text{`}j\text{'} \mapsto \lambda().l \qquad I_2$$

$$\textbf{let } send(o, i, v) = o \bullet i(v) \qquad I_3$$

$$\textbf{let } merge = (\oplus) \qquad I_4$$

Lines I_1 and I_2 show that the objects o_1 and o_2 are constructed using the built-in operator Y to fix the values of k and l. The operator Y constructs cyclic values and k and l will have o_1 and o_2 as values respectively.

The implementation is correct with respect to the specification when the definitions of the objects o_1 and o_2 produce the following computational values:

$$o_1 = \text{`}i\text{'} \mapsto <(), e \oplus (\text{`}k\text{'} \mapsto o_1), k> \qquad V_1$$

$$o_2 = \text{`}j\text{'} \mapsto <(), e \oplus (\text{`}l\text{'} \mapsto o_2), l> \qquad V_2$$

Line V_1 shows the result of defining the object o_1 which contains a closure whose environment refers to o_1 through k. Line V_2 shows a similar outcome for o_2 except that the closure refers to o_2 through l. When these environments are merged using $\oplus$, the values of the identifiers k and l will not be affected.

Two calculation rules can be defined to show object creation and message passing. The rule for message passing is R_{32} and the rule for closed self reference object creation is as follows:

$$\frac{\begin{array}{c}(s, e \oplus (i \mapsto x), \tilde{e}_i : c, d, h_{i-1}) \longmapsto (\tilde{v}_i : s, e \oplus (i \mapsto x), c, d, h_i)\ i = 1, n \\ \#(\tilde{i}) = \#(\tilde{e}) = \#(\tilde{v}) = \#(\tilde{\alpha}) = n \\ x = (\tilde{i} \mapsto \tilde{\alpha}) \oplus (\tilde{j} \mapsto <\tilde{j}, e \oplus (i \mapsto x), \tilde{b}>)\end{array}}{(s, e, o(i, \tilde{i}, \tilde{e}, \tilde{j}, \tilde{k}, \tilde{b}) : c, d, h_0) \longmapsto (x : s, e, c, d, h_n \oplus (\tilde{\alpha} \mapsto \tilde{v}))}\ R_{37}$$

Rule R_{37} shows that attribute expressions are performed and method closures are created with respect to an environment which contains a binding between the identifier for self reference i and the object which is eventually produced x.

Some object-oriented languages provide closed self reference, for example Smalltalk and Eiffel, while others provide closed self reference, for example Self. Suppose that a new language feature is to be designed in which self reference is initially open and can be made closed by the application of an operator:

$$\textbf{let}\ o_1 = \{k; i() = k\} \qquad S_1$$

$$\textbf{let}\ o_2 = \{l; j() = l\} \qquad S_2$$

$$\begin{array}{ll}
\mathit{send}(o_1, \text{'}i\text{'}, ()) \Rightarrow o_1 & S_3 \\
\mathit{send}(o_2, \text{'}j\text{'}, ()) \Rightarrow o_2 & S_4 \\
\mathit{send}(\mathit{merge}(o_1, o_2), \text{'}i\text{'}, ()) \Rightarrow \mathit{merge}(o_1, o_2) & S_5 \\
\mathit{send}(\mathit{merge}(o_1, o_2), \text{'}j\text{'}, ()) \Rightarrow \mathit{merge}(o_1, o_2) & S_6 \\
\mathit{send}(\mathit{merge}(\mathit{seal}(o_1), o_2), \text{'}i\text{'}, ()) \Rightarrow \mathit{seal}(o_1) & S_7 \\
\mathit{send}(\mathit{merge}(o_1, \mathit{seal}(o_2)), \text{'}j\text{'}, ()) \Rightarrow \mathit{seal}(o_2) & S_8
\end{array}$$

Lines S_1 and S_2 define two objects as before. In this case self reference (via identifiers k and l) is open until it is closed using the operator *seal*. The argument to seal is an object o; all the currently open self referential identifiers in o are fixed by *seal* so that they refer to o.

Lines S_3 – S_6 show that open self reference is supported by this mechanism. Lines S_7 and S_8 show that once an object is sealed, the identifiers for self reference are not affected by further extension, *i.e.* the mechanism supports closed self reference.

The Λ-calculus provides fine grain control over the message passing mechanism by implementing objects as environments. Also, the Λ-calculus is a higher order language which supports functions as data values. Both of these properties are exploited in the following implementation of sealable self reference:

$$\textbf{let}\ u(f, o) = f(o) \qquad I_1$$

$$\textbf{let}\ s(o)(f, \_) = f(o) \qquad I_2$$

$$\textbf{let}\ o_1 = \text{'}i\text{'} \mapsto (u, \lambda k.\lambda().k) \qquad I_3$$

$$\textbf{let}\ o_2 = \text{'}j\text{'} \mapsto (u, \lambda l.\lambda().l) \qquad I_4$$

$$\textbf{let}\ \mathit{send}(o, i, v) = \mathit{1st}(o \bullet i)(\mathit{2nd}(o \bullet i), o)(v) \qquad I_5$$

$$\textbf{let}\ \mathit{merge} = (\oplus) \qquad I_6$$

$$\textbf{let}\ \mathit{seal}(o_1) = Y \lambda o_2.eh(\oplus)(\lambda(i, (\_, m)).i \mapsto (s(o_2), m))(\{\})(o_1) \qquad I_7$$

The implementation uses pairs to represent methods (lines I_3 and I_4). The first component of a method pair is a function, either u when the method is unsealed or $s(o)$ when the method is sealed and uses the object o for self reference. The second component of the method pair is a function which implements the method.

A message is sent to an object by $send(o, i, v)$ (line I_5). The receiver of the message o binds i to a method pair (f, m) and the message passing mechanism applies the first component f to a pair containing the second component m and the receiver of the message: $f(m, o)$. In doing so, the function f can choose whether to pass o to m as the object for self reference or whether to discard o entirely; it will do the former when f is u (line I_1) *i.e.* when the method is *unsealed* and it will do the latter when f is $s(o')$ (line I_2) *i.e.* when the method has been *sealed* with respect to o' for self reference.

Sealing an object is implemented in line I_7 by simply changing all the u components to $s(o')$ for some appropriate value o'. The transformation is performed using an environment homomorphism which is constructed using the operator *eh*. The object o' is constructed by taking a fixed point, since the object which is the result of a self reference in a sealed object must also be sealed (lines S_7 and S_8).

There are two possible states for a method: unsealed and sealed. An unsealed method can be represented as $<i, j, e, b>$ where i is the identifier for self reference, j is the method argument, e is an environment and b is the method body. When an object definition is performed, all of the new methods which are created will be of this form. A sealed method can be represented as an ordinary closure $<i, e, b>$ where e contains a binding between the identifier for self reference and an object. Rule R_{35} describes the calculation which occurs when an object definition with sealable self reference is performed. There are two possible calculations which occur when messages are passed using the operator *send*:

$$\frac{o \bullet i_1 = <i_2, i_3, e_2, b>}{(send : (o, i_1, v) : s, e_1, @; c, d, h) \longmapsto (<i_3, e_2 \oplus (i_2 \mapsto o), b> : v : s, e_1, @ : c, d, h)} \; R_{38}$$

Rule R_{38} describes the case where the method which is invoked is open.

$$\frac{o \bullet i = <\_, \_, \_>}{(send : (o, i, v) : s, e, @ : c, d, h) \longmapsto (o \bullet i : v : s, e, @ : c, d, h)} \; R_{39}$$

Rule R_{39} describes the case where the method which is invoked is closed. The following rule describes the action which *seal* has on an object:

$$\frac{o = e_1 \oplus (\tilde{i} \mapsto <\tilde{k}, \tilde{e} \oplus (\tilde{j} \mapsto o), \tilde{b}>) \oplus e_2}{(seal : e_1 \oplus (\tilde{i} \mapsto <\tilde{j}, \tilde{k}, \tilde{e}, \tilde{b}>) \oplus e_2 : s, e, @ : c, d, h) \longmapsto (o : s, e, c, d, h)} \; R_{40}$$

Rule R_{40} shows what happens when the currently open methods in an object are sealed. The object which is to be sealed is represented as $e_1 \oplus \ldots \oplus e_2$ where e_1 contains all the attributes and e_2 contains all the currently sealed methods. The expression $\tilde{i} \mapsto <\tilde{j}, \tilde{k}, \tilde{e}, \tilde{b}>$ represents a collection of currently unsealed methods. They are sealed by transforming each unsealed method into a closure and extending the closure environment with a binding for the self reference identifier and the object o.

4.7 Feature 7: Point of View Self Reference

Feature 7 states that the framework must support the *point of view* notion for self reference. This occurs in the language Self as the sender path tiebreaker rule. The idea is that objects are constructed from modular units into structures. For example, the structures may be tree shaped or graph shaped. The value of the identifier *self* (or whatever is being used for self reference) within an object depends upon where in the object the reference occurs and upon the shape of the overall object.

In the Λ-calculus, environments are tree structured. For example, the following diagram shows an object o which is constructed from three sub-objects o_1, o_2 and o_3:

$$\begin{tikzcd} & \oplus \arrow[dl] \arrow[dr] & & \\ o_1 & & \oplus \arrow[dl] \arrow[dr] & \\ & o_2 & & o_3 \end{tikzcd}$$

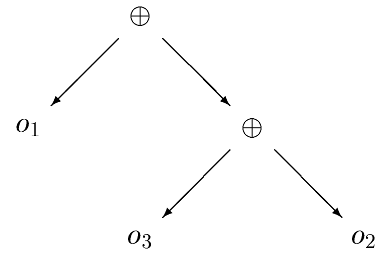
Each of the sub-objects contain methods, which, when activated, may reference the identifier *self*. In most object-oriented languages, the value of *self* will be the same at each point in o, for example no matter whether *self* is referenced in a method defined in o_1, o_2 or o_3, the value of *self* will be o. This causes no problems if there is no overlap between method names defined in o_1, o_2 and o_3. As described in chapter 2, Self identifies a potential source of confusion when method names do overlap, because extending an object can radically change the behaviour of an existing method.

The source of the problem is that shadowing in objects is usually quite inflexible. For example, once a method is shadowed, it cannot be referenced anywhere within the object using self reference. Self proposes that the shadowing rules are made more sophisticated so that the order in which names are shadowed depends upon where you are in the object. Shadowing which occurs due to a recent extension, will have less priority than shadowing which occurs due to an earlier extension.

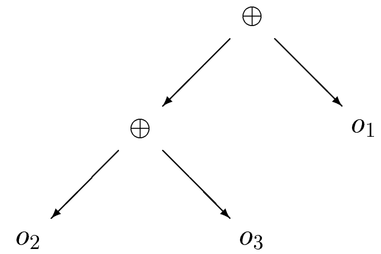
In order to see how this works, consider the value of *self* at different points in the object which is shown above. The rules for determining the value of *self* mean that the point of reference indicates the order in which any shadowing should take precedence. For example, a reference at o_3 will allow methods defined by o_3 to take precedence, whereas, a reference at o_1 will allow methods defined by o_1 to take precedence. The following object is the value of *self* at o_3:

$$\begin{tikzcd} & \oplus \arrow[dl] \arrow[dr] & & \\ o_1 & & \oplus \arrow[dl] \arrow[dr] & \\ & o_2 & & o_3 \end{tikzcd}$$

Since $\oplus$ is defined to shadow on the right, the definitions of o_3 take precedence. Where there is a choice between the other components of the object, we rely upon the original ordering given in the definition of o. The following object is the value of *self* at o_2:



The ordering of the sub-objects o_2 and o_3 has been reversed to allow the methods defined by o_2 to take precedence over those defined by o_3. Finally, the value of *self* at o_1 is:



The following is a specification of the point of view feature:

$$\begin{array}{ll}
\textbf{let } o_1 = \{i() = \mathit{self}\} & S_1 \\[1em]
\textbf{let } o_2 = \{j() = \mathit{self}\} & S_2 \\[1em]
\textbf{let } o_3 = \{k() = \mathit{self}\} & S_3 \\[1em]
\textbf{let } o = \mathit{merge}(o_1, \mathit{merge}(o_2, o_3)) & S_4 \\[1em]
\mathit{send}(o, `i', ()) \Rightarrow \mathit{merge}(\mathit{merge}(o_2, o_3), o_1) & S_5 \\
\mathit{send}(o, `j', ()) \Rightarrow \mathit{merge}(o_1, \mathit{merge}(o_3, o_2)) & S_6 \\
\mathit{send}(o, `k', ()) \Rightarrow \mathit{merge}(o_1, \mathit{merge}(o_2, o_3)) & S_7
\end{array}$$

Line S_1 defines an object with a single method named i. Each method of an object may reference the object using the identifier *self*. Lines S_2 and S_3 define similar objects. Line S_4 defines a composite object o using the associative binary operator *merge* to compose the object o_1, o_2 and o_3. The operator *merge* is required to cause shadowing on the right.

Lines S_5 – S_7 show the outcomes from sending messages to the object o. The outcomes show that the value of the identifier *self* in the object o depends upon which component of o handles the message. In each case the component which handles the message is the rightmost component in the object *self*. This is the point of view property of self reference;

its implementation is as follows:

$$\textbf{let } o_1(\mathit{self}) = (\text{`}i\text{'} \mapsto \lambda().\mathit{self}, o_1) \qquad I_1$$

$$\textbf{let } o_2(\mathit{self}) = (\text{`}j\text{'} \mapsto \lambda().\mathit{self}, o_2) \qquad I_2$$

$$\textbf{let } o_3(\mathit{self}) = (\text{`}k\text{'} \mapsto \lambda().\mathit{self}, o_3) \qquad I_3$$

$$\begin{array}{ll}
\textbf{let } \mathit{merge}(o_1, o_2)(t_1, t_2) = & I_4 \\
\quad \textbf{let} \;\; (e_1, t_3) = o_1(t_1) & \\
\quad \phantom{\textbf{let}} \;\; (e_2, t_4) = o_2(t_2) \textbf{ in} & \\
\quad \textbf{let} \;\; t_5 = \mathit{leaves}(\mathit{addleft}(\mathit{rightleaf}(t_4)))(t_3) & \\
\quad \phantom{\textbf{let}} \;\; t_6 = \mathit{leaves}(\mathit{addleft}(\mathit{rightleaf}(t_3)))(t_4) & \\
\quad \textbf{in } (t_5, t_6) &
\end{array}$$

$$\begin{array}{ll}
\textbf{let } \mathit{rightleaf}(t_1, t_2) = \mathit{rightleaf}(t_2) \mid & \\
\quad \mathit{rightleaf}(o) = o & I_5
\end{array}$$

$$\begin{array}{ll}
\textbf{let } \mathit{addleft}(o)(t_1, t_2) = (\mathit{addleft}(o)(t_1), \mathit{addleft}(o)(t_2)) \mid & \\
\quad \mathit{addleft}(o_1)(o_2) = \mathit{merge}(o_1, o_2) & I_6
\end{array}$$

$$\begin{array}{ll}
\textbf{let } \mathit{leaves}(f)(t_1, t_2) = (\mathit{leaves}(f)(t_1), \mathit{leaves}(f)(t_2)) \mid & \\
\quad \mathit{leaves}(f)(o) = f(o) & I_7
\end{array}$$

$$\textbf{let } \mathit{send}(o, i, v) = (\mathit{1st}(Y \lambda p.o(\mathit{2nd}(p)))) \bullet i(v) \qquad I_8$$

An object is implemented as a function which is applied to a binary tree t of objects and returns a pair (e, t) containing an environment e and a binary tree of objects. Each leaf of t is the value of *self* for the corresponding component of the object which t is supplied to; therefore the object which t is supplied to must have as many component objects as there are leaves in t.

Lines I_1 – I_3 show the implementation of atomic leaf-objects. Each leaf-object is a function with a formal parameter *self* and which returns a pair. The second component of the pair is the object itself.

Line I_4 shows the implementation of the object composition operator *merge*. This is applied to two objects o_1 and o_2 and returns a new object. The new object is a function which is applied to a binary tree containing sub-trees t_1 and t_2 and returns a pair containing an environment $e_1 \oplus e_2$ and a binary tree of objects (t_5, t_6).

The environments e_1 and e_2 are constructed by supplying the component objects o_1 and o_2 with the sub-trees t_1 and t_2. This also produces two trees t_3 and t_4 which are independent of each other. We require that every leaf of tree t_3 be extended on the left with the rightmost leaf of t_4 and that every leaf in t_4 be extended on the left with the rightmost leaf of t_3. This is achieved using the operators *rightleaf*, *addleft* and *leaves* which are defined in lines I_5 – I_7.

Finally, whenever a message is sent to an object the tree of objects which are used as values for *self* must be supplied to the receiver. In response to being applied to a tree of objects, an object produces a pair containing another tree of objects. We require that the supplied tree be the same as the produced tree and therefore can use Y to supply the required tree of objects. This is shown in the implementation of *send* in line I_8.

The operational mechanism of the point of view feature is simplified by defining calculation rules for message passing. Suppose that *send* is supplied with the following argument $((o_1, o_2), i, v)$ where o_1 is an environment which is the target of the message, o_2 is an environment which is used as the value of *self* in the closure which is activated, i is a method name and v is a value. The following calculation rules define the point of view feature:

$$(send : ((\{\}, \_), i, v) : s, e, @ : c, d, h) \longmapsto (\epsilon : s, e, c, d, h) \ R_{41}$$

Rule R_{41} shows that the result of sending a message to the empty environment is ϵ.

$$\begin{array}{l}(send : ((i \mapsto <j, e', b>, o), i, v) : s, e, @ : c, d, h) \longmapsto \\ \quad ([], e' \oplus (\text{`}self\text{'} \mapsto (o \oplus (i \mapsto <j, e', b>))) \oplus (j \mapsto v), [b], (s, e, c, d), h) \ R_{42}\end{array}$$

Rule R_{42} describes the case where a method with the correct name is found and activated. The object o is used for self reference and is extended with the binding for i.

$$\frac{i \neq j}{(send : ((i \mapsto \_, \_), j, \_) : s, e, @; c, d, h) \longmapsto (\epsilon : s, e, c, d, h)} \ R_{43}$$

Rule R_{43} describes the case where a method with the wrong name is found. The outcome is ϵ.

$$\frac{i \in dom(o_2)}{(send : ((o_1 \oplus o_2, o), i, v) : s, e, @ : c, d, h) \longmapsto (send : ((o_2, o \oplus o_1), i, v) : s, e, @; c, d, h)} \ R_{45}$$

Rule R_{45} describes the calculation which occurs when a composite object is sent a message which occurs in a right hand component. The component which does not contain the method (or is shadowed) is added to o.

$$\frac{i \notin dom(o_2)}{(send((o_1 \oplus o_2, o), i, v) : s, e, @ : c, d, h) \longmapsto (send : ((o_1, o \oplus o_2), i, v) : s, e, @; c, d, h)} \ R_{44}$$

Rule R_{44} describes the calculation which occurs when a composite object is sent a message which does not occur in the right hand component. The component which does not contain the method is added to o.

4.8 Feature 8: Labelled Self Reference

Feature 8 requires the framework to support composite objects where the components of the overall objects may be labelled and referenced individually. This is particularly important when an object "inherits" many component parts from different class definitions and methods wish to reference those parts of the object which originated from particular classes. Consider the following specification:

$$\textbf{let } o_1 = \{i() = \mathit{self} \bullet \text{`}\mathit{self}\text{'}\} \qquad S_1$$

$$\textbf{let } o_2 = \{l; j() = \mathit{self} \bullet \text{`}l\text{'}\} \qquad S_2$$

$$\begin{array}{ll}
\mathit{send}(o_1, \text{`}i\text{'}, ()) \Rightarrow o_1 & S_3 \\
\mathit{send}(o_2, \text{`}j\text{'}, ()) \Rightarrow o_2 & S_4 \\
\mathit{send}(\mathit{merge}(o_1, o_2), \text{`}i\text{'}, ()) \Rightarrow \mathit{merge}(o_1, o_2) & S_5 \\
\mathit{send}(\mathit{merge}(o_1, o_2), \text{`}j\text{'}, ()) \Rightarrow o_2 & S_6
\end{array}$$

Line S_1 defines an object o_1 with a single method i. The identifier *self* is scoped over the methods of an object and refers to an environment containing bindings between names and objects. The names in *self* are *self referential labels*. The default label is *self* which refers to the receiver of the message which invokes a method.

Line S_2 defines an object with a self referential label l. This label is used in the body of method j to look up an object in the environment *self*. The object which l refers to in *self* is that portion of the receiver of the message which corresponds to o_2. Note that in the example this *is* o_2 but in general objects may be defined anonymously.

Lines S_3 – S_6 show the outcomes of sending messages i and j to objects o_1 and o_2 individually and after they have been merged. The self referential label l always refers to o_2 whereas the label *self* always refers to the receiver of the message.

The Λ-calculus can implement labelled self reference using the following operator definitions:

$$\textbf{let } \mathit{null}(\_) = \{\} \qquad I_1$$

$$\textbf{let } \mathit{bind}(n, v)(e) = n \mapsto (I(R(v) \oplus (\text{`}\mathit{self}\text{'} \mapsto e), v)) \qquad I_2$$

$$\textbf{let } \mathit{merge}(o_1, o_2)(e) = o_1(e) \oplus o_2(e) \qquad I_3$$

$$\textbf{let } \mathit{label}(n, o)(e) = o(e \oplus (n \mapsto o)) \qquad I_4$$

$$\textbf{let } \mathit{send}(o, n, v) = (o(\text{`}\mathit{self}\text{'} \mapsto o)) \bullet n(v) \qquad I_5$$

$$\textbf{let } o_1 = \mathit{bind}(\text{`}i\text{'}, \lambda().\mathit{self} \bullet \text{`}\mathit{self}\text{'}) \qquad I_6$$

$$\textbf{let } o_2 = \mathit{label}(\text{`}l\text{'}, \mathit{bind}(\text{`}j\text{'}, \lambda().\mathit{self} \bullet \text{`}l\text{'})) \qquad I_7$$

Objects are implemented as functions which map environments to environments. They are constructed using *null* (I_1) for the empty object, *bind* (I_2) which corresponds to $\mapsto$ and associates a name n with a value v, *merge* (I_3) which corresponds to $\oplus$ and using *label* (I_4) which introduces a new self referential label n associated with an object o.

The environment which is passed to each object contains self referential labels mapped to objects. This environment is extended by *label* in line I_4 by passing the object o an environment $e \oplus (n \mapsto o)$. Lines I_6 and I_7 use the operators to implement the objects o_1 and o_2.

Objects with labelled self reference are essentially environments with labels introduced at various points. This can be conveniently represented by extending the environment constructors with $<i, e>$ which corresponds to a value constructed using the *label* operator in line I_4.

Objects which are constructed using the *merge* operator in I_3 are represented as $<o_1, o_2>$. The implementation is then correct with respect to the specification when the definition of objects o_1 and o_2 produce the following outcomes:

$$o_1 = \text{`}i\text{'} \mapsto <(), e, \mathit{self} \bullet \text{`}\mathit{self}\text{'}>$$

$$o_2 = <\text{`}l\text{'}, \text{`}j\text{'} \mapsto <(), e, \mathit{self} \bullet \text{`}l\text{'}>>$$

and the outcomes of the messages are as shown in the specification. The behaviour of the message passing operator *send* is described using the following calculation rules:

$$\frac{(\mathit{send} : (o, i, v_1, \text{`}\mathit{self}\text{'} \mapsto o) : s, e, @ : c, d, h_1) \longmapsto (v_2 : s, e, c, d, h_2)}{(\mathit{send} : (o, i, v_1) : s, e, @ : c, d, h_1) \longmapsto (v_2 : s, e, c, d, h_2)} \; R_{46}$$

Rule R_{46} describes the application of the operator *send*. The operator which is applied to a tuple of three components is overloaded so that it can be applied to a tuple of four components. The rest of the rules describe the outcomes of applying the 4-tuple version of *send*.

$$(\mathit{send} : (\{\}, \_, \_, \_) : s, e, @ : c, d, h) \longmapsto (\epsilon : s, e, c, d, h) \; R_{47}$$

Rule R_{47} shows that the outcome sending an empty object a message is always ϵ.

$$\frac{(<j, e_3 \oplus (\text{`}\mathit{self}\text{'} \mapsto e_2), b> : v_1 : s, e_3, @ : c, d, h_1) \longmapsto (v_2 : s, e_3, c, d, h_2)}{(\mathit{send} : (i \mapsto <j, e_1, b>, i, v_1, e_2) : s, e_3, @ : c, d, h_1) \longmapsto (v_2 : s, e_3, c, d, h_2)} \; R_{48}$$

Rule R_{48} shows the outcome of sending a message to a singleton object. When the name of the message is the same as the binding in the object then the result is produced by applying the corresponding closure. The environment of the closure is extended with a binding associating *self* with the environment of self referential labels.

$$\frac{i \neq j}{(\mathit{send} : (i \mapsto \_, j, \_, \_) : s, e, @ : c, d, h) \longmapsto (\epsilon : s, e, c, d, h)} \; R_{49}$$

Rule R_{49} shows the outcome of sending a message to a singleton object where the name of the message does not correspond to the name bound by the object.

$$\frac{\begin{array}{c}(\mathit{send}(o, i, v_1, e_1) : s, e_2, @ : c, d, h_1) \longmapsto (v_2 : s, e_2, c, d, h_2) \\ v_2 \neq \epsilon\end{array}}{(\mathit{send} : (<\_, o>, i, v_1, e_1) : s, e_2, @ : c, d, h_1) \longmapsto (v_2 : s, e_2, c, d, h_2)} \; R_{50}$$

Rule R_{50} shows the outcome of sending a message to a composite object (constructed using the *merge* operator). If the outcome of sending the message to the right sub-component of the object is not ϵ then this is the outcome of the composite object.

$$\frac{\begin{array}{c}(\mathit{send}(o_2, i, v_1, e_1) : s, e_2, @ : c, d, h_1) \longmapsto (\epsilon : s, e_2, c, d, h_1) \\ (\mathit{send}(o_1, i, v_1, e_1) : s, e_2, @ : c, d, h_1) \longmapsto (v_2 : s, e_2, c, d, h_2)\end{array}}{(\mathit{send} : (<o_1, o_2>, i, v_1, e_1) : s, e_2, @ : c, d, h_1) \longmapsto (v_2 : s, e_2, c, d, h_2)} \; R_{51}$$

Rule R_{51} shows the outcome of sending a message to a composite object where the right hand sub-component produces ϵ in which case the outcome is defined by the left hand sub-component.

$$\frac{(\mathit{send} : (o, j, v_1, e_1 \oplus (i \mapsto o)) : s, e_2, @ : c, d, h_1) \longmapsto (v_2 : s, e_2, c, d, h_2)}{(\mathit{send} : (<i, o>, j, v_1, e_1) : s, e_2, @ : c, d, h_1) \longmapsto (v_2 : s, e_2, c, d, h_2)} \; R_{52}$$

Rule R_{52} describes the outcome of sending a message to an object which introduces a new label i. The result is defined by sending the message to the sub-object o after extending the environment of self referential labels.

4.9 Feature 9: Dynamic Inheritance

Feature 9 requires that the framework be able to support dynamic inheritance. In many object-oriented systems, when an object is extended, a new object is produced and the extensions do not affect the original object. Dynamic inheritance allows an object to be changed by side effect so that all occurrences of the objects which share are affected. Dynamic inheritance is provided by Self as described in chapter 2 and can be used to change an object's implementation to reflect its current state, for example to allow general polygon objects which have exactly 4 edges to be drawn on the screen using a special purpose operation. A specification for dynamic inheritance is as follows:

$$\textbf{let } o_1 = \{x = 10; y = 20\} \qquad S_1$$

$$\textbf{let } o_2 = \{x = 30; y = 40\} \qquad S_2$$

$$\textbf{let } o_3 = \{z = 50; o =_d o_1\} \qquad S_3$$

$$\begin{array}{ll}
\textit{ref}(o_3, \text{`}o\text{'}) \Rightarrow o_1 & S_4 \\
\textit{ref}(o_3, \text{`}x\text{'}) \Rightarrow 10 & S_5 \\
\textit{ref}(o_3, \text{`}y\text{'}) \Rightarrow 20 & S_6 \\
\textit{set}(o_1, \text{`}x\text{'}, 60) \Rightarrow 60 & S_7 \\
\textit{ref}(o_3, \text{`}x\text{'}) \Rightarrow 60 & S_8 \\
\textit{set}(o_3, \text{`}x\text{'}, 70) \Rightarrow 70 & S_9 \\
\textit{ref}(o_1, \text{`}x\text{'}) \Rightarrow 70 & S_{10} \\
\textit{set}(o_3, \text{`}o\text{'}, o_2) \Rightarrow o_2 & S_{11} \\
\textit{ref}(o_3, \text{`}x\text{'}) \Rightarrow 30 & S_{12}
\end{array}$$

Lines $S_1 - S_3$ each define objects. Both o_1 and o_2 define fields named x and y. The object o_3 defines fields named z and o. The o field is defined to be *dynamic* because it is defined using $=_d$. A dynamic field refers to a program location, like non-dynamic fields, but the field value must be an object. The effect is to add all of the fields from the object *in* the dynamic field to the object *containing* the dynamic field. The result of defining o_3 in line S_3 is an object with four fields named z, o, x and y.

Since a dynamic field corresponds to a program location, its value may be updated by side effect. If the contents of a dynamic field is updated then the fields which it contributes to the object containing the dynamic field will change accordingly.

Lines $S_4 - S_{12}$ show the results of referencing the fields x and y of o_3 which are contributed by the dynamic field o. Lines S_7 and S_8 show that fields contributed by a dynamic field are shared since updating the contents of x in object o_1 also updates field x in o_3. The converse is shown to be true in lines S_9 and S_{10}.

Lines S_{11} and S_{12} show that if the contents of a dynamic field is updated then the fields which it contributes are updated accordingly. In the example shown, the fields which are contributed by the value of dynamic field o in o_3 always have the same names, in general this is not necessarily the case.

We describe two ways of implementing dynamic inheritance in the Λ-calculus: the first involves representing objects as pairs $(e, \tilde{i})$ where e is an environment mapping field names to program locations and $\tilde{i}$ is a sequence of the dynamic field names. The lookup operator then knows that it must visit the contents of each dynamic field in addition to the fields of e. The second representation involves extending the underlying Λ-machine with a new environment lookup operator which allows environments to contain machine locations. On encountering a machine location in an environment the machine lookup mechanism will continue lookup using the contents of the location. The first implementation is as follows:

$$\textbf{let } o_1 = ((\text{`}x\text{'} \mapsto loc(10)) \oplus (\text{`}y\text{'} \mapsto loc(20)), []) \qquad I_1$$

$$\textbf{let } o_2 = ((\text{`}x\text{'} \mapsto loc(30)) \oplus (\text{`}y\text{'} \mapsto 40)), []) \qquad I_2$$

$$\textbf{let } o_3 = ((\text{`}z\text{'} \mapsto loc(50)) \oplus (\text{`}o\text{'} \mapsto loc(o_1)), [\text{`}o\text{'}]) \qquad I_3$$

$$\textbf{let } \mathit{field}((e, l), i) = ((e \bullet i)?(lh(?)(\mathit{field}(\_, i))(\epsilon)(l))) \qquad I_4$$

$$\textbf{let } \mathit{ref}(o, i) = !(\mathit{field}(o, i)) \qquad I_5$$

$$\textbf{let } \mathit{set}(o, i, v) = (\mathit{field}(o, i)) := v \qquad I_6$$

Line I_1 defines object o_1 as a pair containing an environment binding field names to program locations and an empty list of dynamic field names. Line I_2 defines o_2 as a similar pair. Line I_3 defines o_3 as a pair where the list of dynamic field names is [‘o’].

Line I_4 defines the field reference operator *field* which returns the program location which is reachable given an object o and a field name i. The location will either be that contained in the object o and bound to i: $o \bullet i$ or will be the result of recursively referencing the fields named i in the contents of the dynamic fields of o. The list l contains the names of the dynamic fields of o and the expression:

$$lh(?)(\mathit{field}(\_, i))(\epsilon)(l)$$

recursively visits each dynamic field value and references the field named i.

Lines I_5 and I_6 define the operators *ref* and *set* which return the contents of a field and update a field respectively. Both of these operators use *field* to extract the required program location.

The implementation is correct with respect to the specification when the outcomes of defining the objects o_1, o_2 and o_3 are as follows:

$$o_1 = ((\text{`}x\text{'} \mapsto \alpha_1) \oplus (\text{`}y\text{'} \mapsto \alpha_2), ()) \qquad V_1$$

$$o_2 = ((\text{`}x\text{'} \mapsto \alpha_3) \oplus (\text{`}y\text{'} \mapsto \alpha_4), ()) \qquad V_2$$

$$o_3 = ((\text{`}z\text{'} \mapsto \alpha_5) \oplus (\text{`}o\text{'} \mapsto \alpha_6), \alpha_6) \qquad V_3$$

$$\begin{aligned} h_1 = \ & (\alpha_1 \mapsto 10) \oplus (\alpha_2 \mapsto 20) \oplus (\alpha_3 \mapsto 30) \oplus \qquad V_4 \\ & (\alpha_4 \mapsto 40) \oplus (\alpha_5 \mapsto 50) \oplus (\alpha_6 \mapsto o_1) \end{aligned}$$

$$h_2 = h_1 \oplus (\alpha_1 \mapsto 60) \qquad V_5$$

$$h_3 = h_2 \oplus (\alpha_1 \mapsto 70) \qquad V_6$$

$$h_4 = h_3 \oplus (\alpha_6 \mapsto o_2) \qquad V_7$$

and the outcomes of the messages have the specified outcomes. Lines $V_1 - V_3$ show the objects, line V_4 shows the initial heap after the objects have been constructed, V_5 shows the heap after the update in line S_7 has been performed, line V_6 shows the heap after the update in line S_9 has been performed and line V_7 shows the heap after the update in line S_{11} has been performed.

Dynamic inheritance operators can be described by calculation rules which determine the outcomes of applying the operators as though they were built-in to the Λ-machine. An object expression is represented as $o(\tilde{i}, \tilde{j}, \tilde{e})$ where $\tilde{i}$ is a sequence of field names, $\tilde{j}$ is a sequence of dynamic field names (which are also contained in $\tilde{i}$) and $\tilde{e}$ is a sequence if initial field value expressions.

The calculation rule for performing object expressions is R_{53}. Each of the initial value expressions is evaluated in turn and program locations are allocated for the results. The result is a pair $(e, \tilde{\alpha})$ where e is an environment mapping field names to program locations and $\tilde{\alpha}$ is a sequence of dynamic field program locations.

$$\frac{\begin{array}{c}(s, e, \tilde{e}_i : c, d, h_{i-1}) \longmapsto (\tilde{v}_i : s, e, c, d, h_i) \ i = 1, n \\ \#(\tilde{e}) = \#(\tilde{i}) = \#(\tilde{v}) = n \wedge e = \tilde{i} \mapsto \tilde{\alpha}\end{array}}{(s, e, o(\tilde{i}, \tilde{j}, \tilde{e}) : c, d, h_0) \longmapsto ((o, o \bullet \tilde{j}) : s, e, c, d, h_n \oplus (\tilde{\alpha} \mapsto \tilde{v}))} R_{53}$$

The calculation rule for applying the operator *field* is R_{54}. The outcome of applying this operator is a value which is constructed by looking the name i up in the object: $o \bullet i$ and in the objects which are contained in the dynamic fields of o: $h \bullet \tilde{\alpha} = \tilde{o}$.

$$\frac{\begin{array}{c}(\mathit{field} : (\tilde{o}_j, i) : s, e, @ : c, d, h) \longmapsto (\tilde{v}_j : i : s, e, c, d, h) \ j = 1, \#(\tilde{\alpha}) \\ o \bullet i = v \\ h \bullet \tilde{\alpha}_j = \tilde{o}_j\end{array}}{(\mathit{field} : ((o, \tilde{\alpha}), i) : s, e, @ : c, d, h) \longmapsto (v?(?(\tilde{v})) : s, e, c, d, h)} R_{54}$$

The calculation rule for applying the operators *ref* and *set* follow directly from the rule for *field* and are defined as R_{55} and R_{56}.

$$\frac{(\mathit{field} : (o, i) : s, e, @; c, d, h) \longmapsto (\alpha : s, e, c, d, h)}{(\mathit{ref} : (o, i) : s, e, @ : c, d, h) \longmapsto ((h \bullet \alpha) : s, e, c, d, h)} R_{55}$$

$$\frac{(\mathit{field} : (o,i) : s, e, @; c, d, h) \longmapsto (\alpha : s, e, c, d, h)}{(\mathit{set} : (o,i,v) : s, e, @ : c, d, h) \longmapsto (v : s, e, c, d, h \oplus (\alpha \mapsto v))}\ R_{56}$$

These rules can be used to show that the implementation given in lines $I_1 - I_{12}$ is correct with respect to the specification given in lines $S_1 - S_{12}$. The second method of implementing dynamic inheritance is to modify the Λ-machine environment lookup operator as follows:

$$\begin{array}{ll}
\{\} \bullet _ = \epsilon & | \\
(k_1 \mapsto v) \bullet k_2 = \begin{cases} v & \textbf{when } k_1 = k_2 \\ \epsilon & \textbf{otherwise} \end{cases} & | \\
(e_1 \oplus e_2) \bullet k = (e_1 \bullet k)\,?(e_2 \bullet k) & | \\
(\alpha) \bullet k = (h \bullet \alpha) \bullet k &
\end{array}$$

where the current machine heap is h. This mechanism allows the machine environment component to contain machine locations and environments which contain machine locations. If this operator is used for the underlying machine semantics then the specification of dynamic inheritance given in lines $S_1 - S_{12}$ is as follows:

$$\textbf{let } o_1 = (\text{`}x\text{'} \mapsto \mathit{loc}(10)) \oplus (\text{`}y\text{'} \mapsto \mathit{loc}(20))$$

$$\textbf{let } o_2 = (\text{`}x\text{'} \mapsto \mathit{loc}(30)) \oplus (\text{`}y\text{'} \mapsto \mathit{loc}(40))$$

$$\textbf{let } o_3 = \textbf{let } l = \mathit{loc}(o_1) \textbf{ in } (\text{`}z\text{'} \mapsto \mathit{loc}(50)) \oplus (\text{`}o\text{'} \mapsto l) \oplus l$$

$$\textbf{let } \mathit{ref}(o,i) = !(o \bullet i)$$

$$\textbf{let } \mathit{set}(o,i,v) = o \bullet i := v$$

which is considerably simpler than the first implementation but requires that the machine semantics be modified to allow environments to contain program locations.

4.10 Feature 10: Name Updates

Few object-oriented languages allow an object's name to be changed once it has been created. Although objects may have the same names in their interfaces, most languages do not allow the names in an interface to be designated as a program location which may be shared with other objects and updated by side effect. The Λ-calculus provides fine-grain access to program locations and can support a wide variety of program features based upon sharing and update. A specification for a simple feature allowing the names of object's fields to be updated by side effect is as follows:

$$
\begin{array}{ll}
\textbf{let } o = \{i = 10; j = 20\} & S_1 \\
 & \\
\mathit{ref}(o, `i\text{'}) \Rightarrow 10 & S_2 \\
\mathit{ref}(o, `j\text{'}) \Rightarrow 20 & S_3 \\
\mathit{setname}(o, `i\text{'}, `k\text{'}) \Rightarrow `k\text{'} & S_4 \\
\mathit{ref}(o, `i\text{'}) \Rightarrow \epsilon & S_5 \\
\mathit{ref}(o, `j\text{'}) \Rightarrow 20 & S_6 \\
\mathit{ref}(o, `k\text{'}) \Rightarrow 10 & S_7 \\
\mathit{setname}(o, `k\text{'}, `j\text{'}) \Rightarrow `j\text{'} & S_8 \\
\mathit{ref}(o, `i\text{'}) \Rightarrow \epsilon & S_9 \\
\mathit{ref}(o, `j\text{'}) \Rightarrow 20 & S_{10} \\
\mathit{ref}(o, `k\text{'}) \Rightarrow \epsilon & S_{11}
\end{array}
$$

Line S_1 defines an object with two fields named j and j respectively, each field represents a program location and the initial values are 10 and 20. Lines S_2 – S_{11} show the results of referencing the fields before and after renaming operations. Lines S_2 and S_3 show that the field referencing operator *ref* produces the current values of the fields named i and j respectively. Line S_4 renames the field named i to be named k using the renaming operator *setname*. Lines S_5 – S_7 show that the field previously named i in the object o has been changed to k. Lines S_8 – S_{11} show that if a field is renamed to a name already existing in the object and the existing field is right of the redefined field then the redefinition occurs but the renamed field is inaccessible.

We describe two ways of implementing this feature using the Λ-calculus. The first mechanism represents objects as program locations containing environments which map field names to field values. Update of both field names and field values is performed by updating the program location containing the environment. The second implementation involves changing the environment lookup mechanism which is performed by the underlying Λ-machine to allow environments to contain program locations. The first implementation is as follows:

$$
\begin{array}{ll}
\textbf{let } o = \mathit{loc}((`i\text{'} \mapsto 10) \oplus (`j\text{'} \mapsto 20)) & I_1 \\
 & \\
\textbf{let } \mathit{ref}(o, i) = (!o) \bullet i & I_2 \\
 & \\
\textbf{let } \mathit{setname}(o, i_1, i_2) = o := (!o)[i_1/i_2] & I_3
\end{array}
$$

Line I_1 defines o to be a program location containing an environment which maps names i and j to the values 10 and 20 respectively. Line I_2 defines the field referencing operator *ref* which extracts the contents of the program location o and then looks up the name i. Line I_3 defines the field name updater *setname* which renames the environment contained in the program location o by side effect.

The implementation I_1 – I_3 of the feature specified in lines S_1 – S_{11} is correct when the object o is as follows:

$$
\begin{array}{l}
o = \alpha \\
h_1 = \alpha \mapsto ((`i\text{'} \mapsto 10) \oplus (`j\text{'} \mapsto 20)) \\
h_2 = \alpha \mapsto ((`k\text{'} \mapsto 10) \oplus (`j\text{'} \mapsto 20)) \\
h_3 = \alpha \mapsto ((`j\text{'} \mapsto 10) \oplus (`j\text{'} \mapsto 20))
\end{array}
$$

where h_1, h_2 and h_3 are the fragments of the heap corresponding to object creation, renaming field i to k and renaming field k to j respectively. In order to show that the field reference and renaming operators produce the specified outcomes, the following calculation rules are defined.

An object expression is represented as $o(\tilde{i}, \tilde{e})$ where $\tilde{i}$ is a sequence of field names and $\tilde{e}$ is a sequence of initial values corresponding to the field names. The rule R_{57} defines the calculation which occurs when an object definition is performed. Each of the initial field values $\tilde{e}_i$ are performed and a new program location is produced which contains an environment mapping field names to field values.

$$\frac{\begin{array}{c}(s, e, \tilde{e}_i : c, d, h_{i-1}) \mapsto (\tilde{v}_i : s, e, c, d, h_i)\ i = 1, n \\ \#(\tilde{i}) = \#(\tilde{v}) = n\end{array}}{(s, e, o(\tilde{i}, \tilde{e}) : c, d, h_0) \longmapsto (\alpha : s, e, c, d, h_n \oplus (\alpha \mapsto (\tilde{i} \mapsto \tilde{v})))}\ R_{57}$$

The rule R_{58} defines the calculation which occurs when the operator *ref* is applied to a program location α. The program location should contain an environment $h \bullet \alpha$ and the outcome is the value associated with the field name by the environment $(h \bullet \alpha) \bullet i$.

$$(\mathit{ref} : (\alpha, i) : s, e, @ : c, d, h) \longmapsto ((h \bullet \alpha) \bullet i : s, e, c, d, h)\ R_{58}$$

The rule R_{59} defines the calculation which occurs when the operator *setname* is applied to a program location α and two field names i_1 and i_2. The outcome is the new field name i_2 and the environment held in the location α is renamed as a side effect.

$$(\mathit{setname} : (o, i_1, i_2) : s, e, @ : c, d, h) \longmapsto (i_2 : s, e, c, d, h \oplus (\alpha \mapsto (h \bullet \alpha)[i_2/i_1]))\ R_{59}$$

These rules can be used to show that the implementation is correct with respect to the specification. The second implementation involves extending the environment lookup operator which is used by the underlying Λ-machine. An environment is allowed to contain mappings of the form $\alpha \mapsto v$ where α is a program location containing an identifier. The machine lookup operator is extended to dereference the program location:

$$\begin{array}{ll}
\{\} \bullet _ = \epsilon & | \\
(k_1 \mapsto v) \bullet k_2 = \begin{cases} v & \textbf{when } k_1 = k_2 \\ \epsilon & \textbf{otherwise} \end{cases} & | \\
(e_1 \oplus e_2) \bullet k = (e_1 \bullet k)\,?(e_2 \bullet k) & | \\
(\alpha \mapsto v) \bullet k = ((h \bullet \alpha) \mapsto v) \bullet k &
\end{array}$$

A new operator is required in order to produce the program location which contains a particular identifier in an environment. The operator is built-in to the machine and is defined as follows:

$$\begin{array}{ll}
\{\} \odot _ = \epsilon & | \\
(e_1 \oplus e_2) \odot k = (e_1 \odot k)\,?(e_2 \odot k) & | \\
(\alpha \mapsto _) \odot k = \begin{cases} \alpha & \textbf{when } (h \bullet \alpha) = k \\ \epsilon & \textbf{otherwise} \end{cases} &
\end{array}$$

Identifiers can now be updated using the following implementation:

$$\textbf{let } o = (\mathit{loc}(\text{'}i\text{'}) \mapsto \mathit{loc}(10)) \oplus (\mathit{loc}(\text{'}j\text{'}) \mapsto \mathit{loc}(20))\quad I_1$$

$$\textbf{let } \mathit{ref}(o, i) = !(o \bullet i)\quad I_2$$

$$\textbf{let } \mathit{setname}(o, i_1, i_2) = o \odot i_1 := i_2\quad I_3$$

Line I_1 defines the object o as an environment in which program locations containing the field names are associated with program locations containing the initial values. Line I_2 defines the field reference operator *ref* which looks up the name i and returns the contents of the resulting program location. Line I_3 defines the field renaming operator which finds the program location associated with the field name i_1 and updates the contents to be i_2.

4.11 Feature 11: Classes as Object Abstractions

Feature 11 requires that the framework supports classes as abstractions over objects. A simple class defines all the names and some of the values which will occur in objects which are referred to as its *instances*. The class is instantiated by supplying the missing values and the result is an object. There is no limit to the number of times which a class can be instantiated. The following is a specification for a simple class:

$$\begin{array}{ll}
\textbf{let } c = & S_1 \\
\quad \textbf{class } (x) & \\
\qquad getx() = !x & \\
\qquad setx(v) = x := v & \\
\quad \textbf{end} & \\
& \\
\textbf{let } o_1 = new(c, 10) & S_2 \\
& \\
\textbf{let } o_2 = new(c, 20) & S_3 \\
& \\
send(o_1, \text{`}getx\text{'}, ()) \Rightarrow 10 & S_4 \\
send(o_2, \text{`}getx\text{'}, ()) \Rightarrow 20 & S_5 \\
send(o_1, \text{`}setx\text{'}, 30) \Rightarrow 30 & S_6 \\
send(o_1, \text{`}getx\text{'}, ()) \Rightarrow 30 & S_7 \\
send(o_2, \text{`}getx\text{'}, ()) \Rightarrow 20 & S_8
\end{array}$$

Line S_1 defines a class c which has a single attribute named x and two methods named *getx* and *setx*. A class is instantiated to produce an object which is referred to as its instance. When the class is instantiated, it must be supplied with initial values for all of its attributes. New program locations are allocated for the attributes and set to the initial values. The attributes may be referenced in the bodies of the methods and different instances of the class will have different program locations. The methods are invoked in an instance by sending it a message using the *send* operator.

Lines S_2 and S_3 create two objects o_1 and o_2 by instantiating c using the operator *new* which is applied to a pair containing a class and initial values for the attributes of the class. Lines S_4 – S_8 show the outcomes of sending *getx* and *setx* messages to the two objects. This also shows that the instances o_1 and o_2 are associated with different program locations for the attribute x.

The Λ-calculus can be used to implement classes as functions. The arguments of the function correspond to the missing values which are supplied when the class is instantiated. The body of the function is an expression which, when performed, will construct an environment which implements the instances of the class. Class instantiation is implemented in the

Λ-calculus as function application. The implementation is as follows:

$$
\begin{array}{ll}
\textbf{let } c(x) = & I_1 \\
\quad \textbf{let } x = loc(x) & \\
\quad \textbf{in } (\text{`}getx\text{'} \mapsto \lambda().!x) \oplus (\text{`}setx\text{'} \mapsto \lambda v.x := v) & \\
\\
\textbf{let } new(c, v) = c(v) & I_2 \\
\\
\textbf{let } send(o, i, v) = o \bullet i(v) & I_3
\end{array}
$$

Line I_1 defines a class c as a function which is supplied with an initial value for the attribute x, allocates a program location with x as the initial value and returns an environment binding the method names $getx$ and $setx$ to functions. The attribute name x is scoped over the bodies of the functions. Line I_2 defines the instantiation operator new which simply applies the given class c to the initial values for its attributes v. Line I_3 defines the message passing operator $send$ which looks the method named i up in the object o and applies it to the message argument v.

A class expression is implemented as a λ-expression which has attribute names as parameters and returns an environment binding names to function closures. Such an expression is represented as $k(\tilde{i}, \tilde{j}, \tilde{k}, \tilde{b})$ where $\tilde{i}$ are attribute names, $\tilde{j}$ are method names, $\tilde{k}$ are method arguments and $\tilde{b}$ are method bodies. For example:

$$k(\text{`}x\text{'}, (\text{`}getx\text{'}, \text{`}setx\text{'}), ((), \text{`}v\text{'}), (!x, x := v))$$

The result of performing a class definition is a class which is implemented as a function closure. Such a closure is represented as $c(\tilde{i}, e, \tilde{j}, \tilde{k}, \tilde{b})$ where $\tilde{i}$ are attribute names, e is an environment referred to as a class closure environment (corresponding to a closure environment e in $<i, e, b>$), $\tilde{j}$ are method names, $\tilde{k}$ are method arguments and $\tilde{b}$ are method bodies.

Given the above definitions, the implementation is correct with respect to the specification when the objects o_1 and o_2 are defined as:

$$c = c(\text{`}x\text{'}, e, (\text{`}getx\text{'}, \text{`}setx\text{'}), ((), \text{`}v\text{'}), (!x, x := v)) \qquad V_1$$

$$o_1 = (\text{`}getx\text{'} \mapsto <(), e \oplus (\text{`}x\text{'} \mapsto \alpha_1), !x>) \oplus (\text{`}setx\text{'} \mapsto <\text{`}v\text{'}, e \oplus (\text{`}x\text{'} \mapsto \alpha_1), x := v>) \qquad V_2$$

$$o_2 = (\text{`}getx\text{'} \mapsto <(), e \oplus (\text{`}x\text{'} \mapsto \alpha_2), !x>) \oplus (\text{`}setx\text{'} \mapsto <\text{`}v\text{'}, e \oplus (\text{`}x\text{'} \mapsto \alpha_2), x := v>) \qquad V_3$$

$$h_1 = h \oplus (\alpha_1 \mapsto 10) \qquad V_4$$

$$h_2 = h_1 \oplus (\alpha_2 \mapsto 20) \qquad V_5$$

$$h_3 = h_2 \oplus (\alpha_1 \mapsto 30) \qquad V_6$$

Line V_1 shows the outcome of defining the class c, the current environment when c is defined is e. Lines V_2 and V_3 show the outcomes of instantiating c to produce the objects o_1 and o_2. Line V_4 shows the heap after defining o_1, h is the heap before c is instantiated. Line V_5 shows the heap after o_2 has been defined and line V_6 shows the heap after the x attribute of o_1 has been updated.

The following calculation rules define what happens when classes are created and instantiated and when messages are sent. Rule R_{60} defines the calculation which occurs when a class is created. A class value is produced which has captured the current environment e as the class closure environment.

$$(s, e, k(\tilde{i}, \tilde{j}, \tilde{k}, \tilde{b}) : c, d, h) \longmapsto (c(\tilde{i}, e, \tilde{j}, \tilde{k}, \tilde{b}) : s, e, c, d, h) \;\; R_{60}$$

Rule R_{61} defines the calculation which occurs when a class is instantiated by applying the operator *new* to it and attribute initialisation values $\tilde{v}$. The result is an environment which maps method names $\tilde{j}$ to function closures. Fresh program locations are allocated for each of the attributes $\tilde{i}$, initialised to the values $\tilde{v}$. An environment mapping attribute names $\tilde{i}$ to the program locations $\tilde{\alpha}$ is part of each closure environment.

$$\begin{array}{l}(new : (c(\tilde{i}, e_1, \tilde{j}, \tilde{k}, \tilde{b}), \tilde{v}) : s, e_2, @ : c, d, h) \longmapsto \\ \quad (\tilde{j} \mapsto \,<\tilde{k}, e_1 \oplus (\tilde{i} \mapsto \tilde{\alpha}), \tilde{b}>\, : s, e_2, c, d, h \oplus (\tilde{\alpha} \mapsto \tilde{v})) \;\; R_{61}\end{array}$$

4.12 Feature 12: Sharing Between Instances

Feature 12 requires that the framework supports the sharing of values between different instances of the same class. This is a simple version of the mechanism which is provided by certain languages, for example it is called *class variables* in Smalltalk and *static members* in C++. When a class is instantiated, fresh program locations are allocated for the attributes of the resulting object. Different instances of the same class will have attributes with the same names but the names will refer to different program locations. The class variable feature allows the program locations associated with designated attribute names to be fixed when a class is defined. Instances of such a class will share program locations for *class variables* but will have distinct program locations for *instance variables*. The following is a specification for a class variable feature:

$$\begin{array}{ll}
\textbf{let } c_1 = & S_1 \\
\quad \textbf{class } (x, y) & \\
\quad\quad getx() = !x & \\
\quad\quad gety() = !y & \\
\quad\quad setx(v) = x := v & \\
\quad\quad sety(v) = y := v & \\
\quad \textbf{end} & \\
& \\
\textbf{let } c_2 = \mathit{seal1st}(c_1, 10) & S_2 \\
& \\
\textbf{let } o_1 = new(c_2, 20) & S_3 \\
& \\
\textbf{let } o_2 = new(c_2, 30) & S_4
\end{array}$$

$$\begin{array}{ll}
send(o_1, \text{`}sety\text{'}, 40) \Rightarrow 40 & S_5 \\
send(o_1, \text{`}gety\text{'}, ()) \Rightarrow 40 & S_6 \\
send(o_2, \text{`}gety\text{'}, ()) \Rightarrow 30 & S_7 \\
send(o_1, \text{`}setx\text{'}, 50) \Rightarrow 50 & S_8 \\
send(o_1, \text{`}getx\text{'}, ()) \Rightarrow 50 & S_9 \\
send(o_2, \text{`}getx\text{'}, ()) \Rightarrow 50 & S_{10}
\end{array}$$

Line S_1 defines a class c_1 which has two attributes named x and y and four methods for accessing and updating the attributes. Line S_2 defines a class c_2 which is created by supplying the first attribute of c_1 with an initial value 10. This is referred to as *sealing* the attribute and is performed by the operator *seal1st*. The class c_2 has two attributes named x and y, the attribute x is already associated with a program location containing 10. When c_2 is instantiated an initial value will be supplied for y.

Lines S_3 and S_4 define two objects o_1 and o_2 which are instances of c_2. The object o_1 initialises attribute y to 20 and o_2 initialises y to 30. Lines S_5 – S_{10} show the outcomes of sending messages to the objects o_1 and o_2. In particular, these messages show that the objects do not share the program locations for their respective y attributes whereas they both share the same program location for the attribute x.

The Λ-calculus can be used to implement class variable behaviour by representing classes as functions whose arguments correspond to the class attributes and sealing the class by partially applying it. Partial application of a function involves supplying it with some of its parameters but not all. A partially applied function must be applied to the rest of its parameters in order to become *saturated* and consequently be activated. The implementation is as follows:

$$
\begin{array}{ll}
\textbf{let}\ c_1(x)(y) = & I_1 \\
\quad \textbf{let}\ \ x = \mathit{loc}(x) & \\
\quad\quad\ \ y = \mathit{loc}(y) & \\
\quad \textbf{in}\ (\text{`}\mathit{getx}\text{'} \mapsto \lambda().!x) \oplus & \\
\quad\quad (\text{`}\mathit{gety}\text{'} \mapsto \lambda().!y) \oplus & \\
\quad\quad (\text{`}\mathit{setx}\text{'} \mapsto \lambda v.x := v) \oplus & \\
\quad\quad (\text{`}\mathit{sety}\text{'} \mapsto \lambda v.y := v) & \\
& \\
\textbf{let}\ \mathit{seal1st}(c, x)(y) = c(x)(y) & I_2 \\
& \\
\textbf{let}\ \mathit{new}(c, v) = \mathit{uncurry}(c, \#(v))(v) & I_3 \\
& \\
\textbf{let}\ \mathit{send}(o, i, v) = o \bullet i(v) & I_4
\end{array}
$$

Line I_1 defines a class c_1 as a curried function of two arguments x and y which returns an environment mapping method names to function closures. When the class is applied to values for x and y, fresh program locations are allocated and initialised to the supplied values. The names x and y are scoped over the function bodies.

Line I_2 implements the sealing operator *seal1st* so that it partially applies the class c to the value for the first attribute x. The result is a function of one parameter y. Line I_3 implements class instantiation as function application. The value v is a tuple of all the outstanding attribute values which are required to saturate c. This cannot be supplied directly to c because c is curried and expects its arguments one at a time. This is remedied by the use of a simple uncurrying operator *uncurry* which is applied to a function and a number of arguments and behaves as follows:

$$
\begin{array}{l}
(\mathit{uncurry} : (< i_1, e_1, \lambda i_2.\lambda.i_3.\ldots.\lambda.i_n.b >, n) : s, e_2, @ : c, d, h) \longmapsto \\
\quad (< (i_1, i_2, i_3, \ldots, i_n), e_1, b >: s, e_2, c, d, h)\ R_{62}
\end{array}
$$

Line I_4 defines the message passing operator which looks the function named i up in the environment o and applies it to the value v.

The implementation is correct with respect to the specification when the objects o_1 and o_2 are as follows:

$$
\begin{aligned}
&o_1 = (\text{`}getx\text{'} \mapsto k_1) \oplus (\text{`}setx\text{'} \mapsto k_2) \oplus (\text{`}gety\text{'} \mapsto k_3) \oplus (\text{`}sety\text{'} \mapsto k_4) \\
&o_2 = (\text{`}getx\text{'} \mapsto k_5) \oplus (\text{`}setx\text{'} \mapsto k_6) \oplus (\text{`}gety\text{'} \mapsto k_7) \oplus (\text{`}sety\text{'} \mapsto k_8) \\
&k_1 =< (), e \oplus e_1, !x >, k_2 =< \text{`}v\text{'}, e \oplus e_1, x := v > \\
&k_3 =< (), e \oplus e_1, !y >, k_4 =< \text{`}v\text{'}, e \oplus e_1, y := v > \\
&k_5 =< (), e \oplus e_2, !x >, k_6 =< \text{`}v\text{'}, e \oplus e_2, x := v > \\
&k_7 =< (), e \oplus e_2, !y >, k_8 =< \text{`}v\text{'}, e \oplus e_2, y := v > \\
&e_1 = e_3 \oplus e_4, e_2 = e_3 \oplus e_5 \\
&e_3 = \text{`}x\text{'} \mapsto \alpha_1, e_4 = \text{`}y\text{'} \mapsto \alpha_2, e_5 = \text{`}y\text{'} \mapsto \alpha_3 \\
&h_1 = h \oplus (\alpha_1 \mapsto 10) \oplus (\alpha_2 \mapsto 20) \oplus (\alpha_3 \mapsto 30) \\
&h_2 = h_1 \oplus (\alpha_2 \mapsto 40), h_3 = h_2 \oplus (\alpha_1 \mapsto 50)
\end{aligned}
$$

and the outcomes of sending the messages in lines S_5 – S_{10} agree with the specification. The heap h_1 describes the machine heap after the objects o_1 and o_2 have been created. The heap h_2 describes the situation after y has been set in line S_5 and heap h_3 describes the situation after x has been set in line S_8.

Class expressions and values are represented as described in §4.11 and the calculation rules for performing class definitions, class instantiation and message passing are R_{60}, R_{61} and R_{32} respectively. Rule R_{63} describes the calculation which occurs when a class is sealed by applying the *seal1st* operator. The outcome is a class in which a fresh program location containing the initial value v has been allocated for the first attribute i.

$$
\begin{aligned}
&(seal1st : (c(i : \tilde{i}, e_1, \tilde{j}, \tilde{k}, \tilde{b}), v) : s, e_2, @ : c, d, h) \longmapsto \\
&\quad (c(\tilde{i}, e_1 \oplus (i \mapsto \alpha), \tilde{j}, \tilde{e}) : s, e_2, c, d, h \oplus (\alpha \mapsto v)) \; R_{63}
\end{aligned}
$$

In general, a class has more than one attribute and we would like a method of sealing each attribute individually. This could be achieved using the following family of sealing operators:

let $seal1st(c, v) = c(v)$

let $seal2nd(c, v_2)(v_1) = c(v_1)(v_2)$

...

let $sealnth(c, v_n)(v_1)(v_2) \ldots (v_{n-1}) = c(v_1)(v_2) \ldots (v_{n-1})(v_n)$

Alternatively, a single seal function can be defined as follows:

let $seal(c, v, n) =$
  if $n = 1$
  then $c(v)$
  else $\lambda v'.seal(c(v'), v, n - 1)$

where $seal(c, v, n)$ will seal the n^{th} attribute of the class c. Using this mechanism, the sharing of program locations between the instances of various classes can be controlled in fine detail. The calculation rule which describes the application of the operator *seal* is R_{64} and shows that the n^{th} attribute is selected from the class, a fresh program location is initialised and allocated to it.

$$\frac{\#(\tilde{i}) = n - 1}{\begin{array}{c}(seal : (c(\tilde{i} +\!\!+ (i) +\!\!+ \tilde{j}, e_1, \tilde{l}, \tilde{k}, \tilde{b}), n, v) : s, e_2, @ : c, d, h) \longmapsto \\ (c(\tilde{i} +\!\!+ \tilde{j}, e_1 \oplus (i \mapsto \alpha), \tilde{l}, \tilde{k}, \tilde{b}) : s, e_2, c, d, h \oplus (\alpha \mapsto v))\end{array}} \; R_{64}$$

For example, suppose that c_1, c_2 and c_3 are class definitions. Instances of class c_1 do not share any program locations, instances of class c_1 all share the program location attribute named x but have different program locations for the attributes named y. Instances of class c_3 all share the same program location for attribute x and also share this location with instances of c_2. Instances of c_3 also share the same program location for attribute y but have different program locations for attribute z. This is described as follows:

$$\begin{array}{l}
\textbf{let } c_1 = \\
\quad \textbf{class } (x, y, z) \\
\quad\quad getx() = !x \\
\quad\quad gety() = !y \\
\quad\quad getz() = !z \\
\quad\quad setx(v) = x := v \\
\quad\quad sety(v) = y := v \\
\quad\quad setz(v) = z := v \\
\quad \textbf{end} \\
\\
\textbf{let } c_2 = seal(c_1, 10, 1) \\
\\
\textbf{let } c_3 = seal(c_2, 20, 1) \\
\\
\textbf{let } o_1 = new(c_1, (10, 20, 30)) \\
\\
\textbf{let } o_2 = new(c_2, (20, 30)) \\
\\
\textbf{let } o_3 = new(c_3, 30) \\
\\
send(o_1, \text{`}setx\text{'}, 20) \Rightarrow 20 \\
send(o_2, \text{`}getx\text{'}, ()) \Rightarrow 20 \\
send(o_3, \text{`}getx\text{'}, ()) \Rightarrow 20 \\
send(o_1, \text{`}sety\text{'}, 30) \Rightarrow 30 \\
send(o_2, \text{`}gety\text{'}, ()) \Rightarrow 20 \\
send(o_3, \text{`}gety\text{'}, ()) \Rightarrow 20 \\
send(o_2, \text{`}sety\text{'}, 40) \Rightarrow 30 \\
send(o_1, \text{`}gety\text{'}, ()) \Rightarrow 30 \\
send(o_3, \text{`}gety\text{'}, ()) \Rightarrow 40
\end{array}$$

In the Λ-calculus implementation objects are environments mapping method names to function closures. Each function closure contains a closure environment part of which is an environment, referred to as an *attribute environment*, mapping attribute names to program locations. All attribute environments in the methods of a given object are the same. The *seal* operator allows parts of attribute environments from different instances of the same class to be the same. The following are attribute environments from two methods in different instances of c_1:

$$('x' \mapsto \alpha_1) \oplus ('y' \mapsto \alpha_2) \oplus ('z' \mapsto \alpha_3)$$
$$('x' \mapsto \alpha_4) \oplus ('y' \mapsto \alpha_5) \oplus ('z' \mapsto \alpha_6)$$

and shows that, in general, two instances of c_1 will not share attribute locations. The following are attribute environments from two methods in different instances of c_2:

$$('x' \mapsto \alpha_7) \oplus ('y' \mapsto \alpha_8) \oplus ('z' \mapsto \alpha_9)$$
$$('x' \mapsto \alpha_7) \oplus ('y' \mapsto \alpha_{10}) \oplus ('z' \mapsto \alpha_{11})$$

and shows that, in general, two instances of c_2 will share the same program location for attribute x but will not share the y and z attribute locations. Finally, the following are attribute environments from two methods in different instances of c_3:

$$('x' \mapsto \alpha_7) \oplus ('y' \mapsto \alpha_{12}) \oplus ('z' \mapsto \alpha_{13})$$
$$('x' \mapsto \alpha_7) \oplus ('y' \mapsto \alpha_{12}) \oplus ('z' \mapsto \alpha_{14})$$

and shows that, in general, two instances of c_3 will share two program locations. The first is for attribute x and is also shared with all of the instances of c_2. The second is for attribute y, but is not shared with instances from any other class. The attribute z is bound to a different program location in every instance of c_3.

4.13 Feature 13: Classes are First Class Values

Feature 13 requires the framework to allow classes to be named, but also to allow classes to be first class data values which may be given different names in different parts of the program, passed to operations as arguments and returned from operations as results. This is a significant extension to the facilities for class manipulation which are provided by most object-oriented programming languages, which usually force the name of a class to be part of it and severely restrict classes occurring as program values.

All the implementations for classes given in this chapter allow classes to be first class data values. The Λ-calculus is therefore a flexible and convenient framework for experimenting with class-based object-oriented language features. A basic rule of most functional programming languages is that data values are named by λ-application and that a value does not contain its name as an intrinsic attribute. The following is a very simple example of a class being used as a first class data value. There is no implementation of this feature since it applies to all representations of classes given in this chapter.

$$\begin{array}{ll}
\textbf{let } c = & S_1 \\
\quad \textbf{class } (x) & \\
\quad\quad getx() = !x & \\
\quad\quad setx(v) = x := v & \\
\quad \textbf{end} & \\
& \\
\textbf{let } f(k) = new(k, 10) & S_2 \\
& \\
\textbf{let } o = f(c) & S_3
\end{array}$$

Line S_1 defines a class, see §4.11 for more details. Line S_2 defines a function f which is applied to a class k and instantiates the class supplying 10 as the attribute value. Line S_3 defines o, using f to instantiate c. As shown, the name c is not an intrinsic part of the class which is instantiated to produce o since the name k is used within the body of f. This trivial example would not be possible in many object-oriented languages due to the restrictions which are placed on expressions involving classes and the inclusion of a class' name as part of its intrinsic properties.

4.14 Feature 14: Classes may be Nested

Feature 14 requires the framework to allow classes to be nested to arbitrary depth. This is not simply a syntactic constraint, since the scoping rules for attributes must be observed when classes are returned from an inner scope. Few object-oriented programming languages let classes be nested, BETA is an exception to this, and when they do, there are often restrictions placed on identifier scopes, such as in C++. Consider the following specification for class nesting:

$$
\begin{array}{ll}
\textbf{let } c_1 = & S_1 \\
\quad \textbf{class } (x) & \\
\qquad c = \ \textbf{class } (y) & \\
\qquad\qquad \mathit{mult}() = !x * !y & \\
\qquad\qquad \mathit{setx}(v) = x := v & \\
\qquad\quad\ \textbf{end} & \\
\qquad \mathit{getx}() = !x & \\
\quad \textbf{end} & \\
& \\
\textbf{let } o_1 = \mathit{new}(c_1, 10) & S_2 \\
& \\
\textbf{let } c_2 = \mathit{ref}(o_1, \text{`}c\text{'}) & S_3 \\
& \\
\textbf{let } o_2 = \mathit{new}(c_2, 20) & S_4 \\
& \\
\mathit{send}(o_1, \text{`}\mathit{getx}\text{'}, ()) \Rightarrow 10 & S_5 \\
\mathit{send}(o_2, \text{`}\mathit{mult}\text{'}, ()) \Rightarrow 200 & S_6 \\
\mathit{send}(o_2, \text{`}\mathit{setx}\text{'}, 20) \Rightarrow 20 & S_7 \\
\mathit{send}(o_1, \text{`}\mathit{getx}\text{'}, ()) \Rightarrow 20 & S_8 \\
\mathit{send}(o_2, \text{`}\mathit{mult}\text{'}, ()) \Rightarrow 400 & S_9
\end{array}
$$

Line S_1 defines a class c_1 which has a single attribute named x, a method named *getx* and a field named c. The field value is a class which defines a single attribute named y and two methods named *mult* and *setx*. The specification requires that the scope of an attribute contains the bodies of all the methods and the initial values of all the fields in the same class. Both *mult* and *setx* refer to the identifier x which is an attribute of the enclosing class.

Line S_2 defines o_1 to be an instance of the class c_1. Line S_3 defines c_2 to be the class which is the value of the c field of o_1. The field reference operator *ref* returns the current value of a field given an object and a field name. Line S_4 creates an instance of c_2 called o_2.

Lines S_5 – S_9 show the outcomes of sending various messages to the objects o_1 and o_2. The messages show that the attribute x is scoped correctly over the bodies of the methods of c_2 and that an instance of c_2 shares the program location for x with the object o_1. In general we require each instance of c_1 to have a different class as the value of field c. This is necessary because the class c references the program location named x which is different for each instance of c_1.

Class nesting can be implemented in the Λ-calculus by representing classes as functions whose formal parameters are the attributes and which return environments binding names to fields and methods. The ordinary rules of lexical identifier scoping ensure that class nesting behaves as required:

$$
\begin{array}{ll}
\textbf{let } c_1(x) = \textbf{let } x = \mathit{loc}(x) \textbf{ in} & I_1 \\
\quad (\text{`}c\text{'} \mapsto \lambda y. & \\
\qquad \textbf{let } y = \mathit{loc}(y) & \\
\qquad \textbf{in } (\text{`}\mathit{mult}\text{'} \mapsto \lambda().!x{*}!y) \oplus & \\
\qquad\quad (\text{`}\mathit{setx}\text{'} \mapsto \lambda v.x := v)) \oplus & \\
\quad (\text{`}\mathit{getx}\text{'} \mapsto \lambda().!x) & \\
& \\
\textbf{let } \mathit{new}(c, v) = c(v) & I_2 \\
& \\
\textbf{let } \mathit{ref}(o, i) = o \bullet i & I_3 \\
& \\
\textbf{let } \mathit{send}(o, i, v) = o \bullet i(v) & I_4
\end{array}
$$

Line I_1 defines class c_1 as a function which is applied to an initial value for the attribute x and returns an environment mapping c to a class and *getx* to a function closure. Class c_1 allocates a fresh program location for x. The class which is named c is defined within the scope of the identifier x so the methods of the inner class may reference the attribute names of the outer class. Lines I_2 – I_4 define the instantiation, field reference and message passing operators as function application, environment lookup and environment lookup then application respectively.

In order to show that the implementation is correct with respect to the specification, the two objects o_1 and o_2 must be shown to have computational occurrences as follows:

$$
\begin{array}{l}
o_1 = (\text{`}c\text{'} \mapsto c_2) \oplus (\text{`}\mathit{getx}\text{'} \mapsto k_1) \\
o_2 = (\text{`}\mathit{mult}\text{'} \mapsto k_2) \oplus (\text{`}\mathit{setx}\text{'} \mapsto k_3) \\
c_2 = c(\text{`}y\text{'}, e \oplus e_1, (\text{`}\mathit{mult}\text{'}, \text{`}\mathit{setx}\text{'}), ((), \text{`}v\text{'}), (!x{*}!y, x := v)) \\
k_1 =< (), e \oplus e_1, !x > \\
k_2 =< (), e \oplus e_1 \oplus e_2, !x{*}!y > \\
k_3 =< \text{`}y\text{'}, e \oplus e_1 \oplus e_2, x := v > \\
e_1 = \text{`}x\text{'} \mapsto \alpha_1, e_2 = \text{`}y\text{'} \mapsto \alpha_2 \\
h_1 = h \oplus (\alpha_1 \mapsto 10), h_2 = h_1 \oplus (\alpha_2 \mapsto 20)
\end{array}
$$

The representations for class expressions, classes and the calculation rules for class creation, class instantiation, field reference and message passing are the same as those given in §4.12 and can be used to show that the outcomes of defining o_1 and o_2 and the outcomes of performing the expressions given in lines S_5 – S_9 are as required.

4.15 Feature 15: Classes with Single Inheritance

Feature 15 requires the framework to support classes with single inheritance. Many class based languages support single inheritance by allowing classes to be defined as an extension to an existing (or super-) class. The extension part defines additional attribute names and methods which are required by the instances of the new class. The attributes and methods which are defined by the super-class are included in the new class and any attribute names which are defined by the super-class are scoped over the methods of the new class. The following is a specification for simple single inheritance:

$$
\begin{array}{ll}
\textbf{let } c_1 = & S_1 \\
\quad \textbf{class } \mathit{nullclass}\ (x) & \\
\quad\quad \mathit{getx}() = !x & \\
\quad\quad \mathit{setx}(v) = x := v & \\
\quad \textbf{end} & \\
 & \\
\textbf{let } c_2 = & S_2 \\
\quad \textbf{class } c_1\ (y) & \\
\quad\quad \mathit{gety}() = !y & \\
\quad\quad \mathit{sety}(v) = y := v & \\
\quad\quad \mathit{sum}() = !x + !y & \\
\quad \textbf{end} & \\
 & \\
\textbf{let } o = \mathit{new}(c_2, (((),10),20)) & S_3 \\
 & \\
\mathit{send}(o, \text{`}\mathit{getx}\text{'}, ()) \Rightarrow 10 & S_4 \\
\mathit{send}(o, \text{`}\mathit{gety}\text{'}, ()) \Rightarrow 20 & S_5 \\
\mathit{send}(o, \text{`}\mathit{setx}\text{'}, 30) \Rightarrow 30 & S_6 \\
\mathit{send}(o, \text{`}\mathit{getx}\text{'}, ()) \Rightarrow 30 & S_7 \\
\mathit{send}(o, \text{`}\mathit{sety}\text{'}, 40) \Rightarrow 40 & S_8 \\
\mathit{send}(o, \text{`}\mathit{gety}\text{'}, ()) \Rightarrow 40 & S_9 \\
\mathit{send}(o, \text{`}\mathit{mult}\text{'}, ()) \Rightarrow 70 & S_{10}
\end{array}
$$

Line S_1 defines a class c_1 with a single attribute named x and two methods for accessing and setting the value of x. Class c_1 inherits from the built-in class *nullclass*. Each class definition must include a reference to another class from which it inherits after the keyword **class**. The inherited class is referred to as a *super-class* of the outcome of the definition which is referred to as a *sub-class* of the inherited class. The effect is to include all of the attribute and method definitions of the super-class into the sub-class definition.

Line S_2 defines a class c_2 which inherits from c_1 and defines a single attribute y and methods for accessing and updating the value of y. The method *sum* adds together the values of the two attributes of c_2 and returns the result. The identifier x is scoped over the bodies of all the methods defined by c_2 because it is inherited from the super-class c_1. Inheritance causes all of the attribute and method definitions from the super-class to be included into the sub-class definition. The definition of c_2 is equivalent to:

$$
\begin{array}{l}
\textbf{let } c_2 = \\
\quad \textbf{class } \mathit{nullclass}\ (x, y) \\
\qquad \mathit{getx}() = !x \\
\qquad \mathit{setx}(v) = x := v \\
\qquad \mathit{gety}() = !y \\
\qquad \mathit{sety}(v) = y := v \\
\qquad \mathit{sum}() = !x + !y \\
\quad \textbf{end}
\end{array}
$$

The scope of an identifier in a programming language determines where the identifier can be referenced within the program text. Standard examples of identifier scoping rules are lexical scoping and dynamic scoping where the former refers to the scope of an identifier being determined by the textual program block in which it occurs and the latter referring to the lifetime of the association between an identifier and its value during program execution.

The scoping of x in the above specification does not fit any of the traditional models of identifier scoping. The scope of an attribute identifier is associated with the class in which it is defined and any of its sub-classes. In this thesis this is referred to as *inheritance scoping.*

Line S_3 creates an instance o of c_2. The operator *new* is used to instantiate a class. It is applied to a pair containing a class and a collection of initial attribute values. The result is a new object. The initial attribute values must be supplied as a data value with a particular structure; this is described as follows. Let c_1 be a class which may be instantiated by $new(c, v)$ where v contains the initial attribute values for c. Any class c' which uses c as its direct super-class may be instantiated by $new(c', (v, v'))$ where v' is a value containing the initial values for the attributes which are defined by c' but not inherited from c *i.e.* the attribute extension. The initial values are supplied to a class as an n-tuple (where () and a single value are the degenerate cases). The empty class *nullclass* has no attributes and therefore is supplied with (). Line S_3 shows that the initial values are supplied as $(((), 10), 20)$ so that 20 is the initial value for y of c_2, 10 is the initial value for x of c_1 (via c_2) and () corresponds to no initial values for *nullclass* (via c_2 and c_1).

Lines S_4 – S_{10} show the outcomes of sending various messages to o. In particular these lines show that c_2 inherits all of the methods of c_1 and the attribute x.

Two key features of single inheritance are the aggregation of definitions from the super-class and the extension, and the inheritance scoping rules for inherited attributes. In the Λ-calculus, classes can be implemented as functions from initial attribute values to environments of methods. The former feature can be implemented conveniently as the concatenation of environments. The latter feature is inconsistent with the conventional scoping rules associated with the λ-calculus since it involves arbitrarily extending the scope of function parameter identifiers over other function bodies. We wish to implement classes as a combination of two functions: the super-class and the extension. Both functions can be defined separately and then combined using a binary inheritance operator. The extension function may contain free references to identifiers which are attributes of the super-class and which are therefore parameters of (part of) the super-class. Under normal scoping rules, the scope of an identifier is just the body of the function which introduces it. This implies that in order to implement inheritance scoping we require primitive operations which allow the scope of an identifier to be extended from the body of one function to another, *i.e.* from the body of the super-class to the body of the extension. For example, suppose that *inherit* is a binary operator which is applied to a pair of functions (c, x) where c is a (super-) class and x is an extension. Both a class and an extension are functions from attribute values to method environments. Suppose also that there is a supplied super-class *nullclass* which defines no methods or attributes. A simple two-dimensional point class is defined:

$$\begin{array}{l}
\textbf{let } \textit{pointextension}(x, y) = \\
\quad \textbf{let } x = \textit{loc}(x) \\
\quad\quad\;\; y = \textit{loc}(y) \\
\quad \textbf{in } \;(\text{`}\textit{getx}\text{'} \mapsto \lambda().!x) \oplus \\
\quad\quad\;\; (\text{`}\textit{gety}\text{'} \mapsto \lambda().!y) \oplus \\
\quad\quad\;\; (\text{`}\textit{move}\text{'} \mapsto \lambda(x', y').(x := x'); (y := y'))
\end{array}$$

$$\textbf{let } \textit{point} = \textit{inherit}(\textit{nullclass}, \textit{pointextension})$$

Now a three-dimensional point class can be defined by inheriting the two-dimensional point class. The extension for three-dimensional points defines an attribute z, an accessing method *getz* and redefines the *move* method to incorporate z. The new *move* method can assume that the attributes x and y are in scope due to inheritance scoping:

$$\begin{array}{l}
\textbf{let } \textit{3dpointextension}(z) = \\
\quad \textbf{let } z = \textit{loc}(z) \\
\quad \textbf{in } \;(\text{`}\textit{getz}\text{'} \mapsto \lambda().!z)) \oplus \\
\quad\quad\;\; (\text{`}\textit{move}\text{'} \mapsto \lambda(x', y', z').(x := x'); (y := y'); (z := z'))
\end{array}$$

$$\textbf{let } \textit{3dpoint} = \textit{inherit}(\textit{point}, \textit{3dpointextension})$$

The implementation of the single inheritance operator *inherit* must arrange for the attribute values to be supplied to the correct extension functions, for the method environments which are returned by each extension function to be concatenated in the correct order and for inheritance scoping of the attribute identifiers to be performed.

The Λ-calculus supports first class environments which allows the current machine environment to be *reified* and *installed*. This feature allows the programmer to control the identifiers which are currently in scope. It is possible to query which identifiers are in scope by reifying the current environment and accessing its bindings and it is possible to modify the identifiers which are in scope by constructing an environment and installing it. Arbitrary use of this programming feature will produce programs which are very difficult to understand and which are inefficient. However we propose that this feature is fundamental to the behaviour of object-oriented programming languages and that its use can be packaged up in related ways which exhibit a wide variety of inheritance programming features. The implementation of classes with single inheritance and the classes specified in lines S_1 and S_2 are as follows:

$$\textbf{let } \mathit{nullclass}() = (\{\}, \{\}) \qquad I_1$$

$$\begin{array}{l} \textbf{let } \mathit{inherit}(c, x)(v_1, v_2) = \\ \quad \textbf{let } (m_1, e_1) = c(v_1) \textbf{ in} \\ \quad \textbf{let } (m_2, e_2) = I(R(x) \oplus e_1, x)(v_2) \\ \quad \textbf{in } (m_1 \oplus m_2, e_1 \oplus e_2) \end{array} \qquad I_2$$

$$\textbf{let } \mathit{new}(c, v) = \mathit{1st}(c(v)) \qquad I_3$$

$$\textbf{let } \mathit{send}(o, i, v) = o \bullet i(v) \qquad I_4$$

$$\begin{array}{l} \textbf{let } x_1(x) = \\ \quad \textbf{let } x = \mathit{loc}(x) \textbf{ in} \\ \quad \textbf{let } e = R(\lambda().()) - R(x_1) \\ \qquad m = (\text{`}\mathit{getx}\text{'} \mapsto \lambda().!x) \oplus (\text{`}\mathit{setx}\text{'} \mapsto \lambda v.x := v) \\ \quad \textbf{in } (m, e) \end{array} \qquad I_5$$

$$\textbf{let } c_1 = \mathit{inherit}(\mathit{nullclass}, x_1) \qquad I_6$$

$$\begin{array}{l} \textbf{let } x_2(y) = \\ \quad \textbf{let } y = \mathit{loc}(y) \textbf{ in} \\ \quad \textbf{let } e = R(\lambda().()) - R(x_2) \\ \qquad m = \begin{array}{l} (\text{`}\mathit{gety}\text{'} \mapsto \lambda().!y) \oplus \\ (\text{`}\mathit{sety}\text{'} \mapsto \lambda v.y := v) \oplus \\ (\text{`}\mathit{sum}\text{'} \mapsto \lambda().!x+!y) \end{array} \\ \quad \textbf{in } (m, e) \end{array} \qquad I_7$$

$$\textbf{let } c_2 = \mathit{inherit}(\mathit{nullclass}, x_2) \qquad I_8$$

Lines I_1 – I_4 define the operators which are used in single inheritance. Lines I_5 – I_8 define the specified classes c_1 and c_2 and give examples of the particular format used for extension functions.

Both classes and extensions are implemented as functions which are applied to attribute values and which return a pair (m, e). The first component of the pair is an environment binding method names to function closures. The second component of the pair is an environment binding attribute names to attribute values. The m component is ultimately returned as the object created by instantiating a class. The e component is returned in order to implement inheritance scoping, *i.e.* inheriting extension functions can install e in order to place inherited attributes in scope. Ultimately, the e component is discarded when instantiating a class, only the m component is returned.

Line I_1 defines the empty class *nullclass*. This is applied to the empty collection of attribute values () and returns the empty (m, e) pair $(\{\}, \{\})$.

Line I_2 defines the *inherit* operator which is applied to a pair (c, x) containing a class c and an extension function x and returns a class. The class which is returned is applied to a pair (v_1, v_2) containing the attribute values v_1 for c and the attribute values v_2 for x and returns a pair (m, e) which is constructed as follows. The super-class c is instantiated by applying it to the attribute values v_1. This produces a pair (m_1, e_1) containing the method and attribute environments produced by instantiating the super-class. In order to implement inheritance scoping, the attribute environment produced by the super-class must be scoped over the methods produced by the extension x. This is achieved by installing the attribute environment e_1 into the extension function x using the expression $I(R(x) \oplus e_1, x)$ *i.e.* the closure environment of x is reified $R(x)$, extended with the attribute environment $R(x) \oplus e_1$ and then replaced by installing it back $I(R(x) \oplus e_1, x)$. This produces a new extension which is applied to the attribute values v_2 in order to produce a pair (m_2, e_2) containing the method and attribute environment produced by the extension. The final pair produced by instantiating the class $\mathit{inherit}(c, x)$ is the result of concatenating both the method environments and attribute environments produced by instantiating c and x.

Line I_3 defines the class instantiation operator. Classes are functions which are applied to attribute values and produce pairs of environments. The instance produced by instantiating a class is the first component of such pairs.

Line I_4 defines the message passing operator which simply looks up a function closure in an object and applies it to the message parameter.

When an extension function is applied to attribute values it will bind each of the attribute names to a fresh program location containing the corresponding value. The resulting environment is referred to as the attribute environment of the extension. Each of the methods defined by the extension are bound to the corresponding method names, each of the attributes are scoped over the method bodies so the attribute environment is associated with each method. The return value of the extension function is a pair containing the method environment and the attribute environment.

Lines I_5 and I_7 show particular instances of extension functions. An extension function defines attributes and methods which will be added to an existing class using the *inherit* operator. Extension functions are intended to be used only in conjunction with a super-class since they may contain free references to identifiers which are attributes supplied by the super-class. An extension function has the general form:

$$
\begin{array}{ll}
X_1 & \textbf{let } x(i_1, \ldots, i_n) = \\
X_2 & \quad \textbf{let } (i_1, \ldots, i_n) = (\mathit{loc}(i_1), \ldots, \mathit{loc}(i_n)) \textbf{ in} \\
X_3 & \quad \textbf{let } e = R(\lambda().()) - R(x) \\
X_4 & \qquad m = (m_1 \mapsto f_1) \oplus \ldots \oplus (m_k \mapsto f_k) \\
X_5 & \quad \textbf{in } (m, e)
\end{array}
$$

where $i_1 \ldots i_n$ are the attribute names, $m_1 \ldots, m_k$ are the method names and $f_1 \ldots, f_k$ are λ-expressions which define the methods. The identifiers x, e and m are newly coined and are not referenced elsewhere in the program. The behaviour of such a function, when applied to attribute values is as follows. At X_1 the initial attribute values are bound to the attribute names. At X_2 each of the attribute names are rebound to fresh program locations containing the initial values. At X_3 the attribute environment e is constructed by reifying the current environment $R(\lambda().())$ and subtracting the environment which was current when x was defined $R(\lambda().()) - R(x)$. This has the effect of producing only those bindings in the current environment which have been created inside the function x, *i.e.* the attribute environment. At X_4 the method environment is constructed by binding each method name to a function closure. The attribute environment is part of the current environment at this point and will be captured as part of each closure environment so the attributes are correctly scoped over each method. Finally at X_5 the pair of environments are returned.

In order to show that the implementation using the Λ-calculus is correct with respect to the specification of the language feature we use a representation for computational occurrences of classes and extension functions and use calculation rules to describe the outcomes of class instantiation. A computational occurrence of an extension function is represented as $x(\tilde{i}, e, \tilde{j}, \tilde{k}, \tilde{b})$ where $\tilde{i}$ is a sequence of attribute names, e is an environment which maps free references of identifiers to their values within the body of the extension function. Notice that references to attribute names within the body of the extension function will not be free since they are captured by the rules of inheritance scoping. $\tilde{j}$ is a sequence of method names, $\tilde{k}$ are method arguments and $\tilde{b}$ are method bodies.

A computational occurrence of a class is either *nullclass* when it is the empty class or $c(k, x)$ when the class has been constructed using the operator *inherit*. The value k is a class (the super-class of $c(k, x)$) and x is an extension function.

The rules describing the calculations which occur when classes are instantiated are described as follows. R_{65} describes the calculation which occurs when the empty class is instantiated. The outcome is a pair of empty environments:

$$(\mathit{nullclass} : () : s, e, @ : c, d, h) \longmapsto ((\{\}, \{\}) : s, e, c, d, h) \;\; R_{65}$$

R_{66} describes the calculations which occur when an extension function is applied to a tuple of values $\tilde{v}$. A collection of fresh program locations are allocated for the attributes $\tilde{i}$ of the extension. Each method definition is performed in a context in which the extension environment e_1 has been extended with bindings for each of the attributes $\tilde{i} \mapsto \tilde{\alpha}$. The outcome is a pair of environments containing the attributes and the methods:

$$\begin{array}{l}(x(\tilde{i}, e_1, \tilde{j}, \tilde{k}, \tilde{b}) : \tilde{v} : s, e_2, @ : c, d, h) \longmapsto \\ \quad ((\tilde{j} \mapsto <\tilde{k}, e_1 \oplus (\tilde{i} \mapsto \tilde{\alpha}), \tilde{b}>, \tilde{i} \mapsto \tilde{\alpha}) : s, e_2, c, d, h \oplus (\tilde{\alpha} \mapsto \tilde{v})) \;\; R_{66}\end{array}$$

R_{67} describes the calculations which occur when a class constructed using the operator *inherit* is instantiated. The super-class k is instantiated first and produces a pair (m_1, e_1). The rules of inheritance scoping are implemented by extending the extension environment e_3 with the attribute environment from the super-class e_1. The new extension function $x(\tilde{i}, e_3 \oplus e_1, \tilde{j}, \tilde{k}, \tilde{b})$ is instantiated to produce a pair of environments (m_2, e_2). The outcome of the entire calculation is the merged pair $(m_1 \oplus m_2, e_1 \oplus e_2)$:

$$\frac{\begin{array}{c}(k : v : s, e, @ : c, d, h_1) \longmapsto ((m_1, e_1) : s, e, c, d, h_2) \\ (x(\tilde{i}, e_3 \oplus e_1, \tilde{j}, \tilde{k}, \tilde{b}) : \tilde{v} : s, e, @ : c, d, h_2) \longmapsto ((m_2, e_2) : s, e, c, d, h_3)\end{array}}{(c(k, x(\tilde{i}, e_3, \tilde{j}, \tilde{k}, \tilde{b}) : (v, \tilde{v}) : s, e, @ : c, d, h_1) \longmapsto ((m_1 \oplus m_2, e_1 \oplus e_2) : s, e, c, d, h_3)} \;\; R_{67}$$

This section has described a basic mechanism for class based single inheritance. It has implemented classes as functions which produce environments and has used environment reification and installation to implement inheritance scoping which occurs when a new class is defined. Object-oriented languages differ with respect to the details of single inheritance and we claim that the mechanism in this section can be modified and extended to represent all such variations. Evidence in support of this claim is supplied in subsequent sections of this chapter.

4.16 Feature 16: Late Binding

Feature 16 requires the framework to support a "late-binding" mechanism. Late binding occurs when the mechanism which associates identifiers and their values takes account of subsequent redefinition. Most non-object-oriented languages employ early binding: redeclaration of an identifier in a nested block will not affect references to the identifier in textually enclosing blocks. C++ is an example of an object-oriented language which supports late binding via the `virtual` function member declaration.

Object-oriented languages seek to promote reuse by allowing redefinition of an identifier in a sub-class to affect references to the identifier in super-class methods. For example, a sub-class may reimplement a method to be more efficient and we would like all the uses of the method by the super-class to benefit from the improved implementation. The following is a simple specification for late binding:

$$
\begin{array}{ll}
\textbf{let } c_1 = & S_1 \\
\quad \textbf{class } \mathit{nullclass}\ (x) & \\
\quad\quad y = 1 & \\
\quad\quad f() = !x * y & \\
\quad\quad g() = z & \\
\quad \textbf{end} & \\
& \\
\textbf{let } c_2 = & S_2 \\
\quad \textbf{class } c_1\ () & \\
\quad\quad y = 2 & \\
\quad\quad z = 3 & \\
\quad \textbf{end} & \\
& \\
\textbf{let } o = \mathit{new}(c_2, (((), 10), ())) & S_3 \\
& \\
\mathit{send}(o, `f', ()) \Rightarrow 20 & S_4 \\
\mathit{send}(o, `g', ()) \Rightarrow \epsilon & S_5
\end{array}
$$

Line S_1 defines a class c_1 with a single attribute x, a field y and two methods f and g. Methods in classes may refer to any field of the class without further qualification. The method f in c_1 refers to x and y. Method g refers to an identifier z which we assume is not in scope when c_1 is defined.

Line S_2 defines a class c_2 which inherits from c_1, redefines the field y and defines a new field z. Line S_3 instantiates c_2 to produce an object o. The attribute x is initialised to 10.

Line S_4 shows that fields in classes are bound using a late binding scheme. This means that the reference to identifier y in the inherited method f refers to the most recently defined field named y, *i.e.* 2 in c_2 rather than 1 in c_1.

Line S_5 shows that inherited methods do not have access to all the fields which are subsequently defined in sub-classes. When the method g refers to the identifier z the result is ϵ. Even though o contains a field named z, this was not a field or attribute of c_1 so methods inherited from c_1 cannot access this field.

The Λ-calculus provides access to identifier environments which may be installed in function closures in order to override existing bindings. This facility is used in order to implement late binding as follows:

$$\textbf{let } \mathit{nullclass}()(\_) = (\{\},\{\}) \qquad I_1$$

$$\begin{array}{ll}\textbf{let } \mathit{inherit}(c,x)(v_1,v_2)(o) = & I_2\\ \quad \textbf{let } (m_1,e_1) = c(v_1)(o\backslash(\mathit{names}(c))) \textbf{ in} & \\ \quad \textbf{let } (m_2,e_2) = I(R(x)\oplus e_1 \oplus o, x)(v_2) & \\ \quad \textbf{in } (m_1 \oplus m_2, e_1 \oplus e_2) & \end{array}$$

$$\textbf{let } \mathit{new}(c,v) = \mathit{1st}(Y(c(v))) \qquad I_3$$

$$\begin{array}{ll}\textbf{let } x_1(x) = & I_4\\ \quad \textbf{let } x = \mathit{loc}(x) \textbf{ in} & \\ \quad \textbf{let } e = R(\lambda().()) - R(x_1) & \\ \qquad m = (\text{`}y\text{'} \mapsto 1) \oplus (\text{`}f\text{'} \mapsto \lambda().!x * y) & \\ \quad \textbf{in } \ (m,e) & \end{array}$$

$$\textbf{let } c_1 = \mathit{inherit}(\mathit{nullclass}, c_1) \qquad I_5$$

$$\textbf{let } x_2() = (\text{`}y\text{'} \mapsto 2, \{\}) \qquad I_6$$

$$\textbf{let } c_2 = \mathit{inherit}(c_1, x_2) \qquad I_7$$

A class is implemented as a function c which is applied to a collection of initial attribute values v and an object o: $c(v)(o)$. The result of the application is a pair (o, e) where o is an object and e is an attribute environment. The object which is supplied to the class is the same as the first component of the result, it is found using Y: $\mathit{1st}(Y(c(v)))$.

Line I_1 defines the empty class *nullclass* which return a pair of empty environments. Line I_2 defines the operator *inherit* which combines an extension function x and a super-class c to produce a new class. Late binding is implemented by extending the closure environment of x with the environment o. This has the effect of placing the most recent bindings for the object's fields in scope for the definition of the methods in x. In order that only the fields of a class are placed in scope, the object o is restricted when it is supplied to the super-class c:

$$c(v_1)(o\backslash(\mathit{names}(c)))$$

where *names* is a built-in operator with the following definition:

$$\mathit{names}(x(\_,\_,\tilde{j},\_,\_)) = \{\tilde{j}\}$$

$$\mathit{names}(c(k,x)) = \mathit{names}(k) \cup \mathit{names}(x)$$

$$\mathit{names}(\mathit{nullclass}) = \emptyset$$

Line I_3 defines the class instantiation operator *new* which finds the object o using a fixed point as described above. Lines I_4 – I_7 define the classes c_1 and c_2.

The implementation is correct with respect to the specification when the object o is the following computational value and the outcomes of the messages are those defined in the specification:

$$o = o_1 \oplus o_2$$

$$o_1 = (\text{`}y\text{'} \mapsto 1) \oplus (\text{`}f\text{'} \mapsto \lt(), e \oplus e_1, !x * y\gt) \oplus (\text{`}g\text{'} \mapsto \lt(), e \oplus e_1, z\gt)$$

$$o_2 = (\text{`}y\text{'} \mapsto 2) \oplus (\text{`}z\text{'} \mapsto 3)$$

$$e_1 = (\text{`}x\text{'} \mapsto \alpha) \oplus (o \backslash \{\text{`}y\text{'}, \text{`}f\text{'}, \text{`}g\text{'}\})$$

$$h_1 = h \oplus (\alpha \mapsto 10)$$

Class expressions, classes and extensions are represented as described as in §4.15. The calculation rules for class instantiation are described as follows.

$$(\mathit{nullclass} : () : \_ : s, e, @ : @ : c, d, h) \longmapsto ((\{\}, \{\}) : s, e, c, d, h) \;\; R_{69}$$

Rule R_{69} shows the calculation which occurs when the empty class is instantiated. Rule R_{66} describes the calculation which occurs when an extension function is instantiated.

$$\frac{\begin{array}{c}(c : v_1 : (o \backslash (\mathit{names}(c))) : s, e, @ : @ : c, d, h_1) \longmapsto ((m_1, e_1) : s, e, c, d, h_2) \\ (x(\tilde{i}, e_3 \oplus e_1 \oplus o, \tilde{j}, \tilde{k}, \tilde{b}) : v_2 : s, e, @ : c, d, h_2) \longmapsto ((m_2, e_2) : s, e, c, d, h_3)\end{array}}{(k(c, x(\tilde{i}, e_3, \tilde{j}, \tilde{k}, \tilde{b}) : (v_1, v_2) : o : s, e, @; @; c, d, h_1) \longmapsto ((m_1 \oplus m_2, e_1 \oplus e_2) : s, e, c, d, h_3)} \;\; R_{70}$$

Rule R_{70} describes the calculation which occurs when a class constructed using the *inherit* operator is instantiated.

$$\frac{(c : v : o : s, e, @ : @ : c, d, h_1) \longmapsto ((o, \_) : s, e, c, d, h_2)}{(\mathit{new} : (c, v) : s, e, @ : c, d, h_1) \longmapsto (o : s, e, c, d, h_2)} \;\; R_{71}$$

Rule R_{71} describes the calculation which occurs when a class is instantiated using the operator *new*.

4.17 Feature 17: The Run Super Mechanism

Feature 17 requires the framework to support a "run-super" mechanism which allows programmers to circumvent the shadowing of method names which occurs as a result of many inheritance strategies. When a new class is constructed by inheriting from an existing class, it is often the case that a number of the inherited methods must be redefined to take into account the new attributes and methods which are offered by the new class. Redefinition occurs when a new class defines a method with the same name as an inherited method. Often, it is useful to be able to use the old method as part of the new definition, for example initialisation methods will want to set the values of inherited attributes (performed by the inherited initialisation method) in addition to the new attributes.

The run-super mechanism provides the programmer with a facility for referencing redefined methods. This facility can take a number of different forms, for example in Smalltalk (which first introduced the facility), all the inherited methods are referenced using an object named `super`, whereas in CLOS each method may reference only the individual method which it redefined using the `call-next-method` facility. A specification of a Smalltalk-like language feature is as follows:

$$
\begin{array}{ll}
\textbf{let } c_1 = & S_1 \\
\quad \textbf{class } \mathit{nullclass}\ (x) & \\
\quad\quad v = 1 & \\
\quad\quad \mathit{reset}() = x := 0 & \\
\quad\quad \mathit{getx}() = !x & \\
\quad \textbf{end} & \\
 & \\
\textbf{let } c_2 = & S_2 \\
\quad \textbf{class } c_1\ (y) & \\
\quad\quad v = 2 & \\
\quad\quad \mathit{reset}() = \mathit{send}(\mathit{super}, `\mathit{reset}', ()); y := 0 & \\
\quad\quad \mathit{gety}() = !y & \\
\quad \textbf{end} & \\
 & \\
\textbf{let } o = \mathit{new}(c_2, (((), 10), 20)) & S_3 \\
 & \\
\mathit{ref}(o, `v') \Rightarrow 2 & S_4 \\
\mathit{send}(o, `\mathit{getx}', ()) \Rightarrow 10 & S_5 \\
\mathit{send}(o, `\mathit{gety}', ()) \Rightarrow 20 & S_6 \\
\mathit{send}(o, `\mathit{reset}', ()) \Rightarrow 0 & S_7 \\
\mathit{send}(o, `\mathit{getx}', ()) \Rightarrow 0 & S_8 \\
\mathit{send}(o, `\mathit{gety}', ()) \Rightarrow 0 & S_9
\end{array}
$$

Lines S_1 and S_2 define classes c_1 and c_2, line S_3 creates an instance o of class c_2 and lines S_4 – S_5 show the outcomes of referencing an attribute and sending various messages to o.

Class c_1 inherits from the empty class, defines a single attribute x, a field v and two methods *getx* and *reset*. Class c_2 inherits from c_1 and therefore extends the attribute, field and methods with a new attribute y and a new method *gety*. The inherited field v and the method *reset* are redefined by c_2. The redefined field is given a new value and the redefined method is given a new implementation.

The new implementation of *reset* in c_2 uses the identifier *super* to refer to the field and methods which are contributed by the class c_1. We can view method lookup as a process: when a message is sent to an object a search is initiated for the method with the message name. The search for the method starts in the class of which the object is an instance; if this class does not define a method with the required name then the search continues with the super-class. If the search process encounters the empty class then an error is signaled otherwise, the process must have encountered a method with the required name in a particular class. This method is invoked by creating a context in which the object's attributes are bound to their values and the method's parameters are bound to the values supplied in the message.

During execution of a method, sending a message to *super* causes the search process to continue where it left off. This mechanism allows redefined methods to be invoked since the search will continue in the super-class and thus encounter an otherwise shadowed method. This occurs in the definition of *reset* in c_2 where a *reset* message is sent to *super*, causing the attribute x to be reset, before the attribute y is reset.

Line S_4 shows that the field value v defined by c_2 is shadowed by the redefinition in c_2. This principle also applies to methods. Lines S_5 – S_9 specifies that the run-super mechanism must allow shadowed methods to be invoked.

The Λ-calculus implementation which is described for classes and inheritance in §4.15 creates intermediate environments which contain all super-class definitions. When these environments are merged with those produced by instantiating an extension function shadowing can occur. Access is easily provided to the shadowed bindings by supplying the environment of methods produced by instantiating the super-class to the extension function, as an argument when it is instantiated. The implementation is as follows:

$$\textbf{let } \mathit{nullclass}() = (\{\},\{\}) \qquad I_1$$

$$\begin{aligned}
&\textbf{let } \mathit{inherit}(c,x)(v_1,v_2) = && I_2\\
&\qquad \textbf{let } (m_1,e_1) = c(v_1) \textbf{ in}\\
&\qquad \textbf{let } (m_2,e_2) = I(R(x(m_1)) \oplus e_1, x(m_1))(v_2)\\
&\qquad \textbf{in } (m_1 \oplus m_2, e_1 \oplus e_2)
\end{aligned}$$

$$\textbf{let } \mathit{new}(c,v) = \mathit{1st}(c(v)) \qquad I_3$$

$$\textbf{let } \mathit{send}(o,i,v) = o \bullet i(v) \qquad I_4$$

$$\begin{aligned}
&\textbf{let } x_1(\mathit{super}) = Y\lambda x_1.\lambda x. && I_5\\
&\quad \textbf{let } x = \mathit{loc}(x) \textbf{ in}\\
&\quad \textbf{let } e = R(\lambda().()) - R(x_1)\\
&\qquad m = (\text{‘}\mathit{getx}\text{’} \mapsto \lambda().!x) \oplus (\text{‘}\mathit{reset}\text{’} \mapsto \lambda().x := 0)\\
&\quad \textbf{in } (m,e)
\end{aligned}$$

$$\textbf{let } c_1 = \mathit{inherit}(\mathit{nullclass}, x_1) \qquad I_6$$

$$\begin{aligned}
&\textbf{let } x_2(\mathit{super}) = Y\lambda x_2.\lambda y. && I_7\\
&\quad \textbf{let } y = \mathit{loc}(y) \textbf{ in}\\
&\quad \textbf{let } e = R(\lambda().()) - R(x_2)\\
&\qquad m = \ (\text{‘}\mathit{gety}\text{’} \mapsto \lambda().!y) \oplus\\
&\qquad\qquad (\text{‘}\mathit{reset}\text{’} \mapsto \lambda().\mathit{send}(\mathit{super}, \text{‘}\mathit{reset}\text{’}, ()); y := 0)\\
&\quad \textbf{in } (m,e)
\end{aligned}$$

$$\textbf{let } c_2 = \mathit{inherit}(\mathit{nullclass}, x_2) \qquad I_8$$

Lines I_1 – I_4 define the operators for inheritance, message passing and class instantiation. These are essentially the same as the definitions given in §4.15. The difference is shown in the definition of the *inherit* operator where the method environment m_1 is supplied to the extension function x as the value of the identifier *super*.

Lines I_5 and I_7 show the definition of the extension functions which are used in the definitions of the classes c_1 and c_2 in lines I_6 and I_8. Each extension function has the general form:

$$\begin{array}{ll} X_1 & \textbf{let } x(\mathit{super}) = Y\lambda x.\lambda(i_1,\ldots,i_n). \\ X_2 & \quad \textbf{let } (i_1,\ldots,i_n) = (\mathit{loc}(i_1),\ldots,\mathit{loc}(i_n)) \textbf{ in} \\ X_3 & \quad \textbf{let } e = R(\lambda().()) - R(x) \\ X_4 & \qquad m = (m_1 \mapsto f_1) \oplus \ldots \oplus (m_k \mapsto f_k) \\ X_5 & \quad \textbf{in } (m,e) \end{array}$$

where the extension is the same as that described in §4.15 except that the function x takes an extra parameter *super* which is an environment of methods, possibly containing definitions which are redefined by the extension.

In order to show that the implementation is correct with respect to the specification we define a representation for computational occurrences of classes, define calculation rules describing the computations which occur during class instantiation and show that the outcome of performing instantiation S_3 is:

$$\begin{array}{l} o = o_1 \oplus o_2 \\ o_1 = (\text{`}v\text{'} \mapsto 1) \oplus (\text{`}\mathit{reset}\text{'} \mapsto k_1) \oplus (\text{`}\mathit{getx}\text{'} \mapsto k_2) \\ o_2 = (\text{`}v\text{'} \mapsto 2) \oplus (\text{`}\mathit{reset}\text{'} \mapsto k_3) \oplus (\text{`}\mathit{gety}\text{'} \mapsto k_4) \\ k_1 =< (), e \oplus \text{`}\mathit{super}\text{'} \mapsto \{\} \oplus e_1, x := 0 > \\ k_2 =< (), e \oplus \text{`}\mathit{super}\text{'} \mapsto \{\} \oplus e_1, !x > \\ k_3 =< (), e \oplus \text{`}\mathit{super}\text{'} \mapsto o_1 \oplus e_1 \oplus e_2, \mathit{send}(\mathit{super}, \text{`}\mathit{reset}\text{'}, ()); y := 0 > \\ k_3 =< (), e \oplus \text{`}\mathit{super}\text{'} \mapsto o_1 \oplus e_1 \oplus e_2, !y > \\ e_1 = \text{`}x\text{'} \mapsto \alpha_1 \\ e_2 = \text{`}y\text{'} \mapsto \alpha_2 \end{array}$$

Classes and extension functions exist as computational values whose representation is defined in §4.15. The calculation rules for class instantiation using the run-super feature are R_{65}, R_{66} and R_{72} which is the same as R_{67} except that the identifier *super* is bound when the extension function is instantiated:

$$\frac{\begin{array}{c}(k : v : s, e, @ : c, d, h_1) \longmapsto ((m_1, e_1) : s, e, c, d, h_2) \\ (x(\tilde{i}, e_3 \oplus (\text{`}\mathit{super}\text{'} \mapsto m_1) \oplus e_1, \tilde{j}, \tilde{k}, \tilde{b}) : \tilde{v} : s, e, @ : c, d, h_2) \longmapsto ((m_2, e_2) : s, e, c, d, h_3)\end{array}}{(c(k, x(\tilde{i}, e_3, \tilde{j}, \tilde{k}, \tilde{b}) : (v, \tilde{v}) : s, e, @ : c, d, h_1) \longmapsto ((m_1 \oplus m_2, e_1 \oplus e_2) : s, e, c, d, h_3)} R_{72}$$

4.18 Feature 18: The Run Inner Mechanism

Feature 18 requires the framework to support the "run-inner" mechanism which occurs in the language BETA. Using the "run-super" feature which is described in §4.17, a new class may redefine any of the methods which it inherits and the new definition replaces the old; the run-super mechanism allows the original definition to be referenced using an identifier which is scoped over all of the new method bodies. The "run-inner" mechanism is almost exactly the opposite where super-class methods are invoked in preference to sub-class methods with the same name. The super-class methods may choose to invoke any redefinitions in sub-classes via an identifier which is scoped over the super-class method bodies.

When a message is sent to an instance of a class c the search for a method with the message name starts in the empty class. If no suitable method is found in a class c' the search process continues with that sub-class of c' from which c inherits, *i.e.* the search process moves from the root of the inheritance hierarchy *down* to c rather than from c to the root of the inheritance hierarchy. This is in direct contrast with the run-super feature described in §4.17.

The search process fails when it reaches c and c contains no definition of the appropriate method. Otherwise, the search process finds a method with the required name in some class c' which is a super-class of c. The method is invoked in a context in which the object's attributes are bound to their values and the method parameters are bound to the message values. During method activation, if a message is sent to *inner*, the search for the appropriate method continues where the search left off, *i.e.* with the appropriate sub-class of c'.

Using the run-inner feature, super-class definitions shadow sub-class definitions and shadowed methods can be invoked via the value of *inner*. A specification for a simple run-inner feature is as follows:

$$
\begin{array}{ll}
\textbf{let } c_1 = & S_1 \\
\quad \textbf{class } \mathit{nullclass}\ (x) & \\
\qquad v = 1 & \\
\qquad \mathit{reset}() = x := 0; \mathit{send}(\mathit{inner}, \text{‘}\mathit{reset}\text{’}, ()) & \\
\qquad \mathit{getx}() = !x & \\
\quad \textbf{end} & \\
& \\
\textbf{let } c_2 = & S_2 \\
\quad \textbf{class } c_1\ (y) & \\
\qquad v = 2 & \\
\qquad \mathit{reset}() = y := 0 & \\
\qquad \mathit{gety}() = !y & \\
\quad \textbf{end} & \\
& \\
\textbf{let } o = \mathit{new}(c_2, (((), 10), 20)) & S_3
\end{array}
$$

$$
\begin{array}{lll}
\mathit{ref}(o, \text{‘}v\text{’}) \Rightarrow 1 & & S_4 \\
\mathit{send}(o, \text{‘}\mathit{getx}\text{’}, ()) \Rightarrow 10 & & S_5 \\
\mathit{send}(o, \text{‘}\mathit{gety}\text{’}, ()) \Rightarrow 20 & & S_6 \\
\mathit{send}(o, \text{‘}\mathit{reset}\text{’}, ()) \Rightarrow 0 & & S_7 \\
\mathit{send}(o, \text{‘}\mathit{getx}\text{’}, ()) \Rightarrow 0 & & S_8 \\
\mathit{send}(o, \text{‘}\mathit{gety}\text{’}, ()) \Rightarrow 0 & & S_9
\end{array}
$$

The specification should be contrasted with that given in §4.17. The differences are that the method *reset* which is defined by class c_1 sends a message to *inner* in order to invoke any extension which a sub-class has added. The definition of *reset* in class c_2 is an extension to the definition of *reset* in c_1, it does not shadow the definition from the super-class.

Line S_4 shows that super-class definitions shadow sub-class definitions when using the run-inner feature. Lines S_5 – S_9 specify that methods may be extended in sub-classes and the extensions are invoked via *inner* from the super-class.

It should be stressed that the rules of inheritance scoping still apply, *i.e.* the names of super-class attributes are still scoped over the bodies of sub-class methods.

The implementation of run-super in §4.17 passes forward to an extension function both the method and attribute environments which result from instantiating a super-class (the *inherit* operator). No information is passed to the super-class as a result of instantiating the extension function. Information must pass both ways for a correct implementation of the run-inner feature:

1. The attribute environment created as a result of instantiating a super-class must be passed to an extension function in order to implement the inheritance scoping rules.

2. The method environment created as a result of instantiating an extension function must be passed to a super-class in order to make extensions to method implementations available.

This implies that the super-class cannot be instantiated until the extension function has been instantiated and the extension function cannot be instantiated until the super-class has been instantiated. This kind of mutual dependence can be achieved using recursive definitions. An implementation of the run-inner feature is as follows:

$$\textbf{let } \mathit{nullclass}(\_,()) = (\{\},\{\}) \qquad I_1$$

$$\begin{array}{l} \textbf{let } \mathit{inherit}(c,x)(m,(v_1,v_2)) = \qquad I_2 \\ \quad \textbf{letrec } (m_1,e_1) = c(m \oplus m_2, v_1) \\ \quad\quad\quad\; (m_2,e_2) = I(R(x(m)) \oplus e_1, x(m))(v_2) \\ \quad \textbf{in } (m_2 \oplus m_1, e_1 \oplus e_2) \end{array}$$

$$\textbf{let } \mathit{new}(c,v) = \mathit{1st}(c(\{\},v)) \qquad I_3$$

$$\textbf{let } \mathit{send}(o,i,v) = o \bullet i(v) \qquad I_4$$

$$\begin{array}{l} \textbf{let } x_1(\mathit{inner}) = Y\lambda x_1.\lambda x. \qquad I_5 \\ \quad \textbf{let } x = \mathit{loc}(x) \textbf{ in} \\ \quad \textbf{let } e = R(\lambda().()) - R(x_1) \\ \quad\quad m = \;(\text{`}\mathit{getx}\text{'} \mapsto \lambda().!x) \oplus \\ \quad\quad\quad\quad (\text{`}\mathit{reset}\text{'} \mapsto \lambda().x := 0; \mathit{send}(\mathit{inner}, \text{`}\mathit{reset}\text{'}, ())) \\ \quad \textbf{in } (m,e) \end{array}$$

$$\textbf{let } c_1 = \mathit{inherit}(\mathit{nullclass}, x_1) \qquad I_6$$

$$\begin{array}{l} \textbf{let } x_2(\mathit{inner}) = Y\lambda x_2.\lambda y. \qquad I_7 \\ \quad \textbf{let } y = \mathit{loc}(y) \textbf{ in} \\ \quad \textbf{let } e = R(\lambda().()) - R(x_2) \\ \quad\quad m = (\text{`}\mathit{gety}\text{'} \mapsto \lambda().!y) \oplus (\text{`}\mathit{reset}\text{'} \mapsto \lambda().y := 0) \\ \quad \textbf{in } (m,e) \end{array}$$

$$\textbf{let } c_2 = \mathit{inherit}(\mathit{nullclass}, x_2) \qquad I_8$$

Line I_1 implements the empty class as a function which is applied to a pair of arguments and which returns the pair containing empty environments. Line I_2 implements the *inherit* operator which constructs classes from a class c and an extension function x. A class is implemented as a function which is applied to a pair (m', v) and which returns a pair (m, e). The input pair contains m' which is an environment of methods and which is used as the value for *inner* in the extension x. The value v contains the initial values for the attributes of the class. The output pair contains an environment of methods m (the instance of the class) and an attribute environment e.

A class is constructed by composing extension functions and the empty class using the operator *inherit*, for example suppose that the extension functions x_1, x_2 and x_3 define attributes a_1, a_2 and a_3 and methods m_1, m_2 and m_3 respectively. Classes are defined as follows:

$$\textbf{let } c_1 = \mathit{inherit}(\mathit{nullclass}, x_1)$$

$$\textbf{let } c_2 = \mathit{inherit}(c_1, x_2)$$

$$\textbf{let } c_3 = \mathit{inherit}(c_2, x_3)$$

We analyse the instantiation of class c_3 by describing the values which take part in the instantiation of each component class of c_3. In particular we are interested in the value of *inner* which will be supplied to the extension part of each class and the contents of the inherited attribute environment.

When *nullclass* is instantiated, the value of *inner* will contain bindings for the methods m_1, m_2 and m_3 since these are used in the definition of classes on the path from *nullclass* to c_3. The empty class does not inherit from anywhere so there are no inherited attributes.

When c_1 is instantiated, the value of *inner* will contain bindings for the methods m_2 and m_3 and there will be no inherited attributes since the empty class does not define any.

When c_2 is instantiated, the value of *inner* will contain a binding for the method m_3 and the inherited attribute environment will contain a binding for a_1.

When c_3 is instantiated, the value of *inner* will contain no method bindings and the inherited attribute environment will contain bindings for a_1 and a_2.

Line I_2 implements the *inherit* operator using a fixed point to allow the instantiation of c to depend on that of x and vice versa. Line I_3 defines the instantiation operator, which supplies $\{\}$ as the initial value for *inner*. Lines I_4 and I_5 define the message passing and field reference operators. Lines I_5 – I_8 define the specified classes from lines S_1 and S_2. An extension function has the same form as that described in §4.17 except that the identifier *inner* is used instead of *super*.

In order to show that the implementation is correct with respect to the specification we define a representation for computational occurrences of classes, define calculation rules describing the computations which occur during class instantiation and show that the outcome of performing instantiation S_3 is:

$$\begin{array}{l}
o = o_2 \oplus o_1 \\
o_2 = (\text{`}v\text{'} \mapsto 2) \oplus (\text{`}\mathit{gety}\text{'} \mapsto k_1) \oplus (\text{`}\mathit{reset}\text{'} \mapsto k_2) \\
o_1 = (\text{`}v\text{'} \mapsto 1) \oplus (\text{`}\mathit{getx}\text{'} \mapsto k_3) \oplus (\text{`}\mathit{reset}\text{'} \mapsto k_4) \\
k_1 =< (), e \oplus (\text{`}\mathit{inner}\text{'} \mapsto \{\}) \oplus e_1 \oplus e_2, !y > \\
k_2 =< (), e \oplus (\text{`}\mathit{inner}\text{'} \mapsto \{\}) \oplus e_1 \oplus e_2, y := 0 > \\
k_3 =< (), e \oplus (\text{`}\mathit{inner}\text{'} \mapsto o_2) \oplus e_1, !x > \\
k_4 =< (), e \oplus (\text{`}\mathit{inner}\text{'} \mapsto o_2) \oplus e_1, x := 0; \mathit{send}(\mathit{inner}, \text{`}\mathit{reset}\text{'}, ()) > \\
e_1 = \text{`}x\text{'} \mapsto \alpha_1 \\
e_2 = \text{`}y\text{'} \mapsto \alpha_2
\end{array}$$

Classes and extension functions exist as computational values whose representation is defined in §4.15. The calculation rules for class instantiation using the run-inner feature are R_{73}, R_{66}, R_{74} and R_{75} which are defined below. The instantiation of the empty class is described by R_{73}:

$$(\mathit{nullclass} : (\_, \_) : s, e, @ : c, d, h) \longmapsto ((\{\}, \{\}) : s, e, c, d, h) \;\; R_{73}$$

The representation and instantiation of a single extension function is described by rule R_{66} in §4.15. The instantiation of a class constructed using the *inherit* operator is described by the calculation rule R_{74}. The rule shows the mutual dependence of the two instantiation calculations for the super-class and the extension:

$$\frac{\begin{array}{c}(k : (m \oplus m_2, v_1) : s, e, @ : c, d, h_1) \longmapsto ((m_1, e_1) : s, e, c, d, h_2) \\ (x(\tilde{i}, e_3 \oplus (\text{`}\mathit{inner}\text{'} \mapsto m) \oplus e_1, \tilde{j}, \tilde{k}, \tilde{b}) : v_2 : s, e, @ : c, d, h_2) \longmapsto ((m_2, e_2) : s, e, c, d, h_3)\end{array}}{(c(k, x(\tilde{i}, e_3, \tilde{j}, \tilde{k}, \tilde{b})) : (m, (v_1, v_2)) : s, e, @ : c, d, h_1) \longmapsto ((m_2 \oplus m_1, e_1 \oplus e_2) : s, e, c, d, h_3)} \; R_{74}$$

Finally, the calculation rule for the operator *new* is defined by R_{75}. The empty environment is supplied as the value for *inner* and the final attribute environment is thrown away:

$$\frac{(k : (\{\}, v) : s, e, @ : c, d, h_1) \longmapsto ((m, \_) : s, e, c, d, h_2)}{(\mathit{new} : (k, v) : s, e, @ : c, d, h_1) \longmapsto (m : s, e, c, d, h_2)} \; R_{75}$$

4.19 Feature 19: Merging the Run Super and Run Inner Mechanisms

The Λ-calculus can be used to generate a new programming feature which combines the facilities of run-super (described in §4.17) and run-inner (described in §4.18). Languages such as Smalltalk and BETA provide one feature and not the other, yet both argue that the feature which they support is useful.

The combined feature will work as follows. A new class is defined by adding an extension to a super-class. The extension defines new attributes and methods and may refer to methods from the super-class using the identifier *super* and may refer to methods from the sub-class using the identifier *inner*. Using this feature, methods may climb up and down the inheritance hierarchy at will.

A question arises as to how the combined feature will handle shadowing of methods, *i.e.* when a super-class and extension define methods with the same name. Since the feature will allow super-class methods to be referenced using *super* and sub-class methods to be reference using *inner* the choice is a matter of taste.

The implementation given below provides scope for super-class and sub-class shadowing within the same programming environment. They even allow sub-class and super-class shadowing to be mixed within the same class, although this could get confusing.

Each extension function is applied to three arguments $x(m_1, m_2)(v)$ where m_1 is the value of *super* and m_2 is the value of *inner*. The operator *inherit_super* is defined so that it creates classes which are instantiated to deliver objects in which super-class methods shadow sub-class methods. The operator *inherit_inner* is defined so that it creates classes which are instantiated to deliver objects in which sub-class methods shadow super-class methods. Both operators implement the rules of inheritance scoping and bind the identifiers *super* and *inner*.

$$\begin{array}{l}
\textbf{let } \mathit{nullclass}(_,_) = (\{\},\{\}) \\[1em]
\textbf{let } \mathit{inherit_super}(c,x)(m,(v_1,v_2)) = \\
\qquad \textbf{letrec} \quad (m_1,e_1) = c(m_2 \oplus m, v_1) \\
\qquad \qquad \qquad (m_2,e_2) = I(R(x(m_1,m)) \oplus e_1, x(m_1,m))(v_2) \\
\qquad \textbf{in } (m_2 \oplus m_1, e_1 \oplus e_2) \\[1em]
\textbf{let } \mathit{inherit_inner}(c,x)(m,(v_1,v_2)) = \\
\qquad \textbf{letrec} \quad (m_1,e_1) = c(m_2 \oplus m, v_1) \\
\qquad \qquad \qquad (m_2,e_2) = I(R(x(m_1,m)) \oplus e_1, x(m_1,m))(v_2) \\
\qquad \textbf{in } (m_1 \oplus m_2, e_1 \oplus e_2) \\[1em]
\textbf{let } \mathit{new}(c,v) = \mathit{1st}(c(\{\},v))
\end{array}$$

The representation for class expressions and computational objects is described in §4.18. The calculation rules are R_{73}, R_{75} and R_{76}:

$$\frac{\begin{array}{c}(k:(m\oplus m_2,v_1):s,e,@:c,d,h_1) \longmapsto ((m_1,e_1):s,e,c,d,h_2) \\ (x(\tilde{i},e_3\oplus(\text{‘}\mathit{inner}\text{’}\mapsto m)\oplus(\text{‘}\mathit{super}\text{’}\mapsto m_1)\oplus e_1,\tilde{j},\tilde{k},\tilde{b}):v_2:s,e,@:c,d,h_2) \longmapsto \\ ((m_2,e_2):s,e,c,d,h_3)\end{array}}{(c(k,x(\tilde{i},e_3,\tilde{j},\tilde{k},\tilde{b})):(m,(v_1,v_2)):s,e,@:c,d,h_1) \longmapsto ((m_2\oplus m_1,e_1\oplus e_2):s,e,c,d,h_3)}\ R_{76}$$

Rule R_{76} is the result of merging rules R_{74} and R_{72}.

4.20 Feature 20: Multiple Inheritance

Feature 20 requires the framework to support multiple inheritance. There are many different forms of multiple inheritance in object-oriented languages but in the simple case, a class is allowed to inherit from more than one super-class and will inherit all of the attributes and methods of each. The following is a specification for simple multiple inheritance:

$$
\begin{array}{ll}
\textbf{let } c_1 = & S_1 \\
\quad \textbf{class } \mathit{nullclass}\ (x) & \\
\quad\quad v = 1 & \\
\quad\quad \mathit{getx}() = !x & \\
\quad\quad f() = !y & \\
\quad \textbf{end} &
\end{array}
$$

$$
\begin{array}{ll}
\textbf{let } c_2 = & S_2 \\
\quad \textbf{class } \mathit{nullclass}\ (y) & \\
\quad\quad v = 2 & \\
\quad\quad \mathit{gety}() = !y & \\
\quad\quad g() = !x & \\
\quad \textbf{end} &
\end{array}
$$

$$
\begin{array}{ll}
\textbf{let } c_3 = & S_3 \\
\quad \textbf{class } (c_1, c_2)\ (z) & \\
\quad\quad v = 3 & \\
\quad\quad \mathit{getz}() = !z & \\
\quad\quad \mathit{reset}() = x := 0; y := 0; z := 0 & \\
\quad \textbf{end} &
\end{array}
$$

Lines S_1 – S_3 define three classes. Class c_1 defines a single attribute x, a single field v, a method *getx* for accessing the value of x and a method f which returns the value of the program location y (we assume that there is no y currently in scope).

Class c_2 defines a single attribute y, a field v, a method *gety* for accessing y and a method g which returns the value of the program location named x (we assume that there is no x currently in scope).

Class c_3 inherits from both c_1 and c_2. This means that all of the attributes, fields and methods defined by c_1 and c_2 are included in the definition of c_3. The class also defines a single attribute z, a field v, a method *getz* for accessing z and a method *reset* which is used to set all of the attributes to 0.

$$
\begin{array}{ll}
\textbf{let } o = \mathit{new}(c_3, ((((),10),((),20)),30)) & S_4 \\
 & \\
\mathit{ref}(o, `v\text{'}) \Rightarrow 3 & S_5 \\
\mathit{send}(o, `\mathit{getx}\text{'}, ()) \Rightarrow 10 & S_6 \\
\mathit{send}(o, `\mathit{gety}\text{'}, ()) \Rightarrow 20 & S_7 \\
\mathit{send}(o, `\mathit{getz}\text{'}, ()) \Rightarrow 30 & S_8 \\
\mathit{send}(o, `f\text{'}, ()) \Rightarrow \epsilon & S_9 \\
\mathit{send}(o, `g\text{'}, ()) \Rightarrow \epsilon & S_{10} \\
\mathit{send}(o, `\mathit{reset}\text{'}, ()) \Rightarrow 0 & S_{11} \\
\mathit{send}(o, `\mathit{getx}\text{'}, ()) \Rightarrow 0 & S_{12} \\
\mathit{send}(o, `\mathit{gety}\text{'}, ()) \Rightarrow 0 & S_{13} \\
\mathit{send}(o, `\mathit{getz}\text{'}, ()) \Rightarrow 0 & S_{14}
\end{array}
$$

Lines S_4 – S_{14} show the behaviour of an instance of c_3. Line S_4 instantiates c_3 to produce the object o. Line S_5 shows that the definition for v in class c_3 shadows the inherited definitions from c_2 and c_1. In general, sub-class definitions shadow super-class definitions with this simple multiple inheritance feature.

Lines S_6 – S_8 show that all the inherited methods are available from o. Lines S_9 – S_{10} show that the attributes of class c_1 are not scoped over the method bodies of class c_2 and vice versa, even though they are part of the same class definition, *i.e.* c_3. When a class (c_3) inherits from multiple super-classes (c_1 and c_2) we refer to the super-classes as *siblings* and they remain completely separate. The inheritance scoping rule does not extend to sibling related classes although the attributes of all super-classes are scoped over the methods defined by a sub-class, *i.e.* c_1 cannot refer to y, c_2 cannot refer to x but c_3 can refer to both x and y. Lines S_{11} – S_{14} show that the inheritance scoping rules are extended to include multiple super-classes.

An implementation for the specified multiple inheritance feature in the Λ-calculus represents classes as functions which are applied to attribute initial values and return a pair (m, e) where m is an environment binding method names to λ-closures and e is an environment binding attribute names to program locations (see §4.15 for more details).

When two classes are used as sibling super-classes, they can be instantiated separately producing (m_1, e_1) and (m_2, e_2). The methods and attributes produced by each of the sibling classes do not affect each other but will both contribute to the sub-class. The result of instantiating a pair of sibling super-classes is therefore $(m_1 \oplus m_2, e_1 \oplus e_2)$.

The result of instantiating a pair of sibling super-classes has the same format as the result of instantiating a single class. Therefore a pair of sibling super-classes can be regarded as a single class. Multiple sibling super-classes can be constructed by pairwise composition of individual sibling super-classes. In the implementation given below, an associative binary operator *siblings* is defined in line I_2 to compose any number of sibling super-classes into a single super-class.

$$\textbf{let } \mathit{nullclass}() = (\{\}, \{\}) \qquad I_1$$

$$\begin{array}{ll} \textbf{let } \mathit{siblings}(c_1, c_2)(v_1, v_2) = & I_2 \\ \quad \textbf{let } \ (m_1, e_1) = c_1(v_1) & \\ \quad \phantom{\textbf{let }} \ (m_2, e_2) = c_2(v_2) & \\ \quad \textbf{in } (m_1 \oplus m_2, e_1 \oplus e_2) & \end{array}$$

$$\begin{array}{ll} \textbf{let } \mathit{inherit}(c, x)(v_1, v_2) = & I_3 \\ \quad \textbf{let } (m_1, e_1) = c(v_1) \textbf{ in} & \\ \quad \textbf{let } (m_2, e_2) = I(R(x) \oplus m_1, x)(v_2) & \\ \quad \textbf{in } (m_1 \oplus m_2, e_1 \oplus e_2) & \end{array}$$

$$\textbf{let } \mathit{new}(c, v) = \mathit{1st}(c(v)) \qquad I_4$$

The following definitions implement the specified class c_3:

$$\begin{array}{ll}
\textbf{let } x_1(x) = & I_5 \\
\quad \textbf{let } x = loc(x) \textbf{ in} & \\
\quad \textbf{let } e = R(\lambda().()) - R(x_1) & \\
\quad\quad m = \ (\text{`}v\text{'} \mapsto 1) \oplus & \\
\quad\quad\quad (\text{`}getx\text{'} \mapsto \lambda().!x) \oplus & \\
\quad\quad\quad (\text{`}f\text{'} \mapsto \lambda().!y) & \\
\quad \textbf{in } (m, e) &
\end{array}$$

$$\textbf{let } c_1 = inherit(nullclass, x_1) \qquad I_6$$

$$\begin{array}{ll}
\textbf{let } x_2(y) = & I_7 \\
\quad \textbf{let } y = loc(y) \textbf{ in} & \\
\quad \textbf{let } e = R(\lambda().()) - R(x_2) & \\
\quad\quad m = \ (\text{`}v\text{'} \mapsto 2) \oplus & \\
\quad\quad\quad (\text{`}gety\text{'} \mapsto \lambda().!y) \oplus & \\
\quad\quad\quad (\text{`}f\text{'} \mapsto \lambda().!x) & \\
\quad \textbf{in } (m, e) &
\end{array}$$

$$\textbf{let } c_2 = inherit(nullclass, x_2) \qquad I_8$$

$$\begin{array}{ll}
\textbf{let } x_3(z) = & I_9 \\
\quad \textbf{let } z = loc(z) \textbf{ in} & \\
\quad \textbf{let } e = R(\lambda().()) - R(x_3) & \\
\quad\quad m = \ (\text{`}v\text{'} \mapsto 3) \oplus & \\
\quad\quad\quad (\text{`}getz\text{'} \mapsto \lambda().!z) \oplus & \\
\quad\quad\quad (\text{`}reset\text{'} \mapsto \lambda().x := 0; y := 0; z := 0) & \\
\quad \textbf{in } (m, e) &
\end{array}$$

$$\textbf{let } c_3 = inherit(siblings(c_1, c_2), x_3) \qquad I_{10}$$

The implementation is correct with respect to the specification when the outcome of defining the object o is as follows:

$$o = o_1 \oplus o_2 \oplus o_3 \qquad V_1$$

$$o_1 = (\text{`}v\text{'} \mapsto 1) \oplus (\text{`}getx\text{'} \mapsto <(), e \oplus e_1, !x>) \oplus (\text{`}f\text{'} \mapsto <(), e \oplus e_1, !y>) \qquad V_2$$

$$o_2 = (\text{`}v\text{'} \mapsto 2) \oplus (\text{`}gety\text{'} \mapsto <(), e \oplus e_2, !y>) \oplus (\text{`}g\text{'} \mapsto <(), e \oplus e_2, !x> \qquad V_3$$

$$\begin{array}{ll}
o_3 = (\text{`}v\text{'} \mapsto 3) \oplus (\text{`}getz\text{'} \mapsto <(), e \oplus e_3, !y>) \oplus (\text{`}reset\text{'} \mapsto k) & V_4 \\
k = <(), e \oplus e_3, x := 0; y := 0; z := 0> &
\end{array}$$

$$e_1 = \text{`}x\text{'} \mapsto \alpha_1 \ e_2 = \text{`}y\text{'} \mapsto \alpha_2 \ e_3 = e_1 \oplus e_2 \oplus (\text{`}z\text{'} \mapsto \alpha_3) \qquad V_5$$

$$h_1 = h \oplus (\alpha_1 \mapsto 10) \oplus (\alpha_2 \mapsto 20) \oplus (\alpha_3 \mapsto 30) \qquad V_6$$

$$h_2 = h_1 \oplus (\alpha_1 \mapsto 0) \oplus (\alpha_2 \mapsto 0) \oplus (\alpha_3 \mapsto 0) \qquad V_7$$

Line V_1 shows that the instance of class c_3 is a composite object made up of an instance o_1 of class c_1, an instance o_2 of class c_2 and an instance of the extension x_3. Lines V_2 – V_4 show the individual instances. Line V_5 shows the attribute environments of the individual instances. Line V_6 shows the heap after the instances are initially created and line V_7 shows the heap after the attributes are reset.

Calculation rules are used to describe the instantiation of classes. A class is represented using the notation described in §4.15 in addition to a pair of sibling classes which are represented as (c_1, c_2). The calculation rules for class instantiation are as shown in §4.15 in addition to the sibling class instantiation rule:

$$\frac{\begin{array}{c}(c_1 : v_1 : s, e, @ : c, d, h_1) \longmapsto ((m_1, e_1) : s, e, c, d, h_2)\\ (c_2 : v_2 : s, e, @ : c, d, h_2) \longmapsto ((m_2, e_2) : s, e, c, d, h_3)\end{array}}{((c_1, c_2) : (v_1, v_2) : s, e, @ : c, d, h_1) \longmapsto ((m_1 \oplus m_2, e_1 \oplus e_2) : s, e, c, d, h_3)}\ R_{77}$$

Rule R_{77} shows that a sibling class (c_1, c_2) is instantiated by first instantiating c_1 and then instantiating c_2. The order is arbitrary and a variation is to define that c_2 is instantiated before c_1.

4.21 Feature 21: Repeated Inheritance

Feature 21 requires the framework to provide support for dealing with repeated inheritance. This occurs when a class inherits from the same super-class twice leading to possible ambiguity with respect to the number of program locations which are allocated for multiple occurrences of the same attribute. There are many possible approaches to repeated inheritance which lead to different outcomes, for example Eiffel defines that all occurrences of the same attribute must share whilst [66] advocates that this can compromise modularity and therefore different occurrences of the same attribute should lead to different program locations.

The Λ-calculus provides program locations as first class data values and therefore can be used as a flexible basis for experimentation with different approaches to repeated inheritance. This section specifies a simple repeated inheritance mechanism, based on that of Eiffel, and shows how it is implemented using the facilities of the Λ-calculus. A specification is as follows:

$$
\begin{array}{ll}
\textbf{let } c_1 = & S_1 \\
\quad \textbf{class } \mathit{nullclass}\ (x) & \\
\quad\quad \mathit{setx}(v) = x := v & \\
\quad \textbf{end} & \\
 & \\
\textbf{let } c_2 = & S_2 \\
\quad \textbf{class } c_1\ () & \\
\quad\quad f() = !x & \\
\quad \textbf{end} & \\
 & \\
\textbf{let } c_3 = & S_3 \\
\quad \textbf{class } c_1\ () & \\
\quad\quad g() = !x & \\
\quad \textbf{end} & \\
 & \\
\textbf{let } c_4 = & S_4 \\
\quad \textbf{class } (c_2, c_3) & \\
\quad\quad h() = !x & \\
\quad \textbf{end} & \\
 & \\
\textbf{let } o = \mathit{new}(c_4, ((((), 10), ((), 20)), ())) & S_5 \\
 & \\
\mathit{send}(o, `f', ()) \Rightarrow 10 & S_6 \\
\mathit{send}(o, `g', ()) \Rightarrow 10 & S_7 \\
\mathit{send}(o, `h', ()) \Rightarrow 10 & S_8 \\
\mathit{send}(o, `\mathit{setx}', 20) \Rightarrow 20 & S_9 \\
\mathit{send}(o, `f', ()) \Rightarrow 20 & S_{10} \\
\mathit{send}(o, `g', ()) \Rightarrow 20 & S_{11} \\
\mathit{send}(o, `h', ()) \Rightarrow 20 & S_{12}
\end{array}
$$

Line S_1 defines a class c_1 which has a single attribute x and a method for updating the value of the attribute. Lines S_2 and S_3 define two classes which each inherit from c_1 and provide methods for accessing the inherited attribute x.

Line S_4 defines a class which multiply inherits from c_2 and c_3. Multiple inheritance is defined so that if a class inherits from a super-class more than once, it will allocate program locations for the inherited attributes once. Note that this is not implemented by the feature in §4.20 which would allocate two separate program locations for each occurrence of an inherited attribute. Where a class is inherited more than once, all but the left-most occurrence is ignored.

Line S_5 creates an instance of the class c_4 and lines S_6 – S_{12} show that all occurrences of the attribute x share the same program location. Note that since all but the left-most occurrence of c_1 is ignored, the initial value 20 for x will have no effect (and could have been supplied as ()).

The specified repeated inheritance mechanism is implemented in the Λ-calculus by associating each extension with a unique tag. During class instantiation a database of tags is maintained which records the classes whose attributes have been allocated program locations.

When an extension is supplied with attribute values it will allocate fresh program locations for each attribute and initialize the locations with the values. As a side effect of this process, the database of tags is updated to record that the extension has been instantiated. If, due to repeated inheritance, an extension is visited more than once then its tag will be present in the database and this is used to test whether or not program locations should be allocated. An implementation is as follows:

$$\textbf{let } \mathit{nullclass}((), r) = (\{\}, \{\}, r) \qquad I_1$$

$$\begin{aligned}
&\textbf{let } \mathit{siblings}(c_1, c_2)((v_1, v_2), r_1) = && I_2\\
&\qquad \textbf{let } (m_1, e_1, r_2) = c_1(v_1, r_1) \textbf{ in}\\
&\qquad \textbf{let } (m_2, e_2, r_3) = c_2(v_2, r_2)\\
&\qquad \textbf{in } (m_1 \oplus m_2, e_1 \oplus e_2, r_3)
\end{aligned}$$

$$\begin{aligned}
&\textbf{let } \mathit{inherit}(c, x)((v_1, v_2), r_1) = && I_3\\
&\qquad \textbf{let } (m_1, e_1, r_2) = c(v_1, r_1) \textbf{ in}\\
&\qquad \textbf{let } (m_2, e_2, r_3) = I(R(x(r_2)) \oplus e_1, x(r_2))(v_2)\\
&\qquad \textbf{in } (m_1 \oplus m_2, e_1 \oplus e_2, r_3)
\end{aligned}$$

$$\textbf{let } \mathit{new}(c, v) = \mathit{1st}(c(v, \{\})) \qquad I_4$$

Line I_1 shows the implementation of the empty class. Line I_2 shows the implementation of the operator which merges classes for multiple inheritance. Note the database of extension tags r is threaded through the instantiation of classes c_1 and c_2. Line I_3 shows the implementation of the operator which constructs classes from a super-class and an extension. Both the class c and the extension x are supplied with and produce tag databases. Finally, line I_4 shows the implementation of the class instantiation operator which supplies the empty environment as the initial tag database. The implementation of the specified classes is as follows:

$$\begin{aligned}
&\textbf{let } x_1(r) = Y\lambda x_1.\lambda x. && I_5\\
&\quad \textbf{if } \tau_1 \in \mathit{dom}(r)\\
&\quad \textbf{then } r \bullet \tau_1\\
&\quad \textbf{else let } x = \mathit{loc}(x) \textbf{ in}\\
&\qquad\quad \textbf{let } e = R(\lambda().()) - R(x_1)\\
&\qquad\qquad\quad m = \text{`}\mathit{setx}\text{'} \mapsto \lambda v.x := v \textbf{ in}\\
&\qquad\quad \textbf{in } Y\lambda t.(m, e, r \oplus (\tau_1 \mapsto t))
\end{aligned}$$

$$\textbf{let } c_1 = \mathit{inherit}(\mathit{nullclass}, x_1) \qquad I_6$$

$$\begin{aligned}
&\textbf{let } x_2(r) = Y\lambda x_2.\lambda x. && I_7\\
&\quad \textbf{if } \tau_2 \in \mathit{dom}(r)\\
&\quad \textbf{then } r \bullet \tau_2\\
&\quad \textbf{else let } x = \mathit{loc}(x) \textbf{ in}\\
&\qquad\quad \textbf{let } e = R(\lambda().()) - R(x_2)\\
&\qquad\qquad\quad m = \text{`}f\text{'} \mapsto \lambda().!x \textbf{ in}\\
&\qquad\quad \textbf{in } Y\lambda t.(m, e, r \oplus (\tau_2 \mapsto t))
\end{aligned}$$

$$
\begin{array}{ll}
\textbf{let } c_2 = \mathit{inherit}(\mathit{nullclass}, x_2) & I_8 \\
 & \\
\textbf{let } x_3(r) = Y\lambda x_3.\lambda x. & I_9 \\
\quad \textbf{if } \tau_3 \in \mathit{dom}(r) & \\
\quad \textbf{then } r \bullet \tau_3 & \\
\quad \textbf{else let } x = \mathit{loc}(x) \textbf{ in} & \\
\qquad \textbf{let } e = R(\lambda().()) - R(x_3) & \\
\qquad\quad m = \text{`}g\text{'} \mapsto \lambda().!x \textbf{ in} & \\
\qquad \textbf{in } Y\lambda t.(m, e, r \oplus (\tau_3 \mapsto t)) & \\
 & \\
\textbf{let } c_3 = \mathit{inherit}(\mathit{nullclass}, x_3) & I_9 \\
 & \\
\textbf{let } x_4(r) = Y\lambda x_4.\lambda x. & I_{10} \\
\quad \textbf{if } \tau_4 \in \mathit{dom}(r) & \\
\quad \textbf{then } r \bullet \tau_4 & \\
\quad \textbf{else let } x = \mathit{loc}(x) \textbf{ in} & \\
\qquad \textbf{let } e = R(\lambda().()) - R(x_4) & \\
\qquad\quad m = \text{`}h\text{'} \mapsto \lambda().!x \textbf{ in} & \\
\qquad \textbf{in } Y\lambda t.(m, e, r \oplus (\tau_4 \mapsto t)) & \\
 & \\
\textbf{let } c_4 = \mathit{inherit}(\mathit{siblings}(c_2, c_3), x_4) & I_{11}
\end{array}
$$

Lines I_5 – I_{10} shows the implementation of the specified classes. Each class is defined in terms of an extension function which is combined with a super-class. In the case of c_4 the super-class is constructed using *siblings*. Each extension function has the following general format:

$$
\begin{array}{ll}
X_1 & \textbf{let } x(r) = Y\lambda x.\lambda(i_1, \ldots, i_n). \\
X_2 & \quad \textbf{if } \tau \in \mathit{dom}(r) \\
X_3 & \quad \textbf{then } r \bullet \tau \\
X_4 & \quad \textbf{else let } (i_1, \ldots, i_n) = (\mathit{loc}(i_1), \ldots, \mathit{loc}(i_n)) \textbf{ in} \\
X_5 & \qquad \textbf{let } e = R(\lambda().()) - R(x) \\
X_6 & \qquad\quad m = (m_1 \mapsto f_1) \oplus \ldots \oplus (m_k \mapsto f_k) \\
X_7 & \qquad \textbf{in } \; Y\lambda t.(m, e, r \oplus (\tau \mapsto t))
\end{array}
$$

Line X_1 defines the extension function x. The function is recursively defined using Y so that it can refer to the environment which is current when it is created. The identifiers $i_1 \ldots i_n$ are attributes defined by the extension. Line X_2 tests whether the extension has already been instantiated. If it has then line X_3 looks the extension's tag up in the database r returning the same program locations as the previous instantiation. Lines X_4 – X_8 are performed when the extension has not been previously instantiated. Line X_4 allocates fresh program locations for the attributes, line X_5 reifies the attribute environment, line X_6 defines the methods and line X_7 creates the return value t. The database which is returned contains the extension tag τ mapped to the return value t.

The implementation is correct with respect to the specification when the outcome of defining the object o is as follows:

$$
\begin{array}{ll}
o = o_1 \oplus o_5 \oplus o_1 \oplus o_6 \oplus o_4 & V_1 \\
o_5 = \text{`}f\text{'} \mapsto <(), e \oplus e_1, !x> & V_2 \\
o_6 = \text{`}g\text{'} \mapsto <(), e \oplus e_1, !x> & V_3 \\
o_1 = \text{`}setx\text{'} \mapsto <\text{`}v\text{'}, e \oplus e_1, x := v> & V_4 \\
o_4 = \text{`}h\text{'} \mapsto <(), e \oplus e_1, !x> & V_5 \\
e_1 = \text{`}x\text{'} \mapsto \alpha & V_6 \\
h_1 = h \oplus (\alpha \mapsto 10) & V_7 \\
h_2 = h_1 \oplus (\alpha \mapsto 20) & V_8
\end{array}
$$

Lines V_1 – V_6 show the components of object o. Line V_7 is the heap after o is created. Line V_8 is the heap after the attribute x is updated.

The representation for classes and extensions is the same as shown in §4.20 except that an extension function has a tag τ associated with it: $x(\tau, \tilde{i}, e, \tilde{j}, \tilde{k}, \tilde{b})$. The calculation rules which describe class instantiation are as follows.

$$(nullclass : ((), r) : s, e, @ : c, d, h) \longmapsto ((\{\}, \{\}, r) : s, e, c, d, h) \quad R_{78}$$

Rule R_{78} shows the instantiation of the empty class. Rule R_{79}:

$$\frac{\tau \in dom(r)}{(x(\tau, \_, \_, \_, \_, \_) : (\_, r) : s, e, @ : c, d, h) \longmapsto (r \bullet \tau : s, e, c, d, h)} \; R_{79}$$

shows the instantiation of an extension function which has already been instantiated with respect to the database r. The outcome is defined by the contents of the database indexed by the tag τ. Rule R_{80}:

$$\frac{t = (\tilde{j} \mapsto <\tilde{k}, e_1 \oplus (\tilde{i} \mapsto \tilde{\alpha}), \tilde{b}>, \tilde{i} \mapsto \tilde{\alpha}, r \oplus (\tau \mapsto t))}{(x(\tau, \tilde{i}, e_1, \tilde{j}, \tilde{k}, \tilde{b}) : (\tilde{v}, r) : s, e_2, @ : c, d, h) \longmapsto (t : s, e_2, c, d, h \oplus (\tilde{\alpha} \mapsto \tilde{v}))} \; R_{80}$$

shows the instantiation of an extension function which has not already been instantiated with respect to the database r. Locations are allocated for each attribute and the database is updated. Rule R_{81}:

$$\frac{\begin{array}{c}(c_1 : (v_1, r_1) : s, e, @; c, d, h_1) \longmapsto ((m_1, e_1, r_2) : s, e, c, d, h_2) \\ (c_2 : (v_2, r_2) : s, e, @; c, d, h_2) \longmapsto ((m_2, e_2, r_3) : s, e, c, d, h_3)\end{array}}{((c_1, c_2), ((v_1, v_2), r_1) : s, e, @ : c, d, h_1) \longmapsto ((m_1 \oplus m_2, e_1 \oplus e_2, r_3) : s, e, c, d, h_3)} \; R_{81}$$

shows the instantiation of sibling classes. Rule R_{82}:

$$\frac{\begin{array}{c}(c:(v_1,r_1):s,e_2,@:c,d,h_1)\longmapsto((m_1,e_3,r_2):s,e_3,c,d,h_2)\\(x(\tau,\tilde{i},e_1\oplus e_3,\tilde{j},\tilde{k},\tilde{b}):(v_2,r_2):s,e_2,@:c,d,h_2)\longmapsto((m_2,e_4,r_3):s,e_2,c,d,h_3)\end{array}}{\begin{array}{c}(c(k,x(\tau,\tilde{i},e_1,\tilde{j},\tilde{k},\tilde{b}):((v_1,v_2),r_1):s,e_2,@:c,d,h_1)\longmapsto\\((m_1\oplus m_2,e_3\oplus e_4,r_3):s,e_2,c,d,h_3)\end{array}}\;R_{82}$$

shows the instantiation of a class composed of a super-class and an extension function. Finally, rule R_{83}

$$\frac{(c:(v,\{\}):s,e,@;c,d,h_1)\longmapsto((o,\_,\_):s,e,c,d,h_2)}{(new:(c,v):s,e,@:c,d,h_1)\longmapsto(o:s,e,c,d,h_2)}\;R_{83}$$

shows the instantiation of a class using the operator *new*.

4.22 Feature 22: Inclusion Polymorphism

Feature 22 specifies that the framework must support inclusion polymorphism. This mechanism is characteristic of strongly typed object-oriented languages and provides a basis for the reuse of operations. The signature of an operation defines the types of its arguments and result. The types are classes whose instances may be passed to the operation as actual parameters and returned as results. Inclusion polymorphism allows the instances of different classes to be passed as arguments or returned as results providing they inherit from the classes defined in the signature. The inheritance relationship guarantees that the methods which are required by the operation are available from any value which is passed to the operation as an argument. Such a type system is usually syntactic in the sense that is guarantees that name references will never go wrong when they are performed at run-time, however there is no way of specifying that the supplied instance behaves as intended.

A specification for a simple inclusion polymorphism mechanism is as follows:

$$\begin{array}{ll}
\textbf{let}\ c_1 = \textbf{class}\ (x)\ i() = !x\ \textbf{end} & S_1\\[1em]
\textbf{let}\ c_2 = \textbf{class}\ c_1\ ()\ j(v) = x := v\ \textbf{end} & S_2\\[1em]
\textbf{let}\ o_1 = new(c_1,((),10)) & S_3\\[1em]
\textbf{let}\ o_2 = new(c_2,(((),10),())) & S_4\\[1em]
\textbf{let}\ f(c)(o:c,i,v) = send(o,i,v) & S_5\\[1em]
f(c_1)(o_1,\text{`}i\text{'},()) \Rightarrow 10 & S_6\\
f(c_1)(o_2,\text{`}i\text{'},()) \Rightarrow 10 & S_7\\
f(c_1)(o_2,\text{`}j\text{'},20) \Rightarrow fail & S_8\\
f(c_2)(o_1,\text{`}i\text{'},()) \Rightarrow fail & S_9\\
f(c_2)(o_2,\text{`}i\text{'},()) \Rightarrow 10 & S_{10}\\
f(c_2)(o_2,\text{`}j\text{'},20) \Rightarrow 20 & S_{11}\\
f(c_2)(o_2,\text{`}i\text{'},()) \Rightarrow 20 & S_{12}
\end{array}$$

Class c_1 in line S_1 defines an attribute x and a method i which returns the current value of the program location named x. Class c_2 in line S_2 inherits from c_1 and defines a method j

which updates the value of the inherited attribute x. The objects o_1 and o_2 in lines S_3 and S_4 are instances of the respective classes.

The function f in line S_5 is defined using inclusion polymorphism for the argument o. The function is applied to a class c and the declaration $o : c$ defines that the argument o must be an object whose method names must conform to those defined by c. It is permissible to supply f with an object which has more methods than those defined by c, but these will be inaccessible via o. If f is passed an object whose method names do not conform to those defined by c then f returns *fail* (providing scope for this mechanism to be integrated into pattern matching – see chapter 3).

The function f is also supplied with a name i and a value v and will send a message with name i and value v to o, providing that o passes the test $o : c$.

Lines S_6 – S_{12} show the outcomes of using f to send messages named i and j in objects o_1 and o_2 using the classes c_1 and c_2 to define the type for inclusion polymorphism. When c_1 is supplied, only the method named i is available. When c_2 is supplied, both i and j are available in o_2, but o_1 is not an instance of c_2 and therefore causes f to produce *fail*.

There are many variations on the theme of inclusion polymorphism. Some are based on the "shape" of the objects which are passed to operations and some are based on the identity of the class of the objects which are passed to the operations. Given the definition:

$$\textbf{let } c_3 = \textbf{class } c_1 \ (y) \ j(v) = x := v \textbf{ end}$$

then a polymorphic type system based on the shape of the objects would allow:

$$f(c_3)(o_2, `j', 20) \Rightarrow 20$$

since c_2 and c_3 have identical method names whereas a type system based on the identity of the classes would distinguish between objects which are created by instantiating c_2 and those created by instantiating c_3, even though they both have the same "shape".

The following simple implementation of inclusion polymorphism in the Λ-calculus is based upon the shape of objects. It defines a class as a pair (S, g) where S is a set containing the method names which are defined by the class and g is a function which is applied to the initial values for the attributes and returns a pair containing two environments. The first component of the pair is an environment associating method names with closures and the second component is an environment associating attribute names with their values. When a class is used in a parameter declaration, the set of method names is used to "project" the supplied object to the given shape. If the supplied object does not contain the required names then the function produces *fail*. The implementation is as follows:

$$\textbf{let } \mathit{nullclass} = (\emptyset, \lambda().(\{\}, \{\})) \qquad I_1$$

$$\begin{array}{ll} \textbf{let } \mathit{generate}(c,x)(v_1,v_2) = & I_2 \\ \quad \textbf{let } (m_1,e_1) = \mathit{2nd}(c)(v_1) \textbf{ in} & \\ \quad \textbf{let } (m_2,e_2) = I(R(x) \oplus e_1, x)(v_2) & \\ \quad \textbf{in } (m_1 \oplus m_2, e_1 \oplus e_2) & \end{array}$$

$$\textbf{let } \mathit{inherit}(c,x) = (\mathit{1st}(c) \cup \mathit{names}(x), \mathit{generate}(c,x)) \qquad I_3$$

$$\textbf{let } x_1(x) = \textbf{let } x = \mathit{loc}(x) \textbf{ in } (\text{`}i\text{'} \mapsto \lambda().!x, R(\lambda().()) - R(x_1)) \quad I_4$$

$$\textbf{let } c_1 = \mathit{inherit}(\mathit{nullclass}, x_1) \qquad I_5$$

$$\textbf{let } x_2() = (\text{`}j\text{'} \mapsto \lambda v.x := v, \{\}) \qquad I_6$$

$$\textbf{let } c_2 = \mathit{inherit}(c_1, x_2) \qquad I_7$$

$$\textbf{let } \mathit{new}(c,v) = \mathit{1st}(\mathit{2nd}(c)(v)) \qquad I_8$$

Line I_1 defines the empty class. The empty class contains no method names and returns a pair of empty environments corresponding to the attribute environment and method environment.

Line I_2 defines an operator *generate* which corresponds to the operator *inherit* as defined in §4.15. The difference is that the class c is a pair whose second component is an object generator.

Line I_3 defines an operator *inherit* which constructs a class from a super-class c and an extension function x. The implementation shows that the result is a pair. The first component of the pair is the union of the sets of method names defined by the super-class and the extension. The operator *names* is applied to an extension function and returns the method names which it defines. The behaviour of *names* is described by the rule R_{84}:

$$(\mathit{names} : x(\_,\_,\tilde{j},\_,\_) : s, e, @ : c, d, h) \longmapsto (\{\tilde{j}\} : s, e, c, d, h) \; R_{84}$$

Lines I_4 – I_7 define the classes c_1 and c_2 and line I_8 defines the class instantiation operator. A function which uses inclusion polymorphism has the following format:

$$\lambda i : e_1.e_2$$

where i is the formal parameter of the function, e_1 is a program expression which denotes a class and e_2 is the body of the function. The expression e_1 is performed once when the function is created and produces a class. The actual parameter which is bound to i must be an object which includes all the method names defined by the class. If this is the case then the method names of i are restricted to be exactly those of the class and e_2 is performed. Otherwise the value *fail* is returned. The inclusion function format is sugar for the following function:

$$\begin{array}{l} \textbf{let } S = \mathit{1st}(e_1) \\ \textbf{in } \lambda i.\ \textbf{if } S \subseteq \mathit{dom}(i) \\ \qquad\quad \textbf{then let } i = i \backslash S \textbf{ in } e_2 \\ \qquad\quad \textbf{else } \mathit{fail} \end{array}$$

Such a function is represented as $<S, i, e, b>$ and its application is described by calculation rules R_{85} and R_{86}:

$$\frac{S \subseteq dom(e_2)}{(<S, i, e_1, b> : e_2 : s, e_3, @ : c, d, h) \longmapsto (<i, e_1, b> : e_2 \backslash S : s, e_3, @ : c, d, h)} \; R_{85}$$

Rule R_{85} shows the application of an inclusion function when the actual parameter e_2 is of the required type.

$$\frac{dom(e_1) \subset S}{(<S, \_, \_, \_> : e_1 : s, e_2, @ : c, d, h) \longmapsto (fail : s, e_2, c, d, h)} \; R_{86}$$

Rule R_{86} shows the application of an inclusion function when the actual parameter e_1 is not of the required type.

Inclusion polymorphism which is based upon the identity of classes could be implemented by associating each class with a set of tags. Sub-classes have a super-set of the tags of each of its super-classes. An inclusion function would be represented as $<S, i, e, b>$ where S is the set of tags associated with the formal parameter type. An instance of a class must contain the set of tags from the class. When an object is supplied as an actual parameter to an inclusion function, the set of tags in the object can be compared against the set of tags from the formal parameter type. If the object's tag set is a super-set of the formal parameter type's tag set then the application is legal.

4.23 Feature 23: Non-Standard Control

Feature 23 requires the framework to support non-standard control mechanisms. Chapter 2 shows that there is a wide variety of non-standard control features in object-oriented languages. All of these features can be implemented using the J operator of the Λ-calculus, this section uses the J operator to implement the Smalltalk block construct which is specified as follows:

$$\begin{array}{ll}
\textbf{let } c = & S_1 \\
\quad \textbf{class } (x) & \\
\quad\quad f() = & \\
\quad\quad\quad \textbf{let } b = \textbf{block } (y) \textbf{ return}(!x * y) \textbf{ end} & \\
\quad\quad\quad \textbf{in } h(b) * 100 & \\
\quad\quad g() = \textbf{block } (y) \textbf{ return}(!x * y) \textbf{ end} & \\
\quad\quad h(b) = send(b, \text{‘}value\text{’}, 3) & \\
\quad \textbf{end} & \\
& \\
\textbf{let } o = new(c, 10) & S_2 \\
& \\
send(o, \text{‘}f\text{’}, ()) \Rightarrow 30 & S_3 \\
send(send(o, \text{‘}g\text{’}, ()), \text{‘}value\text{’}, 3) \Rightarrow \text{‘}error\text{’} & S_4
\end{array}$$

Line S_1 defines a class c which has a single attribute x and three methods. Method f creates a block and supplies it to the method h. A block is expressed using the syntax **block** i e **end** where i is a formal parameter and e is the body of the block. A block is an object with a single method named *value*. Blocks and their interaction with **return** expressions is described in chapter 2. Method g creates a block and returns it without activating it. Method h is supplied with a block and activates it by supplying it with 3.

Line S_2 creates an instance of the class c. Line S_3 shows that the block mechanism is activated as defined in Smalltalk, *i.e.* when a block returns it does so from the method activation which created it. Line S_4 shows that, as defined by Smalltalk, it is illegal to activate a block outside the dynamic scope of the method which created it.

The Λ-calculus provides the operator J for constructing non-standard control mechanisms. An implementation for blocks is as follows:

$$\begin{array}{l}
\textbf{let } b() = \\
\quad \textbf{let } l = \mathit{loc}(\mathit{false}) \textbf{ in} \\
\quad \textbf{let } r(v) = \\
\quad\quad \textbf{if } !l \\
\quad\quad \textbf{then } \mathit{halt}(\text{`}\mathit{error}\text{'}) \\
\quad\quad \textbf{else } l := \mathit{true}; v \\
\quad \textbf{in } r
\end{array} \qquad I_1$$

$$\begin{array}{l}
\textbf{let } c(x) = \\
\quad \textbf{let } x = \mathit{loc}(x) \textbf{ in} \\
\quad \textbf{letrec } f() = \textbf{let } r = J(b()) \textbf{ in } r(h(\text{`}\mathit{value}\text{'} \mapsto \lambda y.r(!x * y)) * 100) \\
\quad\quad\quad\quad g() = \textbf{let } r = J(b()) \textbf{ in } r(\text{`}\mathit{value}\text{'} \mapsto \lambda y.r(!x * y)) \\
\quad\quad\quad\quad h(b) = \textbf{let } r = J(b()) \textbf{ in } r(\mathit{send}(b, \text{`}\mathit{value}\text{'}, 3)) \\
\quad \textbf{in } (\text{`}f\text{'} \mapsto f) \oplus (\text{`}g\text{'} \mapsto g) \oplus (\text{`}h\text{'} \mapsto h)
\end{array} \qquad I_2$$

Line I_1 defines an operator b which allocates a new program location initialised with *false*. The operator is called when a method is activated and the location which is created is used to test whether the method has returned. A method returns by applying the operator which is returned by b. This operator updates the contents of the location l to contain *true*. If the method has already returned then the return operator will detect this and cause an error to occur.

Line I_2 shows the implementation of class c. Each method in the specification has been expanded in the implementation to include a return operator r which is created by the operator b each time the method is invoked. The return operator is used whenever a method returns a value. The return operator is captured by the block created by methods f and g.

The return operator r is a program point created by applying J to the function returned by $b()$. In doing so, r captures the current machine dump *i.e.* the context to which the current method activation will return. When r is applied, all current pending contexts are discarded until the context which was pending when r was created is found. Note, however, that the mechanism defined in b will allow a return function to be used at most once. This ensures that contexts can be created and discarded in a stack-like fashion.

A block is implemented as a singleton environment which binds *value* to a closure. A block may only be created inside a method and therefore captures the return operator r. Since a block is a first class data value it may be passed to other operators as an argument and then invoked by sending it a *value* message. If the body of the block then applies the operator r then all intervening contexts are discarded in order to return from the method activation which created the block.

The Λ-calculus can be extended with evaluation rules for Smalltalk blocks as follows. A machine stack always contains two special components which control the block feature. The bottom two elements of a machine stack are always tags. The bottom-most tag is associated with the current closure activation and is unique for each activation. The second bottom-most tag is used to control where the current activation will return to when it performs a **return** expression. Rule R_{87} describes what happens when a function closure is applied:

$$(<i,e',b> : v : s, e, @ : c, d, h) \longmapsto ([\tau,\tau], e' \oplus (i \mapsto v), [b], (s,e,c,d), h) \ R_{87}$$

a new tag τ is coined and placed twice at the bottom of the machine stack. A block is a data value which is like a function closure except that it contains a tag. It is represented as $<\tau, i, e, b>$ where τ is the bottom-most tag in the current machine stack. This tag describes where the block will return from when it is activated and performs a **return**. Block creation is described by rule R_{88}:

$$(s +\!\!+ [\tau], e, \mathbf{block}\ i\ b\ \mathbf{end} : c, d, h) \longmapsto (<\tau,i,e,b> : s +\!\!+ [\tau], e, c, d, h) \ R_{88}$$

When a block is sent a *value* message it behaves like the application of a function closure, except that the second bottom-most tag is that found in the block:

$$\begin{array}{l}(\mathit{send} : (<\tau_1, i, e_1, b>, \text{`}\mathit{value}\text{'}, v) : s, e_2, @ : c, d, h) \\ \quad \longmapsto ([\tau_1,\tau_2], e_1 \oplus (i \mapsto v), [b], (s,e,c,d), h) \ R_{89}\end{array}$$

Rule R_{89} shows that the bottom-most tag is newly coined on block activation, whereas the second bottom-most tag is that found in the block. When a **return** is performed, there will be a value at the top of the stack which is to be returned to some machine state which is part of the dump component. The current return tag (second bottom-most) defines the machine state which will be returned from. The **return** causes a search for a machine state whose bottom-most tag matches the second bottom-most tag of the returning state. There are two possible outcomes. Rule R_{90} describes the case where the two tags are the same in the current state:

$$(v +\!\!+ [\tau,\tau], \_, \mathbf{return} : \_, (s,e,c,d), h) \longmapsto (v : s, e, c, d, h) \ R_{90}$$

in which case the **return** behaves like ordinary function return. Otherwise the two tags are different as shown in rule R_{91}:

$$\frac{\tau_1 \neq \tau_2}{(v : \_ +\!\!+ [\tau_1, \_], \_, \mathbf{return} : \_, (\_ +\!\!+ [\tau_2], e, c, d), h) \longmapsto ([v, \tau_1, \tau_2], e, \mathbf{return} : c, d, h)} \ R_{91}$$

where the state is discarded and the search continues with the dump. Using this mechanism, a **return** from a block may jump over intervening machine states in order to find the machine state of the method which created the block. This will be the machine state which is returned from.

Control flow mechanisms are not fundamental to object-oriented languages in the same sense as inheritance and environment mechanisms. However, many object-oriented languages provide interesting control flow features such as blocks in Smalltalk, **rescue** clauses in Eiffel and **try** statements in C++. These can all be implemented in the Λ-calculus using constructions based on program points.

Chapter 5

Conclusion

Our thesis is that the Λ-calculus provides the characteristic primitive features of object-oriented programming languages and can be used to represent and analyse all of the major programming constructs from such languages. As such, the Λ-calculus provides a unified framework within which existing constructs can be decomposed into their atomic components, constructs from different programming languages can co-exist, existing constructs can be generalised and completely new constructs can be invented.

The contribution of this work is that it unifies and extends many disparate sources of object-oriented language analysis (see §1.4) and represents them using a language which has been invented specially for the purpose. The proposed characteristic primitives are simple , in particular the invention of new special forms has been kept to a minimum with most of the proposed primitive features being defined as simple built-in operators. Chapter 4 shows that the Λ-calculus can represent object-oriented programming constructs which have not yet been analysed in the literature, *e.g.* feature 19.

The rest of this chapter identifies and analyses the major components of this thesis in more detail.

5.1 The Λ-calculus

Our thesis is that the Λ-calculus is a programming language which provides primitive features which are characteristic of a broad range of object-oriented programming languages. The Λ-calculus is described in chapter 3 by giving its concrete syntax as a BNF definition and its operational semantics as a state transition machine. The Λ-calculus enriches a simple functional programming language with the following primitive object-oriented features:

1. Sharing and update. The Λ-calculus provides operations which construct locations, produce the contents of locations, update locations and copy locations. The semantics of sharing and update is defined using a heap. Locations are used to build objects which have mutable attributes.

2. Fixed points. The Λ-calculus provides the paradoxical combinator Y which is defined as a built in operator and finds the fixed point of a function. Fixed points are used to construct self reference in objects. Fixed points also allow mutually recursive definitions which are used to define *run inner*.

3. Environment reification. The Λ-calculus provides R and I operators which access and update the environments found in closures. Objects in object-oriented languages are associations between names and values, possibly with extra structure, such as layered like an onion, associated with a tag set or paired with a class. Environments are the basis of all of these representations. Along with various environment constructors and mutators, these operators form the basis of an operational description of inheritance. All of the inheritance mechanisms which are supported by current object-oriented languages can be constructed using environment reification and this technique allows new forms of inheritance to be implemented and analysed.

4. Program points. The Λ-calculus provides the J operator which allows control abstractions to be constructed. J can be shown to be the basis of Smalltalk blocks, Eiffel rescue clauses and the C++ exception mechanism.

5. Tags. Class definitions in many languages may be viewed as a type where all instances of the class are identified with the type. Inheritance extends such a view to include a sub-type relation. Many features such as exports and inclusion polymorphism rely upon this view of classes as types and this is represented in the Λ-calculus using sets of tags and an equality relation on tags.

The main features are considered to be: environment reification, program locations, self reference and the operators which are basic to the object-oriented model of programming. These are described in the following sections.

5.2 Environments

Record structures are a standard component of programming languages, but it is only recently that a correspondence between records and identifier binding environments has been made via installation and reification. There is a close similarity between the rules of identifier scoping which are provided in many languages, giving rise to *lexical*, *dynamic*, *fluid*, *exported*, *imported*, *private*, *public* identifiers and the rules governing the visibility and access to names in object and class interfaces.

Given this correspondence, it is perhaps not surprising that objects and identifier binding environments turn out to be two sides of the same coin, where object-oriented mechanisms rely fundamentally on giving first class citizenship to those elements of a programming language which deal with issues of associating identifiers and values. Programming language implementations usually jump through considerable hoops to make the access to values associated with identifiers as efficient as possible. Given a description of such a language implementation, it may be difficult to see the correspondence between identifier bindings and objects. This is where an abstract description of a programming language, such as that used in this thesis, becomes useful since we are interested in capturing the essence of the behavioural characteristics of language components, without the restrictions imposed by efficiency issues.

Environment reification provides scope for *metalevel programming*, *i.e.* performing actions which affect the underlying evaluation rules of the language being acted upon. Since the λ-calculus is viewed as a universal tool for the analysis of programming language issues, we claim that the λ-calculus which has been enriched with environment reification provides a universal tool for the analysis of programming language issues dealing with the control of namespaces. Object-oriented languages certainly fall into this category, but so do other forms of programming dealing with modules, interfaces, module composition, separate compilation, module transformation, *etc.* The Λ-calculus therefore is not limited just to the analysis of object-oriented languages, and can be used to represent, for example, Ada packages and ML structures.

We claim that the environment structures and operations which are defined in this thesis, see chapter 3, are sufficient to represent many diverse namespace constructs such as identifier binding schemes, objects, modules, Ada packages, records, ML structures and functors, classes, *etc.* without resorting to new types of semantic value.

Given that so many diverse programming constructs can be unified simply by providing access to and operations over the underlying identifier environment, a question arises as to how such a facility can be provided in an implementation of a programming language where efficiency concerns are often a premium. It is desirable to replace program identifiers with locations as described in [26]. [38] reviews the literature in this area and suggests that there are a number of local transformations which an optimizing compiler may perform in order to "compile out" the search for the location of an identifier's value, these include: installation of known environments into closures, omission of unused bindings during environment capture, specialisations of abstractions over environments for known applications and clever runtime caching techniques for minimising the effort required for identifier lookup.

5.3 Program Locations

Side effects are present in the vast majority of currently used programming languages and object-oriented languages are no exception. Although we believe that the essence of object oriented-ness is captured by the notion of first class environments, sharing and side effects are typical features which must be present in any framework which claims to be a universal object-oriented representation. Furthermore, not all locations in programming languages are shareable and therefore affected by update, for example in many languages the value parts of identifier environments are shareable locations whereas individual character positions in strings are not shareable. Any representation which will be used for the analysis of object-oriented languages must be able to answer questions dealing with the whether or not locations during program execution are shareable and therefore affected by update. The representation must also be able to answer questions as to when locations are copied, how deep the copy goes and when they they are left sharing.

The Λ-calculus provides ML-style locations which are the only shareable locations. All non-shareable locations are deep copied during program execution and locations are only ever explicitly copied. Typically the attributes of an object are locations which are scoped over the methods and not available explicitly to a client of the object. During program execution, all occurrences of the object within the Λ-program state will share the same attribute locations.

Programming languages which provide shareable locations do not often provide them as first class values. This prevents the modification of certain aspects of the language from non-shareable to shareable and vice versa. For example, many object-oriented languages do not provide facilities for class definitions to be shared by all parts of the program which use them and then updated by side effect. The Λ-calculus provides facilities for these issues to be clearly defined and for different strategies to be intermixed.

5.4 Self Reference

Many programming languages allow a named definition to refer directly to itself. This includes type definitions and operation definitions, but rarely includes value definitions. Object-oriented languages provide a restricted form of value self reference since it is usual for objects to refer to themselves via a predetermined identifier, often *self*, *me* or *current*. As with many other features, it is usual for the underlying mechanisms by which the object refers to itself to be hidden from the programmer who is therefore denied the opportunity of constructing variations on the same theme.

The Λ-calculus provides the underlying primitive of all self referential definitions – Y. All of the currently available self reference mechanisms in these languages can be represented using Y to construct cyclic environments. The mechanisms differ with respect to the number of places where the cycles occur, whether or not the cycles are all identical, whether they share, whether the cycles are bound to a predetermined or programmer coined identifier, *etc.* Providing explicit access to the underlying primitive which constructs cyclic values allows the programmer to invent new variations on the self referential theme.

5.5 Object-Oriented Operators

Chapter 4 shows how the Λ-calculus is used to construct a series of object-oriented programming features. The features which are analysed include: objects, prototypes, message passing, single inheritance, dynamic inheritance, multiple inheritance, repeated inheritance, self reference, run super, run inner, renaming of attributes and methods, and inclusion polymorphism.

The implementations of object-oriented language features which are analysed in chapter 4 involve a collection of operators which repeatedly occur. The implementation of these operators depends upon the particular feature which is being supported. The repeated occurrence of these identifiers suggests that they are basic to the object-oriented model. The operators are briefly described below:

nullclass The empty class which defines no attributes or methods. This class is often called *object* in languages. It is usually the identity value for class composition operators.

inherit A binary operator which constructs classes from a super-class and an extension. The extension defines new attributes and methods which are added to those of the super-class to produce a new class. In the examples shown, a class is a function from a structured collection of initial values to an object. More sophisticated schemes can be supported by the Λ-calculus, for example the argument to a class can be a collection of keywords and values.

siblings A binary operator which constructs classes from a pair of super-classes.

extensions An extension defines attributes and methods. It is composed with a super-class using *inherit*.

new A binary operator which is applied to a class and a collection of initial attribute values and produces an instance of the class.

ref A binary operator which is applied to an object and a field name and returns the contents of the field.

set A tertiary operator which is applied to an object, a field name and a value. The value of the object's field is updated to contain the new value.

send A tertiary operator which is applied to an object, a name and a value and invokes the method of the object with the given name. Methods are functions with some hidden structure such as extra arguments or machinery to modify the actual parameter value.

5.6 Validity

Current object-oriented programming languages do not provide a precise semantic description and often such languages provide programming facilities which hide behind atomic linguistic abstractions which prevent the programmer from modifying the facility in order to exhibit the desired behaviour. This makes it difficult to decide exactly what features constitute an object-oriented language and to compare different languages in order to analyse the similarities and differences. By obscuring these facilities it is difficult for designers to generalise and modify them when producing a new language.

This work contributes to the area of programming language analysis and design by providing a unified framework within which existing object-oriented languages can be analysed and new languages designed. The framework has been given a precise semantic description using a state transition machine.

This work unifies many separate results relating, for example to mixins [22], denotational frameworks [23], environment reification [2] and operations on records [9]. The framework allows these results to co-exist within a single framework.

Chapter 4 describes many object-oriented programming features, using the Λ-calculus, which have not been analysed in the literature, for example repeated inheritance and multiple points of view for self reference have not be addressed before. Furthermore, the language features in chapter 4 use general operators such as *inherit* and *new* to implement a wide variety of different features.

The object-oriented features which have been analysed in this thesis are typical of currently used programming languages. These include Smalltalk (run super, single inheritance, self reference), Beta (run inner, single inheritance, nested classes, self reference, inclusion polymorphism), Eiffel (self reference, multiple inheritance, repeated inheritance, inclusion polymorphism), Self (prototypes, dynamic inheritance, point of view self reference), CLOS, Loops (run super, self reference), Trellis/Owl (self reference, single inheritance) and C++ (late binding, multiple inheritance, self reference, inclusion polymorphism).

Many of the features which have been analysed in this thesis have been implemented by the author in Common Lisp [67]. The implementations are not particularly efficient but prove that the descriptions are operational.

5.7 Further Work

Although many of the characteristic features of current object-oriented languages have been described in this thesis, there are many more features which have not. For example *meta-classes*, *(constrained) genericity*, *abstract classes*, *method combination*, etc. are all facilities available in many languages. It is intended that the work which is presented in this thesis could form the basis of a language design tool which allows the user to experiment with mixing various existing object-oriented features to see if they will co-exist and to design new features. There are a number of directions in which the work could be continued,

- Type systems have not been addressed at all. One way of analysing the type systems within an operational framework would be to add tags to classes (in much the same way as was done in the inclusion polymorphism description in §4.22) and to define a collection of primitive operations in terms of testing tags, merging tag sets, etc. In addition to the references given in chapter 1, the reader is directed to [25], [45] and [58].

- Indiscriminate use of some primitive operators, such as R and I, will make static analysis of Λ-calculus programs difficult. Although such an analysis is beyond the scope of this thesis, we envisage a system whereby these operators are packaged up into constructs which inform a static analyser, such as a type checker, of the names which are placed in/removed from scope.

- Multiple inheritance (MI) is a large and complex area which has not yet been fully explored. There are many possibilities as to the result of a class which inherits from more than one parent, as seen in the number of languages which support MI and which disagree on the outcome. The work presented in this thesis would be an ideal basis for the study of the operational aspects of MI.

- Using environment reification as the basis for inheritance does not give any indication as to the efficient implementation of different inheritance schemes. It should be possible to design clever compilation schemes which allow some of the power of environment reification whilst retaining a reasonable speed of program execution.

- Metaclasses allow part of the underlying object-oriented operational mechanism to be encoded as a program and therefore allows the program to modify its own execution. For example, the Smalltalk metaclass mechanism allows classes to tailor their own instantiation mechanisms. The author has shown in [15] that the Λ-calculus can support the metaclass mechanism of Smalltalk and generalise it in order to control more of the interpretive mechanism, including method lookup. The Λ-calculus and its operational semantics provide a highly suitable platform for experimenting in this area.

- Since the Λ-calculus provides unrestricted access to the underlying program environment and we claim that all program constructions which involve namespaces can be implemented using primitive operations which access and affect the environment, a direction of further work is to use the Λ-calculus to analyse constructs such as Ada packages and ML structures and functors.

- The object-oriented model of computation provides a suitable framework for concurrency. Objects may be viewed as autonomous units of computation which interact by sending both synchronous and asynchronous messages. The Λ-calculus is single threaded and therefore does not support concurrent computations. The Λ-calculus could be extended with primitives for constructing and manipulating multiple execution engines; this would provide a suitable basis for the analysis and design of concurrent object-oriented language features.

5.8 Further Features

Figure 2.3 describes a collection of object-oriented programming language features which are not analysed in chapter 4. We claim that the Λ-calculus is a convenient framework for the analysis of all object-oriented features and this section briefly describes how these features can be implemented.

Feature 24 refers to the different types of program structure which are supported by object-oriented programming languages. For example Eiffel is a collection of mutually recursive class definitions whereas C++ is a collection of sequential class, data and function definitions. The Λ-calculus provides **let**- and **letrec**-expressions and top level program transactions which may be used to implement a wide variety of program structures.

Feature 25 allows the programmer to control the sharing and copying which takes place due to repeated inheritance. C++ provides such a mechanism in the form of `virtual` inheritance and Eiffel shares repeatedly inherited features unless they are renamed. A class may be implemented in the Λ-calculus as a pair (e, g) where e is an environment mapping attribute names to tags and g is a function which generates objects. Repeated inheritance is implemented using a tag database as described in chapter 4 except that every attribute is associated with a tag. By default, repeatedly inheriting from a class will allocate only one program location for each attribute, however modifying the tag associated with an attribute in the first component of a class will cause separate program locations to be allocated.

Feature 26 allows the names of methods to be renamed when a class is inherited. Inclusion polymorphism allows an instance of a sub-class to be viewed as an instance of a super-class. When this occurs the renaming must be undone. This is achieved by implementing an object as a pair (e, t) where e is a method environment and t is a table describing the renamings which must be undone when the object is viewed as an instance of one of its super-classes.

Feature 27 allows classes to be parameterised to produce generic classes. The Λ-calculus supports this feature since classes are first class data values and may be returned from functions. Eiffel provides a mechanism referred to as *constrained genericity* whereby a class is parameterised with respect to another class. The class which is supplied as the parameter value is constrained to be a sub-class of a named class. The Λ-calculus can implement this mechanism by representing classes as pairs (t, g) where g is an instance generator and t is a table describing the class' super-classes.

Feature 28 allows the interpretive mechanisms of the language to be methods of (meta-)classes. The Λ-calculus can support this feature and examples are described in [15].

Feature 29 allows attributes and methods of a class to be declared of various categories. Each category behaves differently with respect to message passing and inheritance. An example of this feature are the `public`, `private` and `protected` categories of C++. This can be implemented in the Λ-calculus by representing attribute and method environments as environments *of environments*. Operators based on *inherit* and *send* will combine the environments in different ways.

Feature 30 allows shadowed methods and attributes to be referenced by including the path names relative to the class making the reference. This is supported by C++ which allows members to be referenced using `C::i` notation. The Λ-calculus can implement such a feature by including the name of the class in the keys contained in method and attribute environments.

Feature 31 implements name clashes due to inheritance by merging the values associated with the name. The Λ-calculus can support this feature by tagging all values in objects with their type: one of *class*, *method* or *value*. Object concatenation will check for name clashes and apply appropriate merging operations (*e.g. inherit* in the case of classes).

Feature 32 allows a multiply inheriting class to name the parent to which a message will be sent or resent. The Λ-calculus can implement this feature using a similar mechanism to that which establishes *super* and *inner* in chapter 4.

Feature 33 allows a class to dictate those classes which may legally reference the methods of its instances. The Λ-calculus can implement this feature by associating tags with classes and methods. The tags on a method will define those classes which may legally reference the method. The field reference operators are extended with a parameter describing the current *import tags i.e.* the tags defining the accessibility of names.

5.9 The Future of Object-Oriented Languages

It seems likely that object-oriented features will be absorbed into the mainstream of programming languages. There are many elements of object-oriented languages which can be seen as desirable, for example modularity, incremental extension, control over identifier scopes, dynamic binding, sub-types, but many of the current languages hide combinations of these facilities behind linguistic abstractions which are atomic. Haskell [36] [37] and Modula-3 [11] are examples of emerging languages where object-oriented facilities are merged with constructs from other fields of programming.

Bibliography

[1] Adams N. and Rees J. *Object-Oriented Programming in Scheme.* In Proceedings ACM Symposium on Lisp and Functional Programming, ACM, 1988, 277 – 288.

[2] Agha G. *The Structure and Semantics of Actor Languages.* In de Bakker J. W., de Roever W. P., Rozenberg G. (Eds.) Foundations of Object-Oriented Languages, In Proceedings REX School/Workshop, LNCS 489, Springer-Verlag, 1991, 1 – 59.

[3] America P. *Designing an Object-Oriented Programming Language with Behavioural Subtyping.* In de Bakker J. W., de Roever W. P., Rozenberg G. (Eds.) Foundations of Object-Oriented Languages, In Proceedings REX School/Workshop, LNCS 489, Springer-Verlag, 1991, 60 – 90.

[4] Bobrow D., DeMichiel L., Gabriel R., Keene S., Kiczales G. and Moon D. *Common Lisp Object System Specification.* ACM SIGPLAN Notices, 23, 9, ACM, 1988.

[5] Bobrow D. and Stefik M. *The Loops Manual.* Xerox Palo Alto Research Center, Knowledge Systems Area, KB-VLSI-81-13, 1981.

[6] Cointe P. *The ObjVLisp Kernel: a Reflective Lisp Architecture to define a Uniform Object-Oriented System.* In Maes P. and Nardi D. (Eds.) Meta-level architecture and reflection workshop on Metal-level Architecture and Reflection, North Holland Publishing Company, 1988, 155 – 176.

[7] Cardelli L. *A Semantics Of Multiple Inheritance.* In Kahn G., MacQueen D. and Plotkin G. (Eds.) Semantics of Data Types, LNCS 173, Springer-Verlag, 1984, 51 – 67.

[8] Cardelli L. and Wegner P. *On Understanding Types, Data Abstraction and Polymorphism.* ACM Computing Surveys, 17, 4, ACM, 1985, 480 – 521.

[9] Cardelli. L. and Mitchell J. *Operations on records.* Mathematical Structures in Computer Science, 1, 1, Cambridge University Press, 1991, 3 – 48.

[10] Cardelli L. *Typeful Programming.* In Neuhold E. and Paul M. (Eds.) Formal Description of Programming Concepts, Springer-Verlag, 1991.

[11] Cardelli L., Donahue J., Glassman L., Jordan M., Kalsow B. and Nelson G. *Modula-3 Definition.* ACM SIGPLAN Notices 27, 8, ACM, 1992, 15 – 42.

[12] Chambers C., Ungar D., Chang B-W and Holzle U. *Organizing Programs Without Classes.* Lisp and Symbolic Computation: An international Journal, 4, 3, Kluwer Academic Publishers, 1991, 37 – 56.

[13] Chambers C., Ungar D., Chang B-W and Holzle U. *Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF.* Lisp and Symbolic Computation: An international Journal, 4, 3, Kluwer Academic Publishers, 1991, 21 – 36.

[14] Clark A. N. *A Layered Object-Oriented Programming Language.* In Walkden (Ed.) GEC Journal of Research, 11, 3, The General Electric Company p.l.c., 1994, 173 – 180.

[15] Clark, A. N. *Metaclasses and Reflection in Smalltalk.* Unpublished Research Report, 1992.

[16] Clinger W. and Rees J. (Eds.) *Revised Report 3 on the Algorithmic Language Scheme.* ACM SIGPLAN Notices, 21, 12, ACM, 1986, 37 – 79.

[17] Cointe P. *Metaclasses are First Class: The ObjVLisp Model.* In Meyrowitz N. (Ed.) Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM, 1987, 156 – 167.

[18] Briot J-P. and Cointe P. *A Uniform Model for Object-Oriented Languages Using the Class Abstraction.* In McDermott J. (Ed.) Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann, 1987, 40 – 43.

[19] W. Cook. *A Proposal For Making Eiffel Type-safe.* The Computer Journal, 32, 4, Oxford University Press, 1989, 305 – 311.

[20] Cook W., Hill W. L. and Canning P. S. *Inheritance Is Not Subtyping.* In Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL), ACM, 1990, 125 – 135.

[21] Cook W. *A denotational Semantics Of Inheritance.* PhD Thesis, Brown University, 1989.

[22] Cook W. and Bracha G. *Mixin Based Inheritance.* In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM, 1990, 303 – 311.

[23] Cook W. *A Denotational Semantics Of Inheritance and its Correctness.* In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications, ACM, 1989, 433 – 443.

[24] Cook W. *Object-Oriented Programming Versus Abstract Data Types.* In de Bakker J. W., de Roever W. P., Rozenberg G. (Eds.) Foundations of Object-Oriented Languages, In Proceedings REX School/Workshop, LNCS 489, Springer-Verlag, 1991, 151 – 178.

[25] Danforth S. *Type Theories and Object-Oriented Programming.* ACM Computing Surveys, 20, 1, ACM, 1988, 29 – 72.

[26] De Bruijn N. G. *Lambda calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser Theorem.* Indagationes Mathematicae, 34, 1972, 381 – 392.

[27] Felleisen M. *On the Expressive Power of Programming Languages.* In Jones N. (Ed.) Proceedings of the European Symposium on Programming, LNCS 432, Springer-Verlag, 1990, 134 – 151.

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

[29] Gelernter D. *Environments As First Class Objects.* In Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL), ACM, 1987, 98 – 110.

[30] Ghelli G. *Modelling Features of object-oriented languages in second order functional languages with subtypes.* In de Bakker J. W., de Roever W. P., Rozenberg G. (Eds.) Foundations of Object-Oriented Languages, In Proceedings REX School/Workshop, LNCS 489, Springer-Verlag, 1991, 311 – 340.

[31] Goldberg A. and Robson D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[32] Hannan J. and Miller D. *From Operational Semantics to Abstract Machines: Preliminary Results.* In Wand M. (Ed.) Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, ACM, 1990, 323 – 332.

[33] Harper R. and Pierce B. *A Record Calculus Based on Symmetric Concatenation.* In Proceedings of the 18th ACM Symposium on Principles of Programming Languages (POPL), ACM, 1991, 131 – 142.

[34] Harper R., MacQueen D. and Milner R. *Standard ML* Edinburgh University, Laboratory for Foundations of Computer Science Research Report ECS-LFCS-86-2, 1986.

[35] Henson. M. C. *Elements of Functional Programming.* Blackwell Scientific Publications, 1987.

[36] Hudak P., Jones, S. P. and Wadler P. *Report on the programming language Haskell.* ACM SIGPLAN Notices, 27, 5, ACM, 1992, Section R.

[37] Hudak P. and Fasel J. *A Gentle Introduction to Haskell.* ACM SIGPLAN Notices, 27, 5, ACM, 1992, 1 – 53.

[38] Jagannathan S. *Metalevel Building Blocks for Modular Systems.* ACM Transactions on Programming Languages and Systems, 16, 3, ACM, 1994, 456 – 492.

[39] Jagannathan S. and Agha G. *A Reflective Model Of Inheritance.* In Lehrmann Madsen O. (Ed.) Proceedings of European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, 1992, 350 – 371.

[40] Kamin S. *Inheritance in Smalltalk-80: A Denotational Definition.* In Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL), ACM, 1988, 80 – 87.

[41] Kiczales G., des Rivières J. and Bobrow D. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[42] Kristensen B. *Abstraction Mechanisms In The Beta Programming Language.* In Proceedings of the 10th ACM Symposium on Principles of Programming Languages (POPL), ACM, 1983, 285 – 298.

[43] Kristensen B., Lehrmann Madsen O., Moller-Pedersen B. and Nygaard K. *The BETA Programming Language.* In Shriver B. and Wegner P. (Eds.) Research Directions In Object-Oriented Programming, The MIT Press, 1987, 7 – 48.

[44] Lieberman H. *Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems.* In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM, 1986, 214 – 223.

[45] Liskov. B. *Data Abstraction and Hierarchy.* ACM SIGPLAN Notices, 23, 5, ACM, 1988, 17 – 34.

[46] Landin P. *The Mechanical Evaluation of Expressions.* The Computer Journal, 6, 4, Oxford University Press, 1964, 308 – 320.

[47] Landin P. *The Next 700 Programming Languages.* Communications of the ACM, 9, 3, ACM, 1966, 157 – 166.

[48] Landin P. *A Correspondence between Algol 60 and Church's lambda notation: Part I.* Communications of the ACM, 8, 2, ACM, 1965, 89 – 101.

[49] Lehrmann Madsen O. *Block-Structure In Object-Oriented Languages.* ACM SIGPLAN Notices, 21, 10, ACM, 1986, 133 – 142.

[50] Lehrmann Madsen O. and Moller-Pedersen B. *Virtual classes a powerful mechanism in object oriented programming.* In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM, 1989, 397 – 406.

[51] Meyer B. *Genericity Vs. Inheritance.* In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM, 1986, 391 – 405.

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

[53] Meyer B. *Object-Oriented Software Construction.* Prentice Hall International Series in Computer Science, 1988.

[54] Miller J. and Rozas G. *Free Variables and First-Class Environments.* Lisp and Symbolic Computation, 4, Kluwer Academic Publishers, 1991, 107 – 141.

[55] Mugridge W., Hamer J. and Hosking, J. *Multi-Methods in a Statically-Typed Programming Language.* In America P. (Ed.) Proceedings of the European Conference on Object-oriented Programming (ECOOP), Springer-Verlag, 1991, 307 – 324.

[56] Nielson H. R. and Nielson F. *Semantics with Applications.* Wiley Professional Computing, 1992.

[57] Nygaard K. *Basic Concepts in Object-Oriented Programming.* ACM SIGPLAN Notices, 21, 10, ACM, 1986, 128 – 142.

[58] Palsberg J. and Schwartzbach M. *Type Substitution for Object-Oriented Programming.* In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA/ECOOP), ACM, 1990, 151 – 160.

[59] Peyton Jones S. *The Implementation of Functional Programming Languages.* Prentice Hall International, 1987.

[60] Plotkin G. D. *Call-by-name, call-by-value and the λ-calculus.* Theoretical Computer Science, 1, 2, 1975, 125 – 159.

[61] Reddy U. *Objects As Closures: Abstract Semantics Of Object-Oriented Languages.* In Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming, ACM, 1988, 289 – 297.

[62] Rentsch T. *Object-Oriented Programming.* ACM SIGPLAN Notices, 17, 9, ACM, 1982, 51 – 57.

[63] Saunders J. *A Survey of Object-Oriented Programming Languages.* The Journal of Object-Oriented Programming, 1, 1989.

[64] Schaffert C., Cooper T., Bullis B., Kilian M. and Wilpolt C. *An Introduction To Trellis/Owl.* In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM, 1986, 9 – 16.

[65] Schmidt, D. A. *Denotational Semantics.* UCB Wm. C. Brown Publishers, 1986.

[66] Snyder A. *Inheritance and the Development of Encapsulated Software Systems.* In Shriver B. and Wegner (Eds.) Research Directions In Object-Oriented Programming, The MIT Press, 1987, 564 – 578.

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

[68] Stein L. A. *Delegation Is Inheritance.* In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM, 1987, 138 – 146.

[69] Stoy J. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

[70] Stroustroup B. and Ellis M. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[71] Ungar D. and Smith R. *SELF: The Power of Simplicity.* In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA), ACM, 1987, 227 – 242.

[72] Wadler, P. *Comprehending Monads.* In Proceedings of the 19th Symposium on Lisp and Functional Programming, ACM, 1991, 61 – 78.

[73] Wegner P. *Inheritance As An Incremental Modification Mechanism.* In Gjessing S. and Nygaard K. (Eds.) Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 322, Springer-Verlag, 1988.

[74] Wegner P. *The Object-Oriented Classification Paradigm.* In Shriver B. and Wegner P. (Eds.) Research Directions In Object-Oriented Programming. The MIT Press, 1987.

[75] Wegner P. *Dimensions of Object Based Language Design.* In Meyrowitz N. (Ed.) ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications, ACM, 1987, 168 – 182.

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

[77] Wolczko M. *Semantics of Smalltalk-80.* In Bèzivin J. Hullot J-M., Cointe P. and Lieberman H. (Eds.) Proceedings of the European Conference on Object-Oriented Programming, LNCS 276, Springer-Verlag, 1987, 108 – 120.

[78] Wolczko M. *The Semantics of Object-Oriented Languages.* PhD Thesis, University of Manchester, 1988, UMCS-88-6-1.