A MULTIMETHOD-BASED ORTHOGONALLY PERSISTENT PROGRAMMING LANGUAGE

Antonio Cunei

A dissertation submitted in partial fulfillment of the requirements for the Degree of M.Sc. in Advanced Information Systems at the University of Glasgow.

SEPTEMBER 2000

Acknowledgements

I would like to thank my supervisor, Prof. Malcom Atkinson, and the coodinator of the course of M.Sc. in Advanced Information Systems, Dr. Richard Cooper. I would also like to thank for their great work all those who spent their time end effort preparing all the aspects of the M.Sc. in AIS course, selecting always the most updated and interesting material and teaching with passion and enthusiasm.

I am grateful to all the friends and people that I met in Glasgow for their support and friendship, and in particular I would like to thank Tony Printezis and Huw Evans for their precious suggestions, and Chantelle MacPhee for her invaluable help.

This work has been made possible by a studentship offered by the Engineering and Physical Sciences Research Council.

Abstract

Orthogonal persistence allows data to persist for an indefinite amount of time, and to be manipulated in a uniform, consistent way regardless of its type. Despite the large amount of research on orthogonal persistence and on programming languages, to the author's knowledge no programming language based on multimethods has ever been integrated with orthogonal persistence, offering an implementation capable of recovery. In this work, a language definition which offers support for multimethods in the context of static and strong typing will be introduced, and it will be shown, with a concrete implementation, how an orthogonally persistent version of that language can be created. The working implementation, which relies on technology developed in the University of Glasgow, gives full support to orthogonal persistence, and the execution of the generated programs can be resumed after an unexpected interruption.

Contents

List of Figures

1. Introduction

Writing computer programs is, in general, a rather complex activity. Often large amounts of data and code have to be managed, and many techniques have been developed to structure both in ways that could simplify and make more efficient both the programming activity and the treatment of data. The introduction of object oriented programming, for instance, has been, for many, a turning point in the evolution of design and implementation of computer systems, changing at the core the way in which computer programs are written and data is organised.

Among the many technologies that have been introduced over time to simplify the manipulation of data, a very interesting one is orthogonal persistence[AM95][ABC+83]. The main idea which is promoted by the technique is that all data should be treated in a single, uniform manner and should be allowed to persist in the system for an indefinite amount of time, regardless of its type. That means that the programmer does not need to deal with separate models for the treatment of data in memory, on mass storage and so on. Furthermore, the semantic of operations is unique, regardless of the specific kind of data manipulated. The obvious result is a great simplification of program structure, and less burden for the programmer. An additional advantage is that, since the actual transfer of data between memory and mass storage is completely demanded to the system, it is possible to organise, in a transparent way, the migration of data in a distributed environment. Since all data movement is under system control, it becomes easier to implement a recoverable system, which offers protection against unexpected system failures, power losses and so on. Many orthogonally persistent systems have been developed, mainly programming languages [ADJ+96], but also entire operating systems [DdBF+94a][VRH93]. One of the most natural ways to support orthogonal persistence is to use objects, which offer a simple way for the system, as well as for the user, to manipulate units of data. If objects are used, the best way to match the requirement, imposed by orthogonal persistence, that all data is handled in a uniform way, is to use a “pure” object oriented language, in which all data is treated uniformly as objects. The question remains which other technologies or mechanisms, used in other object oriented languages, can be included in an orthogonally persistent language.

The use of multimethods will be here proposed for inclusion among the characteristics that can be integrated in an orthogonally persistent language. Multimethods represent a natural extension of the common message dispatching technique, used in conventional object oriented languages, to more than one receiver. The idea is not new, and it is mainly known for its inclusion in the Common Lisp Object System (CLOS[Clo][Dal97]); however, multimethods have been used mainly in dynamically typed languages, and their use in statically typed languages has remained marginal. Among the few which offer static typing are Cecil[Cha93], BeCecil[CL96], Dylan[App95], Kea[MHH91], Polyglot[ADL91] and Tigukat[Leo99]. To the author's knowledge, only Cecil and Dylan have a working compiler, and no multimethod-based language have been integrated with orthogonal persistence, offering an implementation capable of recovery.

In this report, a language definition which offers support for multimethods in the context of static typing will be introduced, and it will be shown, with a concrete implementation, how an orthogonally persistent version of that language can be actually created. The working implementation, which relies on technology developed in the University of Glasgow, gives full support to orthogonal persistence, and the execution of the generated programs can be resumed after an unexpected interruption.

In order to follow the content of the discussion, the reader should have a fair knowledge of use and implementation of object oriented languages and a general idea of the techniques used to implement orthogonally persistent systems. Many useful references are available in literature on both subjects[AMB95][Bea94][PC93]. Some previous knowledge of multimethods is advantageous, although the main ideas and the techniques used in the definition of the language are illustrated in Chapter 2.

1.1. The Project

The main purpose of the project is to show that multimethods, in the context of a statically typed language, are one of the elements that can be included in an orthogonally persistent language. A language based on multimethods and statically typed was already designed and implemented by the author, but, despite the fact that orthogonal persistence was one of the main ideas behind the language design, the test implementation was entirely memory based, and did not offer support for recovery. On the other hand, a versatile and highly efficient general purpose object store, Sphere[PAD98], has been developed at the University of Glasgow, and has been used extensively to support the PJama system[AJ99]. The logical step to obtain a multimethod based, orthogonally persistent language, was therefore to bring together the two elements, writing suitable additional code. The resulting system is a compiler,

with the related runtime support, for a multimethod based, strongly and statically typed language which integrates orthogonal persistence and whose running programs are capable of resuming execution if interrupted for whatever reason. A summary of the previously existing software components follows.

1.1.1. The Language Compiler

The preexisting implementation of the language compiler is composed by a set of programs written in C/C++, with the help of the well-known utilities lex and yacc. The compiler produces, as target code, a source file written in C which can be compiled by GCC to produce the final executable. The choice of the intermediate use of the C language was primarily due to the need of reducing the total development time and enhancing the portability of the test implementation of the compiler. In particular, the high-level features of the C language are not used, and the translation process has a logic similar to the one needed to produce low level assembler code.

The resulting programs are memory based, and the Boehm-Demers-Weiser conservative garbage collector[Boe93] is used to dispose the unused memory automatically. The implementation showed the viability of the overall architecture of the language, although it did not offer any information about the aspects related to orthogonal persistence and recoverability.

1.1.2. Sphere

Sphere is a high performance, general purpose, recoverable object store developed at the University of Glasgow. Its recovery mechanism is based on the ARIES[MHL+92] write-ahead logging technique, and the size of the store managed can be considerably large – Sphere has been used in real-life bioinformatic applications to handle amounts of data as large as 5GBytes. The system uses an efficient object promotion algorithm which allows to obtain high performances during the checkpoint operation. It includes support for evolution and is designed to accomodate different strategies for the storage of objects as compacting, compressing etc. Although Sphere has been used primarily in the implementation of the PJama programming environment, its design offers general purpose support for orthogonal persistence, and useful primitives are available to implement efficiently all the common operations required by a typical orthogonally persistent system.

1.2. Overview

The work describes two main aspects: the language definition and the newly written interface with the store. The first part is covered by Chapters 2 and 3. Chapter 2 describes the rationale behind the language definition, and in particular how multimethods can be used in a statically typed context and the techniques that have been used in the construction of the type system for the language. Chapter 3 discusses the language definition, offering useful hints about how such a language can be implemented efficiently.

The second part, the way in which the compiler has been adapted to the object store, is described in Chapters 4 and 5. Chapter 4 offers an overview of the design process of the interface layer between the compiler and the store, and the way in which the existing compiler had to be changed. Chapter 5 reviews at a high level the implementation of that intermediate layer and the modifications that were required by the compiler, that is the work that has been done more recently. The description of the detailed way in which the compiler was implemented has been intentionally left out, since the related work was done in another context. Chapter 6 concludes the dissertation with a summary the results obtained and possible future developments.

2. Multimethods and Static Typing

The main purpose of this project is to show how multimethods can be integrated seamlessly at the core level in the definition of an orthogonally persistent language. A short review of multimethods and static typing, together with the approach that has been followed during the definition of the language, will be therefore presented during this chapter. The analysis of multimethods and their implication will be not exaustive, neither will it discuss all the aspects related to the issue from a formal point of view. The aim of the discussion will be, instead, to give a general idea of the behaviour of multimethods when used in typed languages, and why they can be useful. In the chapter, the notation "$fun(C_1, C_2, \ldots, C_n) : C$" will be used to refer to a function whose return value has type C, and whose parameters have type $C_1, \ldots, C_n$. This concise notation is similar to that used for function protypes in ANSI C [KR89].

2.1. Overloading

In traditional programming languages, a technique commonly used to simplify the use of operations conceptually similar is to "overload" symbols. This refers to the possibility, given to programmers, to use a single identifier to refer to more than one function or procedure, which differ in the number and/or type of their parameters. In most languages, the symbol "+" is used to refer both to the sum of two integers and to the sum of two floating point numbers. In that sense the symbol "+" is "overloaded" with more actual meanings, representing different operations if used with different parameters. The detection which, among all the available functions, will be invoked in a specific case is performed in this case statically, at compilation type, checking which, among the function defined using that identifier, matches exactly the number and type of the parameters used in that specific function call.

With the advent of object-oriented programming this way of proceeding has been retained, and the use of overloading is found more or less unaltered in many object oriented typed languages as C++, Java and others. However, the new type matching definition commonly used in those languages (inclusive polymorphism) makes the effects of overloading, in this context, much less intuitive.

2.1.1. Overloading vs. Dispatching

In this example, we want to implement two classes, called “parent” and “child”, and a few methods using Java.

```
class child extends parent {}

class parent
{
 void direct(parent b) {
  System.out.println(" :  Called parent - parent");
 }

 void direct(child b) {
  System.out.println(" :  Called parent - child");
 }

 void indirect(parent b) {
  direct(b);
 }

 public static void main(String av[])
 {
  parent p=new parent();
  child  c=new child();

  System.out.print(p); System.out.print(p); p.direct(p);
  System.out.print(p); System.out.print(c); p.direct(c);
  System.out.print(p); System.out.print(c); p.indirect(c);
 }
}
```

Here, two messages are defined, `direct` and `indirect`, which can be sent to instances of the class `parent`. This is the output of the program:

```
parent@80caf4a parent@80caf4a :  Called parent - parent
parent@80caf4a child@80caf4c :  Called parent - child
parent@80caf4a child@80caf4c :  Called parent - parent
```

In this example, the first two calls have, as a result, the execution of the most specific method among those available. This conforms to the principle commonly used in object oriented programming, in which it is possible to redefine part of the behaviour of a class while defining a subclass, in order to obtain a description suitable for a specific case from

the more generic one. However, the result of the third call shows how the overloading can interfere with this principle, causing behaviours that are difficult to forecast.

What happens is that, with the introduction of objects and classes, it is now possible that more than one definition can match a single invocation, due to the very nature of inclusive polymorphism. Since every instance of "child" is also, in a way, an instance of "parent", having a different list of types of parameters for each definition is no longer enough to select a unique one for every invocation. Most statically typed object oriented languages work at this point by reusing the class hierarchy to search for the most specific call applicable, but the selection is done, in this case, only statically.[1] In the second call, the second method definition is recognised during compilation as being more specific, and therefore selected as the best choice. In the third, however, the argument of the only available definition of "indirect" is always "parent", and the compiler selects therefore the first method ("parent-parent") in any case. In other terms, when passing through the "indirect" call the instance `c` loses, somehow, part of its identity and is reduced to a more generic class. The method called will therefore be the more generic one, whose implementation can differ significantly from the more specific one. Clearly, to keep track of this sort of behaviour can be difficult for the programmer, and, as a result, unconsistent operations or unexpected problems could surface.

2.2. Generic Functions

An alternative approach is made possible by using multimethods, which are essentially nothing more than the extension to more then one receiver of the usual message/method mechanism. The idea is to select the method not just considering a single receiver, but using a tuple of receivers to select dynamically the most specific method for a specific invocation. This may also help to model, more accurately, situations in which the use of a single receiver might appear not to fit completely.

Considering the arithmetic sum in some pure object oriented languages (Smalltalk, for instance), we find that the typical implementation defines the sum as a message to be sent to one of the addends, with the second one as a parameter. There is a forced asymmetry in the operation, due to the need of having, in any case, a single receiver; whereas, a more intuitive way of proceeding would be to treat both addends equally, since they both participate in the same way to the actual sum. In cases like this one, multimethods may allow a more immediate computer representation of the reality we want to describe. The messages which

[1] Since methods having different types for their parameters are selected statically, they are actually treated by all means by the compiler as being part of separate messages, and will be grouped accordingly for the purposes of message dispatching.

correspond to a certain number of multimethods are often called “generic functions”, to emphasise the fact that they look like “functions” which behave differently according to the type of their parameters.

To show the different behaviour of the code in case multimethods are used, here is the same example used previously, rewritten using the language that we are about to introduce. Although the language syntax will be described in detail only later, its similarity with the usual C++/Java style and the previous description of multimethods should make the code easily understandable.

```
GenericVsOverload:  uses system
{
!parent:  super object {!parent():  super object() {}}
!child:  super parent {!child():  super parent() {}}

direct(a:parent,b:parent)
{ println(" :  Called parent - parent"); }

direct(a:parent,b:child)
{ println(" :  Called parent - child"); }

indirect(a:parent,b:parent)
{ direct(a,b); }

main()
{
 p:=parent();
 c:=child();

 p.print; p.print; p.direct(p);
 p.print; c.print; p.direct(c);
 p.print; c.print; p.indirect(c);
}
}
```

The program output, obtained using the compiler described in this work, is the following:

```
<parent><parent> :  Called parent - parent
<parent><child> :  Called parent - child
<parent><child> :  Called parent - child
```

In this case, the result of the indirect invocation is identical to the direct one. In both cases the most specific method available is called. The behaviour is homogeneous and there is no longer trace of the unconvenient asymmetries due to the static nature of overloading. According to this example, it is clear that multimethods can represent a very appealing alternative to the traditional use of overloading in object oriented typed languages, offering a versatile and intuitive instrument, suitable to model a range of problems that cannot be fully described using messages restricted to a single receiver.

2.3. Multimethods in Typed Languages

In this section some ideas, similar to many others commonly found in literature, are presented with the purpose of ensuring that the set of multimethod definitions in a program are consistent, and that no runtime error during message dispatching can ever occur, whatever the class of the message parameters. Other suggestions are then made on avoiding ambiguities while using multiple inheritance. The discussion is mostly informal, and its main purpose is to introduce to the techniques used in the language. As a first step we shall describe how to decide when a multimethod is "more specific" than another.

2.3.1. The Multimethods Graph

If inclusive polymorphism is used, an instance of a subclass of a given class *C* can be used throughout the program whenever a variable is expected to be an instance of *C*. This leads, for any occurrence of a variable or an expression, to the definition of two different types: a *static type*, which can be determined at compile time as the most generic type that an expression is expected to have at that occurrence, and a *dynamic type*, which is known only at execution time, which is the actual type of the value. The static type is used to perform the static type checking, while the second is used to perform dynamically the message dispatching.

In the case of messages with a single receiver, the set of possible method definitions for a given message matches the set of the classes, and the related hierarchy. If the class *B* is subclass of *A*, a method defined in *B* will be "more specific", since it can be used only for instances of *B*, while a method defined in *A* can be used both for instances of *A* and *B*. The relation "is more specific", applied to possible method definitions, is isomorphic to the relation "is subclass of" applied to classes.

With multimethods, for any given message of arity *n* there could be a definition of a multimethod in correspondence of each tuple of *n* classes, suggesting immediately the structure of the set to be used for the relation "more specific than," applied to multimethods. The

intuitive concept behind the relation can be described more formally as follows:

- Given two n-tuple $\widetilde{A} = (A_1, A_2, \ldots, A_n)$ and $\widetilde{B} = (B_1, B_2, \ldots, B_n)$, then $(\widetilde{A}, \widetilde{B})$ belongs to the relation if there is a j in $\{1, \ldots, n\}$ such that for every $i = 1, \ldots, n$, with $i \neq j$, A_i is equal to B_i, while A_j is a direct subclass of B_j, that is (A_j, B_j) belongs to the relation "is an immediate subclass of."

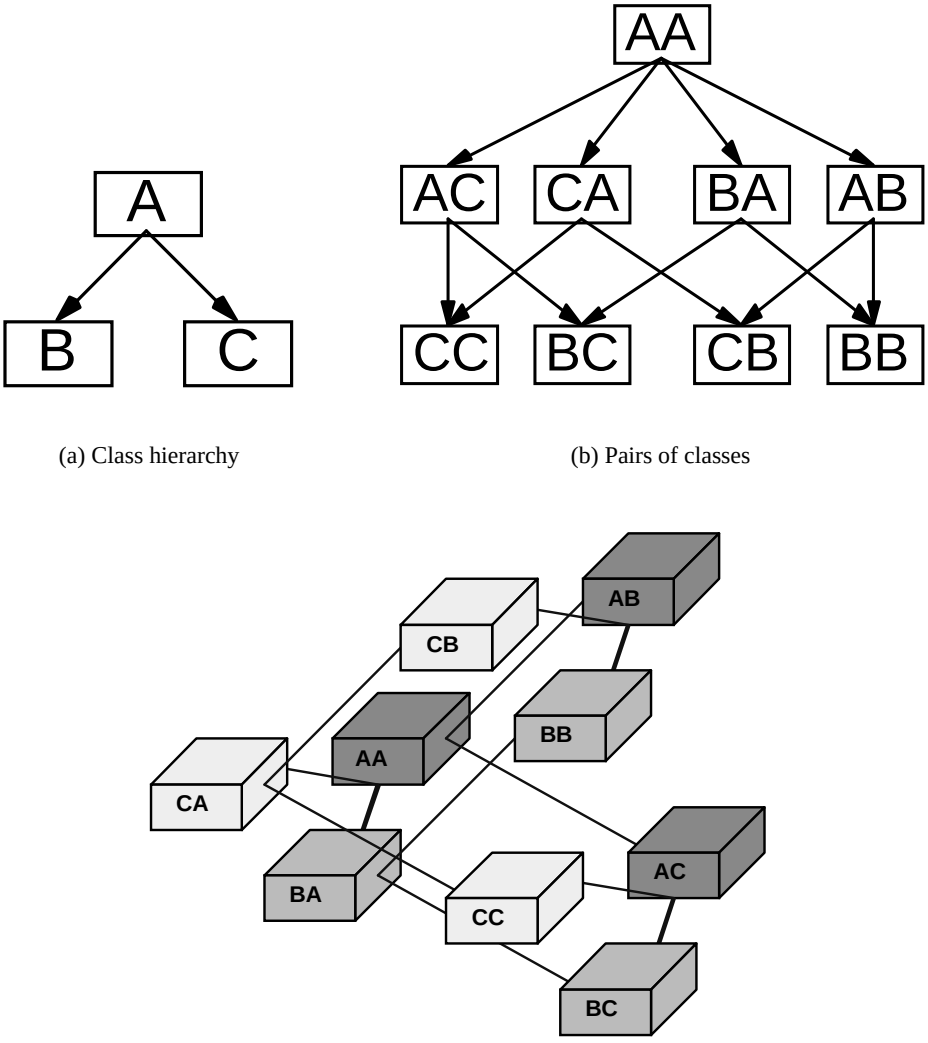
This relation of "greater direct specificity" induces a partial order relation on the n-tuples of classes, which can be used to determine whether an n-tuple is more specific than another or not, either directly or indirectly. A look at the example shown in Figure 2.1 will help clarify the way the construction works [the direction of the arrows is inverted with respect to the relations here described]. It is clearly a straightforward generalisation on multiple dimensions of the usual class. It has to be noted that if the original class graph was connected, the derived one will be connected as well, and that if the former was not containing any cycles, the latter will be similarly without cycles.

As is apparent from this example, even for very simple class hierarchies the derived graph can contain multiple paths which connect two n-tuples of classes. This may lead to a series of ambiguity problems not dissimilar to those found in all object oriented languages when multiple inheritance is introduced. Some techniques useful to avoid ambiguities in methods definitions will be detailed in the following section.

The simple construction shown allows us to perform a static typechecking even when multimethods are used. For example, let us assume we have a class *A* and one of its subclasses *B*, and the following multimethods are defined:

```
fun(B,A)
fun(B,B)
```

In this case, if a call appears statically as `fun(B,A)`, we know that there will be, in runtime, at least one suitable multimethod for every combination of types of the parameters. On the other hand, if the compiler encounters a call like `fun(A,B)`, there is no guarantee that a suitable method will be found during the execution, and the user can be therefore informed, during the compilation phase, that a runtime error might occur – clearly a simple extension of the basic technique commonly used by typed object oriented languages when a single receiver is allowed.



(a) Class hierarchy

(b) Pairs of classes

(c) Tridimensional view

Figure 2.1.: Extension on more dimensions of the class hierarchy

2.3.2. Resolving Ambiguities

The graph presented in the previous section allows the compiler to make sure that, whatever the combination of types of parameters used for any given call, there will always be, during execution, at least one suitable method. However, this is not enough to guarantee that the most specific method applicable is unique. For instance, let us consider again the class *A* and its subclass *B*, this time with the following method definitions:

```
fun(B,A)
fun(A,B)
```

A call which appears statically like `fun(B,B)` could be dispatched to both methods, since neither is more specific than the other. It is clearly a case of ambiguity in the definitions. It would be like defining two recipes to prepare a cake, one using strawberries and a generic cream, and the other using generic fruit and custard cream. In the case in which we have both strawberries and custard cream, none of the two recipies would be preferable a priori. More generally, we have an ambiguity every time that, given a message, there is an n-tuple for which there is no method defined, which is also simultaneously descendant from two or more unrelated n-tuples, each with a different method defined. The ambiguity extends, then, to all the descending n-tuples. To overcome the problem, it is therefore enough to detect the "more general" tuple where a conflict is present, and impose, using an ad-hoc rule, that a method definition must be present on that tuple as well. If multiple inheritance is not present, in fact, it is easy to realise that for every set of methods which originate an ambiguity there is one and only one of such "critical" tuples; each of its components is, for every index i, the last subclass, with respect to the hierarchical relation among classes, of the set $A_i = \{C_{j,i}\}$, where $C_{j,i}$ is the class of the formal parameter i of the method j. If the methods are conflicting each set A_i must be in fact totally ordered. As a consequence, the following rule allows us to detect ambiguities in the method definitions when single inheritance is used, and, more importantly, to suggest to the user which steps to adopt to fix the problem.

- For every subset of the set of methods defined for a message (with the same identifier and arity), whose parameters are of types $(C_{j,1}, C_{j,2}, \ldots, C_{j,n})$, IF for every $i = 1, \ldots, n$, the set $\{C_{j,i}\}$ is totally ordered with respect to the hierarchical reletion on classes, THEN a method must be defined with parameters $(\widetilde{C_1}, \widetilde{C_2}, \ldots, \widetilde{C_n})$, where $\widetilde{C_j}$ is the last element of $\{C_{j,i}\}$.

Interestingly, the same rule can be equally useful in both the static and the dynamic case. From a dynamic point of view, it ensures that every call will have a single most specific method to use; from a static point of view, it allows us to determine that a single most generic method, among the set of those defined for a message, will be usable for every call. That analysis, performed at compilation-time, can then be used to assign inductively a static type not just to variables and parameters, but to arbitrary expressions.

2.3.3. Return Values and Expressions

As we have seen, using the rule previously defined, it is possible to determine, for each call, a single most generic method definition which can be then used to perform static type checking. The next step is to determine what should happen to the returned value, and if it is possible to obtain some information on its static type. In general, in fact, every method among those usable for a specific call could have a different definition of the type of its return value, which would prevent us from statically determining a single limitation of the return type of the function call. Fortunately, the uncertainty can be eliminated using the following rule (covariance):

- For every pair of method definitions for same message $fun(A_1, A_2, \ldots, A_n) : A$ and $fun(B_1, B_2, \ldots, B_n) : B$, IF for every $i = 1, \ldots, n$ A_i is equal or a subclass of B_i, THEN A must be equal or a subclass of B.

The rule follows the intuitive behaviour that can be expected from messages: if a message operates on $A_1, \ldots, A_n$ returning A, it is reasonable to expect that all the matching methods will work in the same way. Since every instance of B_1 "is" (according to inclusive polymorphism) an instance of A_1, and similarly for the other parameters, all the methods should return an instance belonging to class A (or a subclass) for all the parameters of class $A_1, \ldots, A_n$ (or their subclasses). Consequently, if two methods had the same sequence of types for their parameters, their return type should be the same. Since we want to be able to choose a single method for every combination of parameters, it is reasonable to request that a single method can be defined for every message for any given sequence of types of parameters.

Using the rule described, it is possible to use the type of the return value of the single most generic method determined statically for every call as a limitation of the type for all the return values which can be returned from that or more specific methods at run time. Furthermore, a static type can be now determined inductively for every expression, however complex. An additional advantage is that every program extension obtained by defined other methods, will have to conform to the same rule, and will therefore leave the most generic type previously determined for every call unaltered, preserving the validity of the existing code.

In comparison, in Java every message must have a single return type, which can be a substantial limitation. For instance, a message used to build a list with the elements in the reverse order, given a list as an input, will always return a generic list, even if the argument is a list of integers. This behaviour may lead to an excessive use of typecasts, which can be a source of runtime exceptions. Conversely, using the mechanism described, it is possible to

define a method which returns a list of integers given a list of integers, a list of strings given a list of strings and so on, and a more specific return type can be determined statically.[2]

The set of rules, however, needs to be slightly enhanced if object constructors are used. The specific details are available in Section 3.14.1.

2.4. Multiple Inheritance

Multiple inheritance is one of the most controversial features in object oriented languages – many believe that its adoption is an endless source of problems, others say that the introduction of multiple inheritance in a language is like "opening a can of worms"[Mar93]. In many languages it has not been introduced (Smalltalk) or it has been replaced by alternative techniques (Java); in many others (C++,CLOS) it is used in conjunction with exoteric and complex rules. We shall briefly review the problems connected to multiple inheritance, and simple techniques to keep its side effects under control will be suggested.

The problems related to multiple inheritance are essentially of two categories: those related to methods and those related to instance variables.

2.4.1. Instance Variables

Those languages that allow the visibility from outside of instance variables can run into trouble if two different variables defined in two separate classes have the same name. If a new class is defined inheriting from both classes, a reference to a variables with that name is ambiguous. Depending on the design choices, it could happen that one of the two variables hides the other, or that both are available through two differently qualified names.

A similar issue arises when an instance variable defined in a superclass is reachable through multiple paths in the class graph. The variable could be inherited once or in multiple copies, one for every possible path. In C++, for example, unless the keyword "virtual" is used, the instance variables can be actually inherited multiple times[Eck]. This can be regarded as a violation of the general object oriented philosophy. If we want to model the class of animals, we can create a variable instance containing the number of legs. Deriving from that class two classes "domestic animals" and "felines", and then "cat" inheriting from the last two, the variable "number of legs" could be inherited twice. That would mean that a cat can have two different numbers of legs, depending on if it is considered to be a domestic animal or a feline, which is clearly somewhat bizarre.

[2] The language offers also the possibility to link together the type of a parameter with the type of the return value. This allows to define a *single* method, and still having the ability to discover statically more information about the type of the return value. More details in Section 3.12.

2.4.2. Inheriting Methods

A second category of problems concerns the way methods are inherited from superclasses. In the case in which a method is defined with the same name and parameters in two separate classes, and a third class is made a subclass of both, a message sent to an instance of the third class could use either of the two, originating an ambiguity. Both C++ and CLOS, in this case, give preference to the class declared first in the list of superclasses. This technique, however, has some major disadvantages: the programmer must keep in mind which ones will be the preferred classes, which, especially if complex dependencies are used, can be quite difficult. In the second stance, preferring one of the classes to another, using arbitrary rules, could conceal logical inconsistencies in the class model, which could lead to problems and inconsistent program behaviours.

2.4.3. Resolving Ambiguities

There are many possible options to solve the issues related to instance variables. A possible solution is simply not to allow the instance variables declared within a class to be visible from outside the class definition. This approach also improves encapsulation, and forces the actual implementation to be completely "opaque," hiding its internal details from the other classes. Since instance variables are not visible from outside the class definition, every class of the hierarchy will know only about "its" components of the objects, and a class derived from more superclasses is forced to access the internal variables of each exclusively through messages. For what concerns variables inherited through multiple paths, the more straightforward choice is clearly to keep a single copy of each, following the more intuitive interpretation.

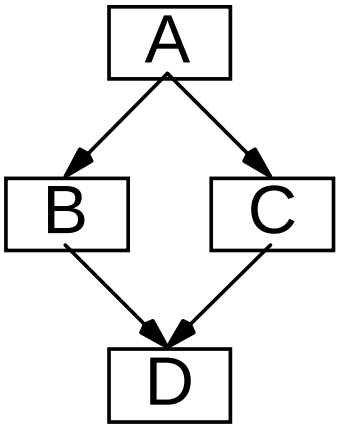


Figure 2.2.: Multiple inheritance

Dealing with method inheritance is, in general, not so simple. Referring to the example in Figure 2.2, we can distinguish several different cases.

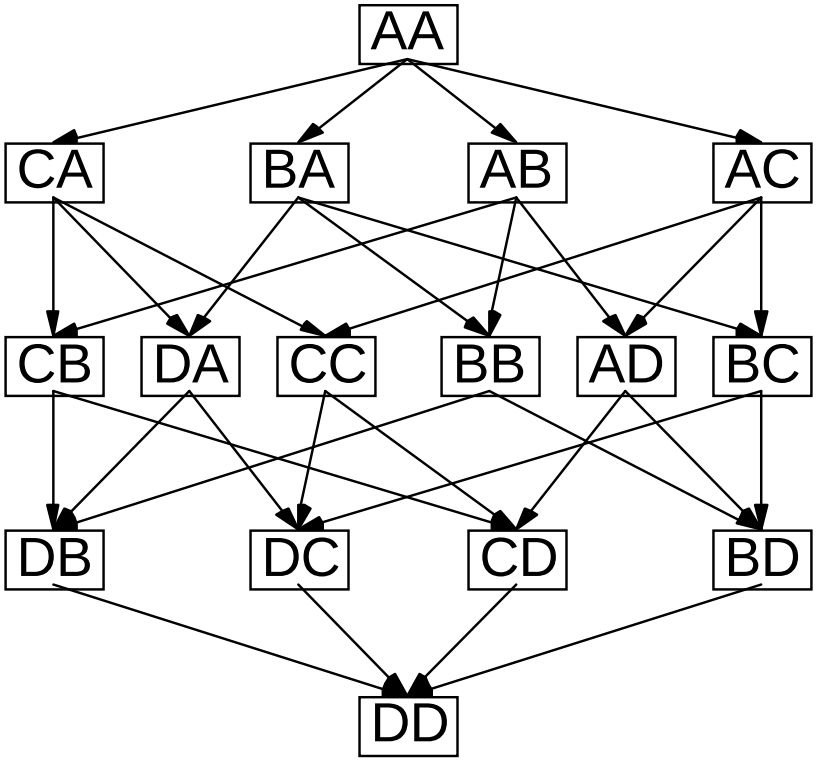
1. If there are two definitions fun(B) and fun(C), there is a conflict in D. As we saw, using explicit rules can be counterproductive. An alternative and more simple approach would be to detect statically these conditions and request an explicit resolution by the programmer, defining fun(D).

2. If there are two definitions fun(A) and fun(B), the case of D can be considered under two different perspectives. In the first one, D inherits two different methods through two different paths, and a conflict is therefore present. In the second one, the definition in B is actually more specific than fun(A), and no conflict is present. Both interpretations are equally viable, but the first is rather strict, whereas the second, more relaxed one, considerd the situation not to be ambiguous. Usually the second one is preferred, since the first generates just too many conflicts, and it may be useless in practice.

In the case of both multiple inheritance and multimethods, things might appear to be rather complex. In reality, there are many similarities between the two, as can be seen comparing the previous diagram with the one in Figure 2.3.

Every tuple that is the destination of other tuples using the defined relation on the constructed graph is actually a specialisation of all them. Whenever there is a discordance among the methods inherited by those tuples there will be an ambiguity. For instance,the pair DB is a specialisation of CB, DA and BB. If two methods inherited by those are in conflict, and there is no local definition, a potential problem is present. If we have two definitions in AB and CA, we will obtain that BB inherits from AB, CB from CA and, when there is the need to establish the more specific method applicable for DB, there is no way to perform a unique choice.

To summarise, by analysing the graph it is possible to detect all the possible conditions of ambiguity, both originated from multimethods and from multiple inheritance and, for every possible conflict, to propose to the programmer how to operate to fix the problem. The analysis can be simplified by the fact that the original class graph does not contain cycles, hence the derived one is also free from cycles and can be linearised. An exaustive examination of the entire graph is also unnecessary, since what matters is only the relationship between the method definitions and their common descendants. Multiple inheritance is not implemented in the current version of the compiler.



(a) Pairs of classes

Figure 2.3.: Extension of the class graph: multiple inheritance.

3. The Language

The ideas exposed in the previous chapter have led to the definition of a simple experimental language, which is based on multimethods, is strongly and statically typed and is well suited for orthogonal persistence. The language defined, which was dubbed with the codename "BOH" (Basic Object Handler), has a working (although not optimised) compiler which produces code capable of recovery. Among its many features is a syntax similar to the one used in C++ and Java, a structure that allows the language to be a "pure" object oriented language while offering techiques to avoid wrapper objects in its implementation, user-defined operators and strong encapsulation. The language itself is not particularly sophisticated, neither is its definition formal. It is just a way to show how a simple multimethod based, statically and strongly typed and orthogonally persistent language can be defined.

3.1. Introduction to the Language

A program written in the language consists in one or more "packages," each of which defines one or more resources in terms of classes and messages. Every package includes zero or more class definitions, global methods and global variables, which will be described soon.[1] Global variables can be used exclusively inside the package in which they are defined, and cannot be exported.

Every class definition may contains zero or more definitions of instance variables, and a number of methods. Since multimethods are used, the association between methods and classes is not unique, as it happens in languages that allow a single receiver. Nonetheless, as a design choice, only those methods defined inside a class body will have access to the internal structure of instances of that class. Instance variables are not visible from outside the class definition. This has positive implications on strict encapsulation, as described in Section 3.7, logically grouping multimethods in separate implementations of the different classes. Global methods, defined outside class definitions, are used to describe generic operations,

[1] The current implementation is restricted to a single package and offers no support for global variables, although for the latter the missing code is a trivial addition to the existing compiler.

not strictly associated with any particular class. As such, they are not allowed to access any instance variable, but can perform their operations using messages, as usual.

As already mentioned, in this description we will use sometimes the term “type” referring to the class of an instance, and the term “field” will sometimes be used to refer to instance variables. To introduce the syntax of the language, the next section will present some examples of class definitions. The following sections will introduce the aspects related to creation and initialisation of variables, standard types, declaration of instance variables, control structures and some syntactic instruments designed to improve the usability of the language. A discussion on constructors, and considerations on the efficiency will close the chapter.

3.2. A First Example

The first example is, of course, the mandatory “Hello, world!” test program, which will give a general idea of the look of the language.

```
first_example :  uses system
{
 !test()
 {
  println("Hello, world!");
 }
}
```

In this example there is a definition of a package “`first_example`,” which contains a single global method which prints the requested text string. The exclamation mark qualifies the method as exportable from the package. The clause “`uses`” defines which packages are imported. If an interactive environment is available, the execution of the program could look like:

```
# test();
Hello, world!
# _
```

3.3. Class Definitions

```
second_package:  uses system
{
//
// first class definition:  glass
//

 !glass :  super object
 {
  !glass():  super object()
  {
  }

  !break(w:glass)
  {
   println("Crash!");
  }
 }
//----------
//
// second class definition:  mattress
//

 !mattress :  super object
 {
  springs:long;

  !mattress(n:long):  super object()
  {
   mattress.springs:=n;
  }

  !break(m:mattress)
  {
   for a:=0; a<m.springs; a:=a+1;
   {
    println("Sproingg!!");
   }
  }
 }
}
```

Table 3.1.: Example of class definitions

After this first example, it is time to define some new classes, introducing a few other key elements of the language. The example in Table 3.1 is somewhat long but its meaning is easily understandable. Two new classes are defined, one defining glasses and one defining mattresses. Both accept the message “break,” which is used to ask instances to break themselves. According to the principles of object oriented programming, each object reacts in its

proper way to the message: a glass will print “Crash!,” while a mattress will print that many times “Sproingg!” as many are the springs contained in it. What follows is an example of interactive use of the classes, after which the new elements introduced will be described.

```
# a:=glass();
# b:=mattress(3);
# break(a);
Crash!
# break(b);
Sproingg!!
Sproingg!!
Sproingg!!
# _
```

Every class definition has the following form:

```
<classDef>::=<optionalBang> <id> ":" <superList> "{" <classBody> "}"
```

For every class it is possible to specify one or more superclasses from which to inherit characteristics (messages accepted and internal structure). The exclamation mark before the class name declares the class as usable outside the package. The message body is composed as follows:

```
<classBody>::=<fieldList> <methodList>
```

The declarations of instance variables precede all the local methods of the class, as, for example, the number of springs in the definition of “mattress.”

```
springs:long;
```

In BOH, the identifier of the object declared is in first position, followed by colon and the type specification, similarly to what happens in Pascal.[2] The possible type specifications for instance variables will be shown in the following sections.

The list of methods consists in zero or more method definitions, which have the following form:

```
<method>::= <methHead> <retVal> <methTail>
```

[2]For further details, it is possible to refer to the BNF description of the Pascal language, often listed in the appendix of the language manuals, or alternatively to the syntactic diagrams, as [Met], or [Gro92, page 527].

where

```
<methHead>::= <optionalBang> <id> "(" <paramList> ")"

<retVal>::=
<retVal>::= ":" <id>
<retVal>::= ":" "super" <superCallList>

<methTail>::= "{" <cmdList >"}"
```

The list of parameters is composed of zero or more parameter declarations, and `cmdList` of zero or more commands.

While the first two forms of `retVal` correspond to methods which work on existing objects, the third, which uses the token “super” identifies a constructor. Constructors have a behaviour similar to the one found in C++: the method invokes the constructors of subinstances corresponding to superclasses. Those subconstructors are listed in `superCallList`. Clearly, their number and type must match the list `superList` used in the class definition. In the example of Table 3.1, mattress is subclass of object; therefore, when a new object is created using mattress(), the call object() will initialise appropriately the “object” part of a mattress. The rest of the constructor will deal with the initialisation of the instance variable defined in this level of the hierarchy (in this case, the variable “springs.”) There can be more than one constructor for every class, and their name and parameters can be arbitrary.[3]

A return value is referred using a pseudovariable which has the same identifier used for the method, as it happens in Pascal. In this case, though, the variable is also usable on the right side of assignements, since there is no confusion with message invocations, which are characterised by the use of parentheses after the identifier.

3.4. Declarations and initialisations

In the definition of the method break(), in the above example, the variable “a” appears not to have any declaration. As a matter of fact, the declaration is unified with the first assignment “a:=0”. While parsing the source code, the compiler treats the first assignment to an unknown variable as a declaration plus an initialisation. This is made possible by the fact that, using the mechanisms shown previously, in every point of the source it is possible to determine the static type (the most generic type) of an expression, and therefore to obtain the (most

[3] More information on constructors and their use is in Section 3.14.1.

generic) type of the variable which is being declared. A static limitation for the range of possible types for the variable is present, and the typical advantages of static type checking are fully retained.

The fact of binding together declarations and initialisations have several advantages. First of all it is impossible to have uninitialised variables[4] (which is, for instance, a weak point of both C and Pascal.) Furthermore it is not necessary to declare, at the beginning of the method body, all the variables used, including those of minimal relevance as temporary variables or cycle indexes. On the other hand, it is still possible to declare, at the beginning of the method, all the crucial variables, but in this case a suitable initial value must be assigned to them.

As a concrete example, let us consider the following source fragment:

```
a:=object(); // the static type of a is now object
a:=5;        // legal, long is subclass of object
a:="ciao";   // legal, text is subclass of object
println(a);  // legal only if println() is
             // defined for object
```

The legality of the call println(a) is determined statically according to the static type of a (in this case object). Dynamically the most specific method applicable will be called for println – in this specific case println(text).

Variables are treated by all means like references to instances. This implies that, if the source code contains an assignement like “a:=b”, its meaning will be: “b refers now to the same object referred by a.” In practice, the same object will be referred by both variables. This behaviour could lead to problems of referential opacity, but the only viable alternative would be to duplicate completely the referred object, which would severely affect the program performance. For an analysis of the problem, a comparison with other languages and a description of the treatment of primitive types, the reader can refer to Section 3.14.2, “Instances Sharing and Side Effects.”

Primitive Types and Constants

As in any other language, BOH offers a number of primitive types, listed in Table 3.2 together with their definitions.[5]

[4] There is no similar guarantee for the initialisation of instance variable inside a constructor. That responsibility is left to the programmer, who must define the proper semantic for the implementation and the initialisation of the object and its components.

[5] The choice of using the term “word” for 16 bit integers is mostly due to historic reasons: the terms byte, word, long and quad are probably very familiar to all those who used a Motorola 680x0 processor, which allowed those as basic data types.[Mot92] In reality the term “word” is often referred to the unit by which memory is accessed, and is therefore dependant on the hardware architecture. An alternative choice could

byte	integer number in the range -128...127 (-2^7...2^7-1)
ubyte	integer, 0...255 (0...2^8-1)
word	integer, -32.768...32.767 (-2^{15}...2^{15}-1)
uword	integer, 0...65.535 (0...2^{16}-1)
long	integer, -2.147.483.648..2.147.483.647 (-2^{31}...2^{31}-1)
ulong	integer, 0...4.294.967.296 (0...2^{32}-1)
quad	integer, -9.223.372.036.854.775.808...9.223.372.036.854.775.807 (-2^{63}...2^{63}-1)
uquad	integer, 0...18.446.744.073.709.551.615 (0...2^{64}-1)
bool	true or false
text	text, or a string or characters
float	single precision floating point numbers (IEEE-754)
double	double precision floating point numbers (IEEE-754)

Table 3.2.: Primitive types available in BOH.

Since BOH is strongly typed, every constant must be recognisable as belonging to a unique type. If a specific kind of numeric constant is needed, a suffix can be added to specify the type, similarly to what happens in C,[6] as follows. If D is a digit between 0 e 9, and L an optional minus sign, then:

```
<L><D>+          has type long  as in:  61342
<D>+"u"          has type ulong as in:  3183714311u
<L><D>+"b"       has type byte  as in:  34b
<D>+"ub"         has type ubyte as in:  34ub
<L><D>+"w"       has type word  as in:  -11621w
<D>+"uw"         has type uword as in:  34uw
<L><D>+"l"       has type long  as in:  -81l
<D>+"ul"         has type ulong as in:  63ul
<L><D>+"q"       has type quad  as in:  711q
<D>+"uq"         has type uquad as in:  3uq

"true"           has type bool
"false"          has type bool

<L><D>+"."<D>+             has type double  as in:  3.25
```

have been “short” instead of “word.” The standard IEEE-754 is explained in detail in many microprocessors manuals.[Mot93][Mot94] Floats are considered to be 32 bits values, and doubles are twice that much, that is 64 bits.

[6] In C a constant like 4L is a long, 42112U is unsigned, 133UL is unsigned long, the constant 4.2f5 is a float and 4.2e5 is a double.[KR89]

```
<L><D>+"."<D>+"f"         has type float   as in:  -2.4f
<L><D>+"f"                has type float   as in:  61f
<L><D>+"f"<L><D>+         has type float   as in:  4f-11
<L><D>+"."<D>+"f"<L><D>+  has type float   as in:  3.6f9

<L><D>+"."<D>+"e"         has type double  as in:  -2.4e
<L><D>+"e"                has type double  as in:  61e
<L><D>+"e"<L><D>+         has type double  as in:  4e-11
<L><D>+"."<D>+"e"<L><D>+  has type double  as in:  3.6e9
```

Other unusual ways to represent floating point numbers are described in Section 3.10, “Syntactic Tools.”

Text objects have no predefined limitations in length, and no assumption is made on their internal format. The only mandatory operations are the few imposed by the standard libraries: a print operation, concatenation, input from terminal and a few others. In the source code a text constant is enclosed in double apices ("). Two consecutive double apices inside a text constant have the effect of having one double apices character inside the constant, as it happens in Pascal. With the exception of text constants, the source code is treated by the compiler as case insensitive.

3.5. Identifiers

The range of characters usable in identifiers is considerably more extended than the one available in other languages. Every identifier can in fact be an arbitrary sequence of the following:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789'~|\/?><+=_-*&^%$#@!
```

with the restriction that the first character cannot be a number. It is therefore perfectly acceptable to write a method like:

```
!  ~B%#@@$?  ():long
{
 ~B%#@@$?  := 15;
}
```

From the language point of view, even the usual infix operators like "+", "-" and similar are considered to be identifiers rather than separate tokens. A suitable parsing algorithm internally transforms the infix operators in ordinary function calls.

Actually, an interesting feature of the language is the ability for the programmer to extend freely the language with custom infix operators, using identifiers composed by all the characters of the above list. This feature is reminiscent of the similar one available in Prolog[Proa][Prob]. A description of the mechanism is available in Section 3.11, “Operators”.

3.6. Instance Variables

Every class can contain one or more instance variables, defined using the following syntax:

```
<field>::=<idList> ":" <fieldType>
```

idList is a sequence of one or more identifiers separated by comma. *fieldType* is defined as:

```
<fieldType>::=<id>
<fieldType>::=<id><indexList>

<indexList>::=<index>
<indexLst>::=<indexList><index>

<index>::="[" <numLong> "]"
<index>::="[" <numLong> ".." <numLong> "]"
```

Every instance variable can therefore be defined as an object or an array (possibly multi-dimensional) of objects. For every dimension is possible either to specify the total number of elements (in which case the index will be between 0 and n-1, as in C), or alternatively specifying both the lower and upper bounds, in the style of Pascal. This is the only part of the program in which it is possible to define arrays. This is due to the fact that an explicit specification of a maximum number of elements for a data type external to the class establishes an undesirable dependency from the implementation. If, on the other hand, every access to the array and its maximum number of elements are encapsulated inside the class, it is easier to change the size of the array without adverse effects, or even to replace the array altogether with another kind of data structure. Where in other languages common practice is to define a global variable that is an array, in BOH the programmer is asked to first define a simple

class containing the array definition, and then to define the global variable as an instance of that class. The extra work is more than compensated by the greater generality of the code.

3.6.1. Parameters

The syntax of the list of formal parameters is the following:

```
<paramList>::=
<paramList>::=<idList> ":" <id>
```

where `idList` is a list of identifiers separated by comma and `id` is the identifier of a class name.

As already mentioned, the parameters are always treated as references to the objects used as actual parameters. The parameters passed to the method, from an implementation point of view, could be pointers to instances or directly the data in the case of primitive types. However, it is possible to obtain the same behaviour in both cases using suitable techniques, as described in Section 3.14.2, “Instances Sharing and Side Effects.”

3.7. Encapsulation

In the object oriented languages in which a single receiver is allowed, there is a natural association between every method and the class of its receiver. This connection is normally used to support some form of encapsulation, by grouping together messages which refer to a single class. Those methods consitute therefore an interface to the internal implementation of the class. Different policies are then possible to control the visibility of the instance variables defined for each class from the internal methods, from the methods related to the subclasses of the given class, and from other unrelated methods.

Using multimethods, there is no longer a unique connection between a single class and each method, since all the parameters are considered symmetrically. To reestablish some form of encapsulation, a new approach is, therefore needed. In Cecil, every multimethod can access all the instance variables of all the classes of the parameters involved. This policy is one of the possible forms of access control, but it is not immediately clear how the interface to the various parts of the program is defined.

The technique used in the language allows every method to be related in the priviledged way described to at most one class. The instance variables of that class will be directly accessible by the method, while the internal structure of the instances belonging to other classes will be accessible exclusively using messages. Therefore, every method is a part, from a logical point of view, of the implementation of a single class. This accounts for a

better separation between the implementations of different classes, enhancing encapsulation and reducing the interdependencies. To describe the association between a method and a class, BOH requires that the source of method which is part of the implementation of a class has to be physically enclosed in the module describing the class, obtaining therefore a single piece of source code for the implementation of every class. To handle the case in which a method does not need to be tied to a specific class, the language allows, additionally, the definition of “global” methods, which are placed in the package, but outside all class definitions. Their code is not allowed to use directly the internal structure of any instance, but can use freely messages to operate on other objects as required. An additional advantage of this way of organising the code is the ability to implement efficiently primitive data types with little effort, as explained in Section 3.14.2.

3.8. Nested Contexts

As in C and similar languages, it is possible to create blocks, or contexts, that is parts of program with their own local variables, which are treated like single instructions. In this simple definition it is not possible to create nested class definitions or methods.

This is the syntax used:

```
<context>::= "{" <cmdList> "}"
```

Contexts can be nested arbitrarily, and the variables defined inside a block will appear as undefined outside the block. Here is an example:

```
a:=5;        // a is the only var defined in this point
{
 b:=a;       // b is local to this context
 {
  c:="ciao"; // c is defined and forgotten right after
 }
 // x:=c;    // it would be an error.  c is undefined, here
 {
  c:=811.23; // a variable distinct from the previous one
 }
}            // b and c are no longer usable
```

Interestingly, because of the way variables are defined and simultaneously initialised, it is impossible to have local variables that hide other variables with the same name defined

previously. For instance, in:

```
{
 x:=expression;
 ...
```

there are two possible cases: if the variable “x” was already defined, then this is an ordinary assignment. If “x” was not previously defined, then it is a declaration with the related initialisation. This is in no way a limitation, but helps instead to avoid potential error conditions. In fact, the mechanism by which a more deeply nested local variable hides a previously defined one with the same name is very rarely used, and its practical usefulness is questionable, since it requires an unnecessary mnemonic effort by the programmer to distinguish the various variables with the same name. Indeed, such an occurrence is likely to be the result of an involontary mistake by the programmer, and many compilers issue a warning when a similar situation is detected.[7]

Blocks are freely usable any time the programmer desires to introduce an isolated fragment of code with its own local variables. Their most natural use is in conjunction with control structures, as detailed in the following section.

3.9. Control Structures

3.9.1. While

The “while” loop has the following structure:

```
<cmd>::= "while" <expr> "{" <cmdList> "}"
```

The command sequence `cmdList` is executed while the expression is true. The evaluation of the expression is done before each iteration. If initially the condition is false, the commands listed are never executed. `expr` must have static type bool. The part enclosed between "{" and "}" is a context, and it is therefore possible to define local variables. An alternative syntax is available:

```
<cmd>::= "do" "{" <cmdList> "}" "while" <expr> ";"
```

In this second form, the sequence `cmdList` is always executed at least once.

[7] For instance, gcc has a command line flag (-Wshadow) to activate a warning if a local variables hides a more external one, since this could be the result of an error.

3.9.2. If...elseif...else

This is the form of the choice instruction:

```
<cmd>::= <ifHead>
<cmd>::= <ifHead> else "{" <cmdList> "}"

<ifHead>::= "if" <expr> "{" <cmdList> "}"
<ifHead>::= <ifHead> "elsif" <expr> "{" <cmdList> "}"
```

If the evaluation of the boolean expression following "if" gives, as a result, "true," then the context which follows is executed. If the result is "false," all the expressions following the various "elsif" clauses are evaluated, one at a time (if they are present), and only one context, the one following the first true expression encountered, is executed. If none of the expressions are true, the context following "else," if present, will be executed.

3.9.3. For

The definition of the “for” construct is:

```
<cmd>::= "for" <cmd> <expr> ";" <cmd> "{" <cmdList> "}"
```

Similarly to the equivalent C, the first command is executed once and is used to set up the loop. Subsequently, the expression is evaluated. If it is false, the loop exits, while if it is true the list cmdList is executed, followed by the second command, usually used to increment indexes, after which the evaluation is repeated and so on. It is worth noting, in the definition, the use of the semicolon. The sign is used to conclude an instruction, and is therefore considered to be part of the command, as in:

```
<cmd>::= <id> ":=" <expr> ";"
```

Examples of “for” cycles:

```
for a:=0; a<10; a:=a+1;
{ println(a);}

for {a:=0;b:=2;} a<10; {a:=a+1;b:=b*2;}
{ println(b);}
```

3.9.4. Case

The syntax of the “case” construct is the following:

```
<cmd>::=<caseStruct>

<caseStruct>::=<caseBody> "}"

<caseBody>::=<caseHead>
<caseBody>::=<caseBody> <exprList> ":" <cmd>
<caseBody>::=<caseBody> "default" ":" <cmd>
<caseHead>::= "case" <expr> "{"
```

The expression following the “case” clause is evaluated and compared against the test expressions. The first matching expression causes the following command to be executed, after which the execution resumes after the “case” structure. The expressions can be of arbitrary type; the comparison is made using the binary message “=”, whose behaviour can be obviously customised for different classes.

3.9.5. Other Constructs

At the current stage of definition of the language no “break” instruction, used to force an anticipated exit from cycles, is defined; its effect can be however simulated using logic variables. Intentionally, there is no provision for unconditional jumps (“goto” instructions) or labels associated with instructions.

3.10. Syntactic Tools

In most programming languages there are instruments whose purpose is to make the code more understandable and to simplify the programming activity by simplifying or adding functionalities on the syntactic level; the language which is here defined is no exception. Those instruments are usually referred with the term “syntactic sugar,” since they do not change the way the language works, but are a commodity offered to the programmer to improve aspects of the source code, such as its legibility or its comprehensibility. We shall review here which are the features offered by this language definition.

3.10.1. Comments

It is possible to make the code more comprehensible by adding to the source program textual annotations, which, although ignored by the compiler, can offer useful information on the

way the code works. Comments can be either written on a single line or as a block of text. On each line of the source code, the compiler ignores everything that follows the characters "//" (unless they appear inside a text constant). For instance:

```
a:=5;   // the value of a is now 5.
```

The compiler considers as a comment a block of text which is included between the sequences "/*" and "*/". Differently from other languages, in BOH block-style comments can be nested, which can be useful during the debugging when other comments are already present, as in:

```
a:=5;
/*
a:=4;   /* the value of a is now 4 */
*/
// actually, it is 5.
```

3.10.2. The Dot

The dot is used in a particular way, using a notation to the one found in the language Dylan.[App95][Sha97][Unia] In BOH, the use of the dot is perfectly equivalent, at the syntactic level, to the use of a pair of parentheses, according to the following criteria:

```
.x             is equivalent to x()
a.x            is equivalent to x(a)
a.x()          is equivalent to x(a)
a.x(b,c,...)   is equivalent to x(a,b,c,...)
```

This equivalence holds for message calls, access to instance variables and even for numbers containing a decimal point.

For instance, instead of writing:

```
println("Hello!");
```

it is perfectly legal to write:

```
"Hello!".println;
```

Similarly, referring to one of the first examples, the notation:

```
springs(a)
```

could have been used to refer to the field a.springs.

Finally, although surprising, it is perfectly acceptable, for instance, to write the number 4.25 as 25(4), and the number 6.3e-8 as 3e-8(6). The current implementation of the compiler can indeed accept either forms.

Apart from the aesthetical factor, there is a concrete practical aspect. For instance, a programmer used to Smalltalk could be pleased to discover that 4+5 can be alternatively written as 4.+(5) – this use can appear much more familiar to those who use a traditional object oriented language where there is a single receiver for objects. The code

```
myWindow.close();
```

can be more easily interpreted as a message close() sent to the object myWindow. Similarly, the code:

```
myWindow.move(30,80);
```

has exactly the same notation of that used in a more traditional language. In reality, the message dispatching is performed considering the dynamic type of all three the parameters, instead of considering just the first, but it still possible to obtain a behaviour similar to the case of single receivers by defining the methods appropriately,[8] if desired.

To return to the syntactic aspect, using the dot as described allows us also to improve the clarity of the code when an expression contains many nested function calls. The two following forms are equivalent:

```
nested:          four(three(two(one(a,b),c,d)),e);

using the dot:   a.one(b).two(c,d).three.four(e);
```

The two expressions are equivalent, but the second form is simpler, and gives a better idea of the sequence of the calls chaining. As an additional advantage, the language could be more practical, at least in principle, to be used interactively as a scripting language. For instance, it is easy to notice the similarities between the two following lines:

[8] If for each message there is a single method definition for each type of the first parameter, the actual code behaviour is undistinguishable from that of a language based on messages sent to single receivers.

```
Unix:   cat mytext | cut -c1-5 | grep "pattern" | more

BOH:    cat(mytext).cutc(1,5).grep("pattern").more;
```

What is usually done with pipes in the Unix shell could be replicated by chaining functions in the way shown.

3.11. Operators

One of the most peculiar characteristics of BOH is the treatment of operators. In fact, while in most languages infix operators are hard-coded in the syntactic specification, and it is not possible to change them, in BOH the definition of operators is a characteristic which is modifiable and extendible by the user.[9] To the set of predefined operators (imported from the system package), the user can add custom ones with definable levels of priority and associativity, which can help to make the code clear and understandable.

For instance, we might want to add a custom operator %, left-associative, with priority greater than "+" and "-", but lower than "*" e "/", which can be simply obtained writing:

```
operator x % xx 450;
```

It is now possible to write in the package body something like:

```
a:=5+4%6*9%8;
```

and the compiler will internally transform the command into:

```
a:=+(5,%(4,%(*(6,9),8));
```

The syntax of a new operator definition is as follows:

```
<operDef>::="operator" "x" <id> "x" <numLong> ";"
<operDef>::="operator" "x" <id> "xx" <numLong> ";"
<operDef>::="operator" "xx" <id> "x" <numLong> ";"
<operDef>::="operator" <id> "x" <numLong> ";"
<operDef>::="operator" "x" <id> <numLong> ";"
```

[9] This functionality descends quite directly from the equivalent one found in Prolog, where, on the other hand, it is rarely used, given the particular nature of the language.[Proa][Prob]

All the operator definitions must appear at the beginning of the package definition, before every class. The number refers to the priority: operators with higher priority have higher precedence (e.g.: “*” has higher priority than “+”). The default priorities for the most common operators, imported from the system package, are listed in Section 3.13, "Library Functions".

The form `x id x` refers to a non associative operator (like ":="); `x id xx` is an left-associative operator, that is `a id b id c` becomes `id(a,id(b,c))` ; `xx id x` is a right-associative operator, that is `a id b id c` becomes `id(id(a,b),c)` ; `id x` is a monadic prefix operator and `x id` is a monadic postfix operator.

It is important to note that the symbol used to identify an operator is a generic identifier, which can include characters other than letters and numbers. It is therefore possible to define operators like "suitable?", "+-almost", "miXed", "!$*??o91@#" and so on. A custom preprocessor internally transforms all the infix operators into normal functions calls. The following code is an example:

```
operator x leone x 150;
operator a+ww x 150;
operator x *rrww 55;
operator x +j x 800;
operator x -j x 800;
operator xx + x 300;
operator xx - x 300;
operator xx * x 500;
operator xx / x 500;
operator - x 750;

 tok:=4+5*3--6/2.5e-6+-y*rrwwleonex*rrww;
```

This is automatically transformed in the following:

```
 tok:=*rrww(leone(*rrww(+(-(+(4,*(5,3)),/(-(6),5e-(6)(2))),-(y))),x));
```

As a concrete example, the following is a minimal definition of complex numbers as operators:

```
operator x +j x 650;

class complex:  super num
{
 re,im:double;

 +j(r,i:double):  super num()
 {
  +j.re:=r;
  +j.im:=i;
 }
```

Using this minimal definition, it is possible to write directly in the source code expressions like `5.2+j6.4`, or `-3e-8+j2.25e6`. A more complete definition of complex numbers, available in the appendices, allows the programmer to use more complex forms like:

```
println( 3.0+j1.24 + 4.21-j7.11 );
```

Using operators, the notation becomes in this case very clear and simple to use.

3.12. Type Tunnels

The language offers a limited support for methods that return a value whose static type is the most generic among some of the actual message parameters. This allows to define a single method to operate on a range of classes while maintaining the most specific possible definition for the return type. In:

```
!tunn1(aa:A):aa
{
 tunn1:=aa;
}
```

the identifier of the parameter is used as the type of the return value. This means that the parameter and the return value belong to the same type and, as long as the method implementation maintains that connection, the compiler is able to obtain statically a much more specific information about the return value. It is therefore possible to write a single method that operates on an instance of A, but the compiler will still be able to determine that, since the return type is the same one of the actual parameter, the returned value belongs to something more specific than a generic A. In the example above, if A has as a subclass B, bb is

an instance of B and a message “testB()” is available only in class B, the following code is perfectly legal:

```
testB(tunn1(bb)); // legal!  The static type is B
```

The information concerning the “type tunnel” is preserved through intermediate variables and function calls. In the following definition:

```
!tunn2(aa:A):aa
{
 one:=aa;
 two:=tunn1(one);
 tunn2:=two;
}
```

the compiler is able to detect that the binding between the return value of tunn2 and its parameter is still valid, and the definition will work as expected. Additionally, it is also possible to bind the return value to a group of parameters, in the following way:

```
!combine(aa:A,bb:aa):aa
{
 if Atest(aa,bb) {
  combine:=aa;
 } else {
  combine:=bb;
 }
}
```

In this case the return value is bound to two parameters. The effect is that the return type will be statically, in any case, at least the most generic between the static type of the two parameters. If combine is invoked on parameters whose static type is (A,A), (A,B) or (B,A), its return value will be detected statically as A, if the message parameters have static types (B,B), the compiler can safely use B as the static type of the return value.

The technique is not so useful as an implementation of parametric polymorphism, but it can be used to solve a few practical problems. The mechanism described is fully supported by the current implementation of the language compiler.

3.13. System Library

To obtain a working compiler, the core definition of the language must be integrated with an essential set of standard functionalities, like basic I/O, boolean and arithmetic functions, and standard operators. While older languages used to incorporate all those functions in the main language definition, the trend of all modern languages is to keep as much as possible of the above mentioned features in a separate module, to allow an easier replacement, extension or customisation of the more common utilities without the need to change the language definition. This section contains some indications on the content of an essential system library, needed to obtain a usable language; however, the specification would need to be expanded to obtain a more complete programming environment in case the language were to be used for real life programming tasks. A usable subset of the proposed system library is available in the current implementation.

3.13.1. Standard Classes

In Figure 3.1 a diagram shows which could be a possible minimal class hierarchy to be included in the system package.

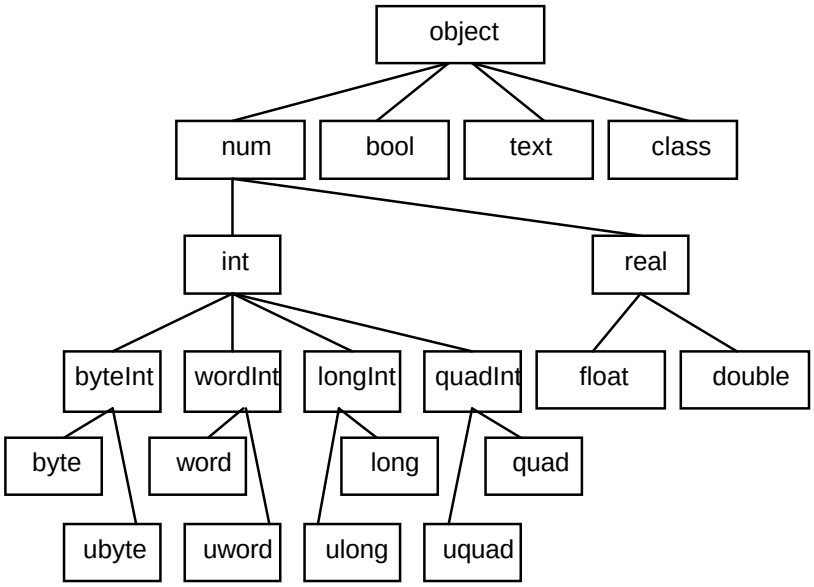


Figure 3.1.: Standard classes

Num, int, real, byteInt, wordInt, longInt and quadInt define common characteristics for the various classes of numbers.

3.13.2. Arithmetic Operations

In the system package there should be a provision for at least the four basic numeric operations ("+", "-", "*", "/") on every numeric type. To convert numbers from one type to another, a method with the name of the destination should be available, such as: long(5.2) returns 5L. For real numbers the trunc() call, logarithms and the main trigonometric functions are needed, while others, such as round(), can be a useful addition. Arithmetic operations must be applicable to arguments of heterogeneous type, performing the necessary type conversions internally. To convert a longer integer type to a shorter one, two functions to extract upper and lower part would be useful (for instance: upper and lower word of a long int, or lower and upper byte of a word and so on.) On integer numbers, shift operations (both logical and arithmetic[10]), increments and decrements should be defined.

3.13.3. Comparison Operations

On numeric values must be defined the operations "<", "<=", ">", ">=", "=" and "<>", with the usual mathematic meaning. All those operators return a boolean value.

3.13.4. Boolean Operations

The standard library should support the usual "not", "and", "or" operations. Optionally there could be a provision for additional operations like "eqv","xor" etc.

3.13.5. Standard Operators

These are the operators which should be supported, listed together with their priority:

```
operator xx + x 300;
operator xx - x 300;
operator xx * x 500;
operator xx / x 500;
operator - x 750;
operator x < x 200;
operator x > x 200;
operator x = x 200;
operator x <= x 200;
operator x >= x 200;
```

[10] In a logical shift, the value is considered as a bit field; bits set to zero are used to fill in the void space. In an arithmetic shift, the value is considered as a signed integer number; a right shift on a negative number will set the most significant bit to one, in order to preserve the sign of the number.

```
operator x <> x 200;
operator x and x 150;
operator x or x 130;
operator not x 170;
```

The dot has priority 700, if used to join identifiers (as in id.id(abc)), while it has priority greater than every other operator if it is detected inside a floating point number.[11] The assignment instruction ":=" is *not* an operator, and it behaves quite differently from usual functions. Section 3.14.2 has more details.

3.13.6. Input/Output Operations

To offer a minimal support for user interaction, the library should define the messages "print(object)" (which, according to the type, print a textual representation of an object), println(object) (print line, as the previous one plus a newline character), nl() (prints a newline character). Furthermore, the functions readLong(), readText() and so on should be defined (to read objects from the console).

3.13.7. Other Functions

The generic test "=(object,object):bool" has the default behaviour of equality between references to object (two objects are equal if they are the same object). This behaviour is changed in arithmetic equality for numbers, and can be redefined as needed for user-defined classes. A function "class(object):class" will return the class to which the parameter belongs, while a message "panic(text)" will abort execution printing the text as an error message. There is currently no definition for a more sophisticated error handling mechanism).

3.14. Further Considerations

3.14.1. Ensuring Coherence: Constructors

As previously seen, a few simple rules allow us to obtain a type system which enables a static type checking and is generally coherent. The rules, simplified for clarity, were the following:

To ensure the coherence in the types of return values, a rule asserts:

[11] It is always possible to recognise a floating point number since its first character is numeric, which is illegal for identifiers.

- If two methods have equal identifier and arity, then they correspond to the same message, and their definitions must be covariant, that is if two methods are fun(A1,B1):C1 and fun(A2,B2):C2, and A1 is subclass of or equal to A2 and B1 is subclass or equal to B2, THEN C1 must be subclass or equal to C2 as well.

By applying the rule above it is possible to obtain, for every combination of parameters for a message, a single limitation of the type of the return value, that is the most generic type of that value. More generally, that allows us to obtain statically the most generic type of an arbitrary expression.

A second rule is used to ensure that in runtime one and only one more specific method will be applicable for every combination of dynamic types of the parameters:

- If in two methods corresponding to the same message fun(A1,B1) and fun(A2,B2), A1 is subclass of A2 and B1 is superclass of B2, THEN there must exist a definition of the method fun(A1,B2).

Whenever there is a potential ambiguity in the method definitions, the user is requested to specify exactly what is the intended behaviour of the message.

For ordinary methods this technique works quite well. However, a little complication arises for constructors. If there is a method definition like:

```
!constructorSubClass():  super constructorSuperClass() {...
```

the meaning is that the part of the instance which corresponds to the superclass is created and initialised by constructorSuperClass(). It is clearly required that the constructor returns an object belonging to the superclass, but never to one of its subclasses. That would mean that the part of the object corresponding to the superclass belongs to still another class, which does not make much sense. Unfortunately, in this circumstance, constructors operate somewhat outside the general principle according to which any value belonging to a subclass of a given class is usable wherever an object of that class is expected. The exception is due to the fact that constructors deal not exactly with the value of the object but with the object itself, which is being created. Admittedly, the use and the requirement which are imposed by constructors are somewhat inconvenient and the mechanism could probably be refined.

In the meantime, it is necessary to impose rules devoted to ensuring that the method definitions are such that they comply with the requirement above. The following set of rules is suitable for that purpose.

- Given two methods corresponding to the same message, if one of them is a constructor costr(A,B):C and the other is a normal method fun(A1,B1):C1, THEN it cannot be the case that simultaneously A1 is subclass of or equal to A and B1 is subclass of or equal to B.
- If there are two definitions costr(A,B):C and costr(A1,B1):C1, with A1 is subclass or equal to A and B1 is subclass of or equal to B, THEN C1 must be equal to C. (no variance for constructors)

The first rule ensures that no ordinary method can be more specific than a given constructor; therefore, if a constructor is called with given parameters, no conventional method will "steal" its role. In other words, we will be sure that a real constructor is called. The second rule ensures that every constructor more specific than another constructor will return a value of the same class, and not of a subclass. That allows us to determine statically not only the most generic type, but the exact one, which is what is required (in the sole case of constructors). The above set of rules is not particularly restrictive, but ensures effectively that constructors can be called safely. In the implementation which is here described both rules are enforced, and it is possible to check their efficacy using some of the test programs.

3.14.2. Instance Sharing and Side Effects

In their implementation, most compiled object oriented languages keep their instances in blocks of contiguous memory. If this is the approach, a variable which refers an object can be handled essentially in two ways: either considering the whole memory block as the variable, or using a single pointer to the real object structure kept in memory. The first approach, conceptually cleaner, is extremely difficult to implement efficiently, since, every time an object is used as a parameter to a function, the entire object structure should be replicated. This would lead to an extremely high overhead in terms of memory used and time spent copying structures. For this reason, most implementations use the second solution, managing the variable as a pointer.

This more efficient solution, however, has an important consequencet, that deserves to be carefully considered: every time there is an assignement, and a variable receives a value, what is transferred in the variable is, in reality, just a pointer to an already existing object. The result is that, if a variable is assigned to another variable, the memory block containing the object, after the assignment, is shared by the two variables. This sharing is not harmful in itself, but can have, as a result, unexpected side effects.

Let us consider, as an example, the following Java program:

```
class sideEffect
{
 long n;
 sideEffect(long v) { n=v;}
 static void dangerous(sideEffect a,sideEffect b)
 {
  System.out.println("b is "+b.n);
  a.n+=20;
  if (b.n>10)
   System.out.println
      ("Hey!  The variable b has changed!  Now it is:  "+b.n);
 }
 public static void main(String av[])
 {
  sideEffect a;
  a=new sideEffect(5);
  dangerous(a,a);
 }
}
```

The output of the program is this:

```
b is 5
Hey!  The variable b has changed!  Now it is:  25
```

As shown, when the parameter a is incremented, the parameter b gets incremented as well, despite no explicit action is performed explicitly on b. The result is that a variable can, in certain circumstances, change unexpectedly its value. Unfortunately, there is no immediate solution to this problem, unless a non imperative programming paradigm is used, or a considerable overhead is accepted. The language which is here defined, therefore, accepts this compromise as well, and uses pointers (although implicitly) to refer to objects. On the other hand, even the elegant language Smalltalk has the same behaviour, as can be proved writing a small test program.

In this language, therefore, the instances are implemented using memory blocks referred by pointer, as already mentioned in Section 3.4. The result is that the semantic of the assignment operation is as follows:

```
a:=b;
```

means: “The variable a does no longer refer the previous object; after the assignment the variable refers the same object referred by b.” As a result, after the assignment, a and b are two references to the same object, and the two will remain shared until one of them is assigned again to some other value. The object that was referred by a before the operation is no longer used and, if no other references to the same object are present, the related memory area can be safely disposed.

The semantic of the operation is now defined. Naturally, it would be reasonable to expect that the same behaviour holds for every type of object, regardless of their type. This, however, could involve a penalty in the use of simple, primitive types. If even the most basic types like numbers and characters were implemented as objects, in the following code:

```
b:=5;   // b is a pointer to the numeric object "5".
a:=b;   // a and b refer now to the same object
inc(b);
```

In that example, the variable “a” would be expected to change as well, according to what has been said concerning the sharing of objects. Maintaining the same behaviour, though, would prevent to keep those simple data types in machine registers. In fact, if the “5” were kept in a register, instead of keeping a pointer, an increment would affect exclusively a single variable, instead of every variable that, according to the definition, should be connected to the same object. On the other hand, using objects for every simple numeric value, boolean etc would involve a huge number of bookkeeping operations, wasting system resources. For this reason Java, for instance, offers a number of primitive data types which are explicitly not objects. A variable which has one of those types is not managed in same way an object is, and cannot receive messages. For every basic type, a symmetric “wrapper” type is available, which is used to build an object with the same characteristics of the basic type, and an explicit conversion between the two forms is requested by the user. Therefore, Java offers the types int (primitive), Integer (object), long (primitive), Long (object), etc. This solution is somewhat inconvenient in which not every variable is treated in the same way, and multiple representations for the same conceptual data type are present, requiring a conversion.

An alternative, much cleaner solution, can be obtained in BOH. Since, as previously described, the only methods authorised to change the internal structure of an objects are those defined inside the class, and since the basic types are defined in the system library, it is sufficient to define the functions which work on those primitive types in such a way that they *never* change the content of those object used as their parameter. For instance, the function inc(), as defined in the system library, should not change the “internal” state of its parameter,

but can instead simply return the incremented value as a new object.

As a consequence, the only ways to change objects belonging to one of the primitive data types are either to use assignment, or to use one of the library functions (which could pretend to create a new object as a return value). In both cases the behaviour of the assignment is perfectly consistent with the semantic defined, which therefore holds for every data type. The fact that the object is internally managed as a single machine word is completely transparent to the user, and the illusion of using "true" objects is fully preserved. A single semantic is sufficient, and it is no longer necessary to sacrifice execution efficiency.

3.15. Parameters

The "trick" described, very useful to obtain an efficient implementation, solves however only part of the problem. The issue of how to deal with parameters needs to be addressed as well. The point can be made clear with a small example. In the call:

```
test(a:fruit);
```

the parameter of the message is passed to the dispatcher using a pointer to the object. The pointer will be typically stored temporarily in a register, or on the stack if the architecture does not have enough general purpose registers. The dispatcher will use the pointer to find the class of the object and will select the appropriate method. However, if the parameter is a value of a primitive type, as in:

```
test(a:long);
```

the matter becomes more complex. For efficiency reasons, the more logical place to store temporarily the object would be the register itself, bypassing the use of a pointer and the need to store the value in a real object in memory. On the other hand, the dispatcher could now receive a generic value which could be a pointer or a value, and there is no way to distinguish them.

There are several possible solutions, with different degrees of complexity. The most straightforward solution would be to create on-the-fly a temporary wrapper object in memory, so that the dispatcher receives always a pointer. The overhead involved would be, however, very high and there would be no advantage in keeping data of primitive types in registers or single memory locations. Another solution would be to tag the value with a flag that makes it possible to distinguish between the two cases. The downside is the loss of part of the number of bits usable, and the extra time needed to set and to extract the tag. A slightly

better solution could be to group together all of the tags into an extra word, using groups of bits to specify whether the corresponding parameter is a pointer or a primitive value, and its type. This is a rather interesting solution, but other options are available.

Thinking of the way the dispatcher operates, the only information needed to select the right method is the class of each parameter. An alternative could therefore be to pass a pointer to the class alongside each parameter. That does not require tags and speeds up the dispatcher, but on the other hand it requires to use two machine words instead of one for each parameter, which could lead, especially on architectures with few registers, to basically doubling the number of accesses to the stack, in memory, to park the parameters of each method. A much better solution, however is possible.

The mechanisms described for the type checking allow the language to determine the most generic type of each invocation. In a case like:

```
test(a); // static type of a:  fruit
```

the compiler can discover statically that the parameter is always an instance of fruit or a subclass, a complex object in any case. Therefore, a specialised dispatcher, possibly inline, does need to deal only with objects, and the use of a single pointer is sufficient. If, instead, the call is:

```
test(a); // static type of a:  long
```

the dynamic type of a can only be long or a subclass. By simply imposing that primitive types cannot be used to generate subclasses (which is a fairly rare eventuality anyway), the type can be determined entirely statically, and the dispatcher can be avoided altogether. If there is more than one parameter, there is no need to check the type of that one to select the proper method. Summarising, the parameter can be passed using simply a single register.

The only critical case is:

```
test(a); // static type of a:  object
```

In the case in which the static type corresponds to a class that has subclasses of either kinds, and only in this case, it could be used a pair of registers, one for the data (pointer or the value) and one for the class. The classes which need this treatment are typically a very small number (object and a few abstract classes), therefore this last soultion should allow to use optimally the available registers while saving time during the message dispatching.

3.16. Dispatching

As mentioned, by generating different, specialised forms of the dispatching call depending on the specific invocation it is possible to reduce the time required to locate the most suitable message and use more efficiently the available registers. In general, the way in which dispatching is implemented in object oriented languages falls into one of two categories: table-based and cache-based. Using tables, created during the compilation, the dispatcher essentially uses a reference to the class of the object as an index into a dispatching table, to obtain the address of the suitable method. The approach is usually very fast and efficient. An alternative, more common in dynamically typed languages, is to maintain a cache of the most recently used resolved method invocations, to minimise the time required to the dispatcher to find the right method in case the same one had been encountered recently during the code execution. A small problem of the dispatching tables in the case of multimethods, is that the space required grows (potentially) in an exponential way with respect to the number of parameters used. This pessimistic picture is in practice much less dramatic, since many optimisations can be performed to reduce considerably the size of the tables. There is a great deal of literature devoted to finding algorithms suitable to compress tables in the case of multimethods [PHLS99][DAS98]; however, by considering carefully the graph of multimethods, the optimisation can be even more drastic, eliminating, in many cases, the need for a dispatcher altogether. To do so, it can be noted that, out of all the possible tuples of the graph, only a limited number have actual multimethod definitions. In particular, it is possible to determine, inside that graph, a subgraph composed by the tuples that have a method definition plus all of their descendants. The resulting subgraph will be formed, in general, by a certain number of disjoint components. The interesting part is that, from the point of view of the dispatcher, those components can be considered completely different messages, reducing the number of steps needed to select the right multimethod. If a component contains a single multimethod, the dispatcher can be completely eliminated.

For instance, in graph of Figure 3.2, there are three distinct components, corresponding to the methods:

```
msg(B,B)
msg(B,C)
msg(C,B)
```

Since each of them originates a distinct subgraph, the dispatcher is simply not needed. By just determining statically the most generic types of the parameters it is possible to determine uniquely, and entirely statically, the right multimethod to use. This leads to a massive

improvement in the call time, and, more generally, to a drastic simplification of the dispatching calls and the related tables.[12]

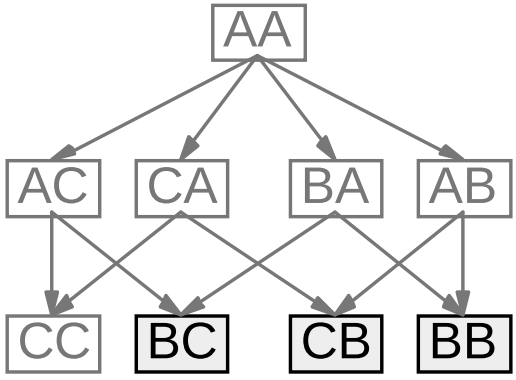


Figure 3.2.: Static resolution of multimethods

The point of the discussion is that, although the current test implementation is not optimised in any way, the overall language definition is designed to allow a very efficient implementation, likely to be comparable in speed to C++ and similar languages, despite the use of multimethods and its being a "pure" object oriented language. The fact that all of the data data usable in the language has to be treated in the form of objects of equal behaviour, instead, opens the door to a variety of different implementations, suitable to different circumstances, making the language a perfect candidate for the realisation of an orthogonally persistent system, as this project shows.

[12] The example refers to a small hierarchy in which a class A has two subclasses B and C – the ability to distinguish statically between the methods is retained, for instance, adding further unrelated subclasses to B and C. This is due to the fact that statically it is always possible to discover whether a class is a subclass of B or C, and therefore only one of the three methods is usable, whatever the complexity of the hierarchy, as long as B and C do not have common descendants.

4. Design

The work described in this report aims at the creation of an orthogonally persistent implementation for the language defined. This chapter contains an overview of the required work and an analysis of the possible design choices. The motivations behind the adoption of some technical solutions versus others will be explained. The following Chapter 5 will show how those ideas have been put into practice to obtain a working implementation.

Figure 4.1 summarises the preexisting architecture of the test implementation, and the structure of the new system. The central idea is to replace the simple memory manager with a more general purpose and persistent support.

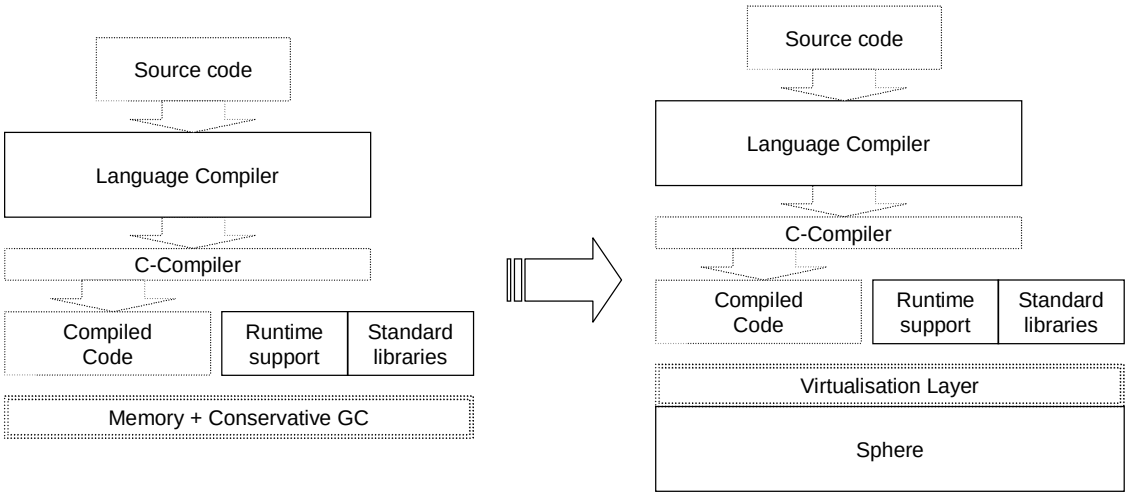


Figure 4.1.: The new system structure

The new support is composed by the persistent store, Sphere in this case, plus an intermediate layer, which adapts the calling interface of the store to the needs of the language, offering additional services. This intermediate layer is called the “Virtualisation Layer,” since its function is to decouple the generated code and the runtime support from the specific details of the store in use and the architectural details of the host operating system, offering therefore a “virtual” environment which can support the execution of the running

program. To achieve a better modularity, the new layer is not specifically targeted to the specific language here described, but its design should be general enough to be reusable in other contexts. The previous implementation of the compiler needs to be modified, although not dramatically, to conform to the new calling conventions dictated by the new execution environment and to implement a new support for recovery.

Section 4.1 describes the set of functionalities required from the new Virtualisation Layer, and the design choices that have been subsequently adopted during its development. Section 4.2 describes the modifications that are needed by the existing compiler, and the which of its components need to be revised. In this chapter, the term “user program” will refer to the source program written in the source language, while the term “user code” will refer to the output of the compiler, or to an interpreter or a virtual machine running the same program. The entire discussion assumes the use of persistence by reachability.

4.1. The Virtualisation Layer

The purpose of the virtualisation layer is to offer a simple interface that allows the running code to use system specific features (object store, memory pool etc.) in a simple way. In other words, the virtualisation layer is the component which offers the required set of system functionalities to the code produced by the language compiler while insulating it from the actual object store in use, the algorithms used for the handling of the memory space, etc. From the point of view of the running programs, this is the only visible interface; therefore, the set of calling conventions must match the needs of the user code. Given that both parts (the virtualisation layer and the modified compiler) were designed more or less simultaneously, the interface between the two could have been structured in a variety of ways, according, at least, to the following criteria:

1. the memory management could be enclosed in the virtualisation layer or delegated to the user program or to the runtime support specific to the language.

2. consequently, the layer could offer the programs a mechanism to load/unload objects or offer, instead, objects already mapped in memory

3. the access to the memory could take place in terms of a unique memory space or maintaining the individuality of objects

4. the unit of manipulation of memory buffers could be a memory page or an object

5. the checkpoint operation could be explicit or performed automatically by the layer

6. the compiled program could be requested to perform residency checks on every access to pointers (to ensure that the referred object is already available in memory), or the virtualisation layer could set up a trapping mechanism to detect all accesses to objects not yet resident, and perform transparent loading on demand.

For what concerns the management of the memory space, not all the stores available are capable of handling autonomously memory buffers, and in this specific case Sphere, while offering a very useful set of primitives to perform swizzling and unswizzling of pointers inside memory blocks, does not manage the allocation or disposal of main memory. On the other hand, automatic memory management is a common requirement for most of modern languages, and it is indeed required by the language defined. Therefore, it definitely did make sense to incorporate some form of memory management into the virtualisation layer.

The loading/unloading of objects can be managed by the user code (by explicitly separating the operations of memory allocation and transfer from/to the object store) or automatically performed by the store, so that objects are presented to running programs as memory blocks and the mapping between persistent identifiers and memory addressed can be potentially delegated completely to the underlying layer. While the first option would allow user programs to use memory independently from the actual use of objects, therefore offering a better support for languages which are not completely object oriented, the second choice simplifies the automatic swizzling/unswizzling of pointers, which can be performed in any case autonomously by the virtualisation layer. The actual details of the algorithm used to perform those operations remain hidden from the user code, and can be replaced subsequently if necessary. The result is a simplification of the calling conventions needed to use the intermediate layer. With the intent of keeping the code produced by the compiler as simple as possible, the second option has therefore been selected.

Access to the memory can be offered in the form of a flat, single memory space or instead keeping track of objects and their location. The first option is especially suitable for legacy systems, in which it is difficult to know in advance the structure of data, the location of pointers and their pattern of access, while the second option imposes a more restrictive usage, requiring the user code to follow stricter rules while creating and using memory objects. Notably, the first technique is often used in a class of persistent operating systems known as Single Addressing Space Operating Systems [Voc98][SM98b], in which there is essentially no distinction between pointers and persistent identifiers.

Although the first option is much more general, the aim of this virtualisation layer is to offer support primarily to object oriented languages, in which the structure and identity of objects are well known. Since keeping track of objects allows a much simpler internal implementation, and a straightforward mapping of the objects used by the user program on

the data structures used by the store, potentially enhancing efficiency, the second choice has been preferred.

Deciding whether to use an object or a page, as a basic memory manipulation unit, is a related issue. Some stores are based on pages, while others on objects. Usually the store organisation reflects the intended scope of use for the store: handling pages is usually preferred in systems targeted to the porting of legacy systems, while handling objects is generally related to systems based essentially on objects, as are most modern programming languages. Even in this second case, however, there is often some form of interaction with the page structure dictated by the underlying hardware architecture.

Almost all of the microprocessors commonly available on the market, in fact, organise memory in pages of a fixed size, usually of a few kilobytes. Consequently, the page is often also the smallest unit of memory on which it is possible to specify individually access restrictions. This influences the algorithm used for object eviction, since the hardware can only assist while detecting if an entire page has been changed, but not a single object in the case in which a page contains more than one object. Given that the average size of an object in a system can be much smaller than a page size, the implications on the store architecture can be significant. A similar argument holds for disk space, which is organised in disk blocks.

The virtualisation layer described should be as general as possible, and on the other hand it is the underlying store that ultimately dictates the concrete unit of storage, which means that neither of the two choices (pages vs objects) for the layer interface can be ideal under all circumstances. To be coherent with the previous choices, however, and to keep the interface simple, it seems reasonable not to request the user code to deal with the actual subdivision of memory in pages. The allocation unit will, therefore, be the object, although the virtualisation layer is left free to manage pages individually in the internal implementation.

The checkpointing operation could be explicitly accessible via a specific call or instead performed automatically by the layer according to some criteria, such as the lapse of a timeout since the last checkpoint, or a certain number of object updates in memory. Allowing the generated code to perform explicit checkpoints can be useful, for instance, during the initialisation stage, to ensure that a proper initial stage from which to recover the state of the program is stable in the store. On the other hand, without an automatic form of checkpointing, the virtualisation layer would have to rely entirely on the running program for a proper periodic checkpointing, either with some construct available in the language or through calls embedded by the compiler into the generated code. The risk would be to run out of memory buffers to hold the modified objects if the checkpointing call is not invoked frequently enough.

The two options, however, are not mutually exclusive. It is possible to let the virtualisation layer perform automatic checkpoints when needed while still preserving the option of an explicit call to ensure a complete stabilisation of the store at critical points. This choice is made for the current design of the virtualisation layer. It could be argued that, since automatic checkpoints are performed unknowingly to the program, the user code must constantly take care of maintaining the state of all the objects in a situation from which a recovery is possible, which can be rather difficult. To reduce the complexity of the problem, part of the calling convention will specify a finite set of conditions in which the virtualisation layer can decide to perform autonomously a checkpoint, allowing therefore the user program to proceed in discrete steps between "critical points," which are the ones in which the layer can act automatically.

An alternative solution could be not to let the virtualisation layer perform automatic checkpoints, and just to abort computation in an emergency situation. In that case, though, the user code would be required, as mentioned, to perform frequent calls to the checkpointing routine, possibly much more frequent than necessary, therefore reducing efficiency. The checkpointing calls would represent, in that picture, just another aspect of the same subdivision of the execution flow in small steps. Avoiding unnecessary checkpoints is probably worth the little extra effort needed to take care of the situations in which an automatic checkpoint can occur.

The last point concerns the need to have objects actually resident in memory when they are accessed. The two possible options are either requiring the user code to perform residency checks before every pointer dereference or instead to use a transparent mechanism to perform the loading of objects on demand. The second option requires less operations during the normal program executions, but can only be used if the virtualisation layer has control over the memory space. Since this is our case, according to the choices already made, the latter option is certainly viable and, as an added advantage, could help to reduce the complexity of the code produced by the compiler. It is therefore the choice adopted in this design.

Summarising, here is the set of design choices:

- the virtualisation layer manages the memory automatically
- objects are presented to the user code as memory chunks
- the unit of memory management, as far as the user code is concerned, is the object
- objects are automatically loaded/evicted, and the user code does not need to perform residency checks

- the user code does not need to perform explicit checkpoints, but has to be aware that automatic checkpoints can occur under well defined conditions. The option of performing explicit checkpoints is still available.

As additional, desirable characteristics, the virtualisation layer should moreover:

- be easily modifiable to accomodate different underlying stores to be used
- be, wherever possible, independent from architectural details like page size, 32/64 bit addressing, etc.

Acccording to this set of guidelines, the existing implementation of the language compiler must be adapted to use the virtualisation layer as the main component of the runtime environment.

4.2. The Language Compiler

The previous implementation of the compiler worked by processing a source file written in the source language and generating C code, which was then compiled, together with a standard library and a minimal runtime support, to produce an executable program. The resulting code allocated objects exclusively in memory, and did not offer any form of real persistence. To offer a minimal form of automatic memory reclamation, the code was linked with the Boehm-Demers-Weiser conservative garbage collector, which, although not an exact garbage collector, was suitable to the needs of the test implementation.

The minimal alterations to be performed on the existing compiler concerned, therefore, the removal of the Boehm garbage collector, the substitution of the memory allocation routine with primitives offered by the virtualisation layer, and the transfer into objects of all the system structures needed to fully recover the state of the running program. The part of the compiler which produces the actual code needed modifications to accomodate the new style of calls used by the newly designed virtualisation layer. Furthermore, the standard runtime code, which offers the set of primitive classes and methods, needed similarly to be adapted to the new environment.

Apart from the necessary changes, several other modifications or improvements could have been introduced in the compiler, like the change for the output format from C source to some other format, or the substitution of wrapper objects, previously used for primitive types, to increase efficiency. The possible design options ranged therefore from minimal changes to a complete overhaul of the existing compiler. Mainly due to time constraints, the choice adopted was to introduce, at this stage, as little change in the existing code as

possible, so as to obtain in a short time a prototype relying on the logic and the code of the existing implementation. That decision avoided the risks and the possible high time penalty involved in the creation of a brand new implementation, or in radical changes on the existing code. It has to be noted that the option of applying improvements to the compiler remains open for further developments of the project, especially given the genericity of the virtualisation layer, which can be used, if needed, to accommodate a completely different compiler implementation.

To perform the required changes to the compiler (system structures into objects, user code adapted to the virtualisation layer and modified libraries) the work could have been potentially very complex. Fortunately, a modular architecture was already in place in the existing implementation, and therefore the set of changes were limited to a few modules. Figure 4.2 shows a diagram of the structure of the previous implementation, with an indication of the parts that needed to be changed. In particular, the code generator works is syntax-directed, which implies that, while the actual code generation steps had to be revised, the overall structure of the parsing productions was left unaltered.

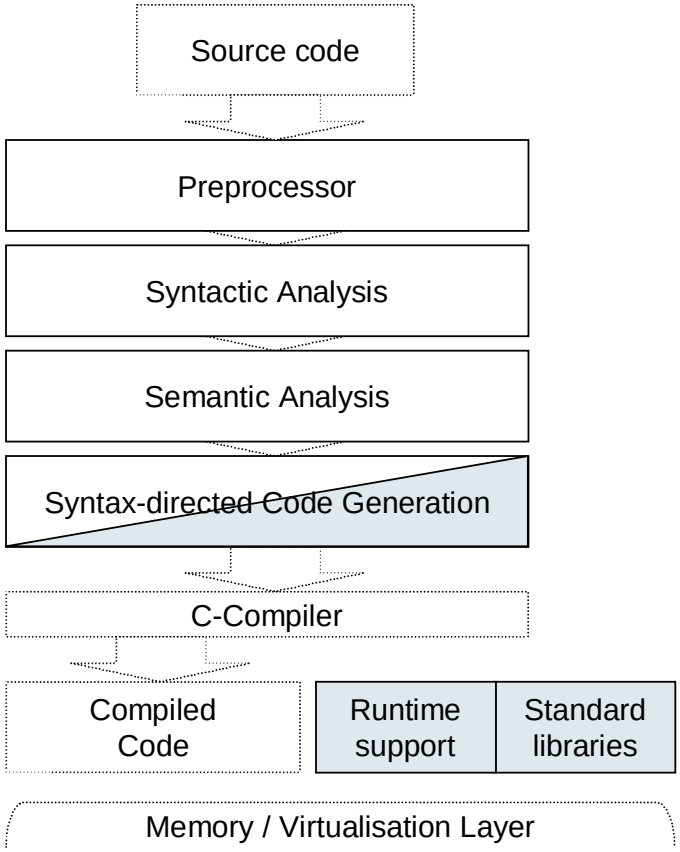


Figure 4.2.: Modifications required by the compiler

The next section contains a general description of the concrete implementation choices and the main algorithms adopted. It should be noted that the current implementation has no pretense of being a complete and efficient system. Instead, it should be considered an instrument to show that a certain result can be obtained. Since the need to obtain a working prototype was greater than achieving the maximum efficiency, many aspects related to performance have been carefully considered and then happily ignored, both during the develpement of the virtualisation layer and that of the language compiler. It is worth pointing out, however, that the architectural choices and the design of the language have had the efficiency factor as one of their main driving aspects. An alternative, more efficient implementation can be obtained while maintaining the same techniques and overall structure.

5. Implementation Overview

5.1. The Virtualisation Layer

As previously described, the virtualisation layer must manage memory allocation and offer a transparent support for the loading on demand of objects, their eviction and automatic checkpointing. The obvious way to achieve such a result is to use the native, hardware assisted facilities of memory protection, offered by all of the modern hardware architectures. For instance, by imposing that a page of virtual memory cannot be read or written it is possible to obtain an exception for every attempt to access the corresponding memory area, and therefore transparently loading the corresponding object or objects before resuming execution. Additionally, by protecting a page from write attempts, it is possible to force an exception to be taken whenever an object is modified in memory, so as to keep track of which among the objects resident in memory needs to be saved in the store before freeing the memory buffer for later reuse, which helps avoiding unnecessary updates on disk for those objects which have been only read since the last loading, increasing therefore the overall efficiency.

It would have been rather complex to access the hardware memory management unit directly. First of all, accessing the MMU would have required a detailed study of the specific hardware details of the platform in use, with the possible need to introduce very system specific code written in assembler language, and a strong limitations to the portability of the resulting software would have followed. Furthermore, the development of the system needed to take place in the context of a host operating system (a Unix variant, in this case), and in all modern operating systems there is already a native mechanism of virtual memory which uses the MMU, preventing its direct use by user programs. To avoid those problems, the only viable alternative was to use the system calls mmap()/unmap(), which, although generally used to perform file mapping operations, can be used alternatively to protect arbitrary areas of the user virtual memory space, in a way suitable to implement the protection technique described. The downside is a little penalty in performance, due to the extra overhead imposed by the operating system.

The next step was to decide which mechanism to use to manage the actual loading/unloading

of objects, and their automatic swizzling/unswizzling. After a brief investigation, the more promising and simple technique appeared to be to use “pointer swizzling at page fault time”, in a way similar to the one described in relation to the Texas Store[SKW92].

The basic idea is rather simple: whenever an object is requested for the first time, its persistent copy is loaded from the store and all the references to other objects (in the form of persistent identifiers) that are contained in the object itself are checked. For all the references to objects not already resident, an area of virtual memory of the size of the referred object is preallocated, but not attached to any physical memory, and the persistent id is replaced by the pointer. As soon as any of those “ghost” objects is accessed, an exception is thrown, the object is loaded and again its references scanned, and so on. The following diagram can help to clarify the mechanism.

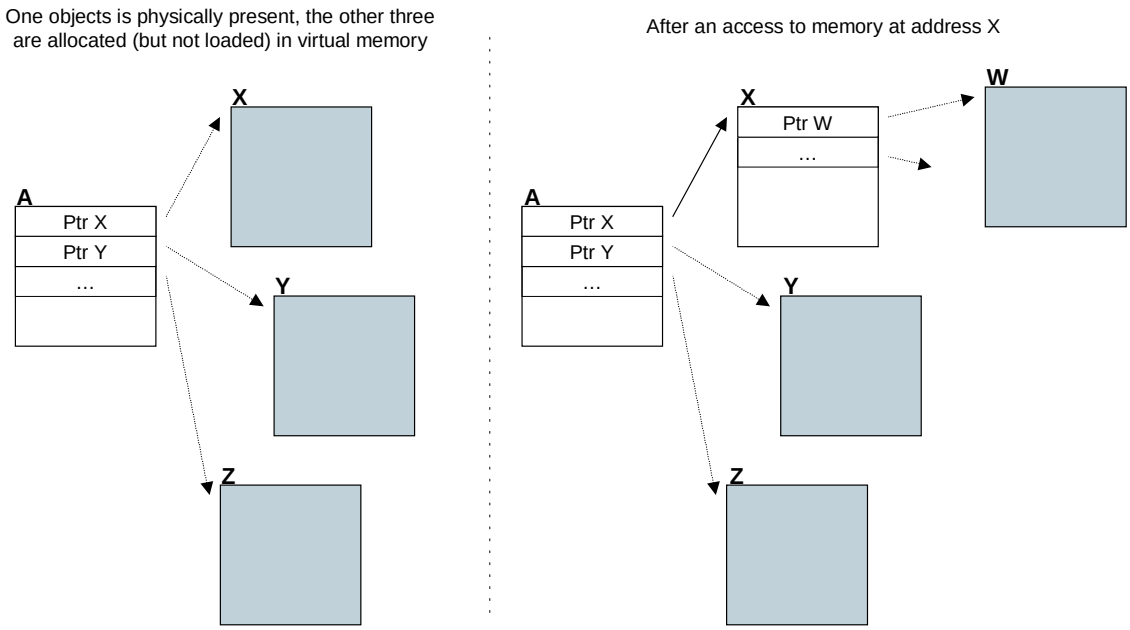


Figure 5.1.: Pointer swizzling at page fault time

When location X is accessed, the corresponding object is loaded from the disk storage and its references to further objects checked. Right after the object is loaded from disk, its references to other objects are in unswizzled format, that is in the form of persistent identifiers of other objects. Each of the references is checked against a system table in which the association between persistent identifiers and virtual memory addresses is kept. If the object referenced is already present, its address is simply substituted to the persistent id, otherwise an area in virtual memory is preallocated but not mapped onto any physical memory space,

and the newly created association between virtual memory address and persistent id is added to the system table.

The starting point of all the chain is the persistent root: to initialise the algorithm it is sufficient to preallocate the space of the root object and to store in the table its address in memory and its pid. The first access to memory must happen inside the root, therefore the first object will be loaded and swizzled and from this point on the algorithm will work as previously described. While the preallocation of the space for referred objects is done immediately, the actual loading happens "lazily", that is only when the objects are used.

The algorithm described for the Texas Store performs actually the loading and swizzling operations in terms of pages, while in this implementation, since most of the work is done on objects, the preallocation of memory buffers takes place one object at a time; the principle remains the same in both cases. There are some implications on the efficiency, since at the hardware level the architecture is based on pages; the issue will be discussed soon.

From the description of the algorithm, it follows immediately that the virtualisation layer needs to deal with three different kinds of storage: virtual memory space, physical memory and disk storage. Each of these three resources is finite, and the need for the recycling of space in each of them has to be kept into consideration. This subdivision leads logically to a first organisation of the virtualisation layer on three different levels, each of which has to deal with one specific kind of storage and is implemented by a distinct software module. In addition to those modules, other components can be defined to handle different resources: class definitions (the object structures must be known to perform correctly the swizzling/unswizzling operations), error conditions, hardware dependent aspects, and system tables. In the following diagram the main components and their relations are represented. A description of the function of the various parts will follow, while a more detailed explanation of the interface available to the user code is available in Appendix A.1.

While this kind of organisation is not the only possible one, it proved to be quite effective to keep the overall complexity of the project low, and to simplify the maintenance of the code during the development.

5.1.1. Virtual Manager

The Virtual Manager is the most external layer, and exports towards the user code the main functions needed to allocate memory, in the form of objects and classes. To perform the actual initialisation of objects and classes, the Virtual Manager makes use of the routines offered by the Physical Manager and the Class Manager, described below. In the current implementation, the virtual memory space is never reused, and therefore the user code can be informed, through the Error Manager, that an exception has occurred if the available ad-

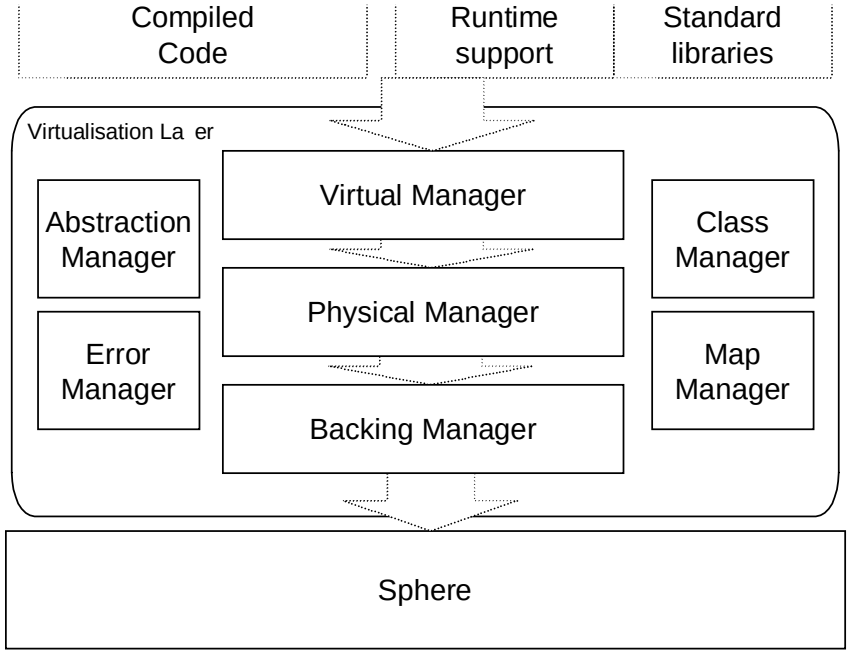


Figure 5.2.: Virtualisation Manager

dressing space is exhausted. The user code can then perform a checkpoint of the current state, ask the virtualisation layer to reinitialise, and resume the computation as if nothing happened. An alternative implementation could reuse the virtual memory space incrementally, and some interesting ideas about applicable techniques are described in the work by Wilson and Kakkak in reference to the Texas Store[WK92].

5.1.2. Physical Manager

The Physical Manager controls the use of a pool of physical memory pages, which are attached as needed to the different areas of virtual memory allocated by the Virtual Manager. The binding of a physical memory area to a virtual address can take place explicitly during the creation of a new object, or automatically if an access to an unmapped address of virtual memory is detected. The Physical Manager has therefore the responsibility of handling memory fault exceptions, and keeping track of which objects are allocated but not resident, resident but not modified since the last checkpoint, or modified and to be updated on disk.

Added functionalities of the Physical manager include the handling of objects which are guaranteed to stay permanently in physical memory (“permanent objects”). Those objects

are always treated like multiple persistent roots and are guaranteed not to generate implicit checkpoints when accessed, which makes them ideal to store system structures. At the lower level, the availability of a single persistent root is required from the disk store.

The Physical Manager is able to detect if the available physical memory is exhausted and to evict objects in order to accommodate new ones, according to the needs of the running program. The choice of the victim for eviction is delegated to the Map Manager. In the current implementation, the victim is transferred to the object store in any case, even if it is not reachable from already persistent objects. This is, of course, a source of considerable inefficiency, since it implies many unnecessary writes on disk of objects that are not reachable anyway, and extra work on the disk-based garbage collector of the object store. Nontheless, the present implementation can be adapted with relatively little effort to detect which objects are certainly not reachable while in memory and therefore do not need to be transferred to the object store.

5.1.3. Backing Manager

The Backing Manager is the only component that has knowledge about the specific details of the object store in use. It offers calls for the preallocation of a new object (obtaining a new persistent identifier), to set and get properties about the persistent root, to load an object from the store into memory and to write it back. A special function is used to inform the Backing Manager that a new object is initialised and ready to be written for the first time.

All of these calls constitute the only interface through which the other modules are allowed to access the disk store. Internally, the calls can be remapped onto the native functions and calling conventions required by the store as needed, offering enhanced flexibility. The current implementation of the virtualisation layer uses Sphere as its concrete object store on disk, but only the Backing Manager has knowledge of the way in which Sphere works. In particular, the function used to declare a new object as initialised is not directly connected, as it could appear, to the algorithm used by Sphere to perform fast first writes[PAD98]. The newly created objects are in fact kept exclusively in memory even after the Backing Manager has been told that the objects can be written for the first time. This allows the user code to alter the more recently created objects in memory again and again, until there is the need to perform a checkpoint. At this point Sphere is invoked, and the actual first writes of the pending objects are performed, reducing the total number of disk writes required. In other words, the calls which are offered to the upper levels by the Backing Manager and the real operations performed on the store are decoupled, and there is no preestablished connection between the two interfaces. In this sense, the Backing Manager acts like the lowest level “wrapper” around the disk based store, hiding completely the real architecture of the latter.

The checkpointing logic is managed primarily by this module, since its behaviour is closely related to the structure of the underlying store, but some information is obtained from the Physical Manager using its accessible interface (no global data is shared between modules, which enhances the independence of the respective implementations). The checkpoint operation ensures that the current state of the objects is safely stored on disk, and through the Physical Manager, marks all objects currently in memory as updated. From this point on, the Physical Manager will keep track of the objects that have been modified at least once, plus all the newly created ones, so as to prepare a new list of objects that need to be saved during the next checkpoint.

5.1.4. Class Manager

The Class Manager is used to define the structure of objects, which has to be known to the system to enable the correct swizzling/unswizzling of references when required. To enable other modules to perform operations on the references contained in an object without having to be concerned with the actual object structure, the Class Manager offers suitable iterators that perform the scanning of all the references in an object, or a subrange of them, invoking a callback routine. The current call used to define a new class is rather simplified, and assumes that all of the references are actually grouped at the beginning of the object. A more generic definition can however be added to the current one without prejudice for the existing interface. Once created, the classes can be accessed only through their class identifiers, which remain unchanged if the program is stopped and resumed. There is currently no support for modifications in the class structure once the class has been created, and no provisions for evolution have been included at this stage.

5.1.5. Error Manager

The Error Manager includes a generic interface to handle errors which could be generated in various circumstances by the system. A list of error identifiers is provided, as is a function that returns the error message in a textual form given the error id. The default behaviour of the Manager in case of an error, which consists of printing the error message and aborting execution, can be overridden by installing a custom error handler, useful to react appropriately to some recoverable error conditions.

5.1.6. Abstraction Manager

The Abstraction Manager is a simple utility module which encapsulates some hardware specific aspects, mainly by offering definitions of standard types which are guaranteed to have

a known size in memory (integers, signed and unsigned, whose size is exactly 8, 16, 32 and 64 bits), and defining implementation-specific types for pointers, persistent identifiers and references (whose type is essentially the union type of the previous two).

5.1.7. Map Manager

The Map Manager is used to maintain some of the main system structures essential to keep track of the objects in use, their address in the virtual memory and other attributes. The three maps currently managed are as follows:

- the association between persistent identifiers of objects and their virtual memory address
- the list of the objects which are currently resident in physical memory
- the list of critical objects which are marked as "permanent" and are not eligible for eviction (although their state is saved together with the state of all the other objects during a checkpoint)

Since this module has knowledge of all the resident objects, their position and their size, the Map Manager is the perfect candidate for handling the task of selecting a victim for eviction. The current simple algorithm used, which is probably not the best one, selects as a victim the object that has been resident in memory for a longer period of time.

5.1.8. Objects and Pages

A critical issue is the actual mapping of objects into pages, since the hardware assisted control of memory access is usually available only for entire pages, whose typical size may be considerably bigger than that of an object. There are several techniques that can be used to remap objects onto pages. The first, and obvious one, is to use an integer number of pages for every object, whatever the object size. While this very simple technique allows us to detect individual accesses to objects, the side effects are a waste of virtual (and physical) space, since large portions of each page can remain unused, and additionally a significant load on the memory management unit, since an independent mapping has to be established for each object, which reduces the effectiveness of the MMU translation cache. On the other end, a strategy finalised to obtain the maximum occupation of pages could try to pack together objects, or parts of them, in each page. The effect, in this case, is that it is no longer possible to use the page faulting mechanism individually for each object. Once a page has been accessed, all of the objects preallocated in the same page would have to be loaded

(and swizzled) at the same time, even if they are not needed by the present computation. In alternative to those two solutions, a more complex strategy, involving objects moving in memory, indirect pointers or other techniques, could to be used. An intermediate approach can be to use the MMU to map a single page of memory multiple times using different virtual addresses, one for every object. The objects in the same page could therefore be loaded all in one step, but the actual swizzling (and preallocation of the referred objects, with relative binding of further virtual memory addresses) can be postponed to the actual access time of the individual objects. Still, other solutions are possible, although there are obvious limits on the effectiveness of a memory management unit designed to handle pages when used to manipulate objects. In this specific case, to keep this implementation as simple as possible, the first, naïve technique has been used, although it would be definitely advisable to adopt a more sophisticated approach in a successive implementation.

5.1.9. Automatic Checkpointing

As previously mentioned, the design choice was to allow the system to perform automatic checkpoints under well defined conditions. That choice relieves the user code from the task of explicitly invoke the checkpointing operation, and permits the virtualisation layer to automatically reclaim buffer space, should there be no more free memory available. On the other hand, the user code must have a certain degree of control about when an automatic checkpoint can occur, so that the checkpointed state of the objects can be kept consistent and usable for a possible restart of the execution.

As previously described, any access to memory can potentially trigger a page fault exception, to which the system responds by automatically loading the object that has been preallocated on that memory area. During the loading routine, therefore, the system could discover that there is no longer any physical memory available, and that one or more evictions of objects currently in memory are necessary. The point in the code where the eviction takes place is a perfect candidate to call, every once in a while, an automatic checkpoint: the routine gets called ofted, it is invoked transparently, its occurrence is synchronous (when it happens) with precisely defined operations in the user code. The only other obvious way to obtain automatic checkpoints, having a separate thread performing the operation, would be much less straightforward and would require some form of explicit synchronization between the thread and the user code.

The downside is that, if no other mechanisms are in place to control when the checkpoint can occur, potentially in every point in the user code where an access to memory is made, a snapshot of the objects state could be taken. The effect is that the user code would have to ensure that there is a consistent condition during every access to memory, which is clearly

too strict a requirement to be practical. A simple solution, however, can be obtained by introducing objects that are guaranteed to stay permanently in memory, and that therefore do not generate memory faults (and, consequently, implicit checkpoints) when accessed. By storing critical system structures inside permanent objects, it is possible to reduce significantly the number of occurrences of possible implicit checkpoints, hence maintaining consistency in the critical points while keeping the user code reasonably simple. Another factor that can help to organise the user code to maintain consistency in the state checkpointed is that during a memory write it is very easy to guarantee that the (possible) automatic checkpoint will be completed *before* the actual modification of the memory content. To be precise, a memory write can trigger a page fault exception. If that happens, an automatic checkpoint may be performed by the exception handler. The handler has just to wait for the checkpoint to be completed, evict some objects, load the object requested and return to the user code, to the same instruction that caused the exception in the first place. It is only at this final stage that the real memory write will take place. It is therefore known to the user code that, if an automatic checkpoint happens while a write is performed, it is enough to ensure that the program can recover from the saved state as it was before the memory write.

Given the mentioned criteria, a range of possible options become available to ensure that any checkpoint will save a snapshot of the object's state in such a way that an easy recovery is possible. The simple schema suggested here, which is the one used in the current adaptation of the language compiler, works by keeping a "program counter" inside a permanent object, in the following way:

…	
• reads / computations	
• reads / computations	
• reads / computations / write	during the final write, an automatic checkpoint may happen *before* the actual object modification
• PC changed to "n"	the program counter is in a permanent obj -> no checkpoint
jump_point_n:	if the execution flow arrives here, the PC is in synch with the new state of the object
• reads / computations	any automatic checkpoint in these instructions would save the changed object and the new PC together
• reads / computations	
• reads / computations / write	a checkpoint may happen *before* the new object modification
• PC changed to "n+1"	no implicit checkpoint!
jump_point_n+1:	again, if the prog. arrives here the PC and the new objects state are in synch
…	

Using this technique, any automatic checkpoint that should occur at any point in any of the blocks will save the program counter together with the state of all the objects modified up to that point. Since the following read operations from objects do not modify any internal state, the execution can be safely resumed from the matching jump point, should the program execution be interrupted for any reason. This very simple but effective mechanism allows, in a straightforward way, to make sure that it will always be possible to resume the execution from a consistent, checkpointed state by just jumping to the entry point corresponding to the saved copy of the program counter. For a practical example of how the actual code generated by the compiler looks like, we refer the interested reader to the example code in Appendix A.2.

Summarising, an automatic checkpoint can take place every time an object eligible for eviction is accessed, or when a new object is created. By saving system critical information in a permanent object, it is however possible to organise the generated code to always have consistent system conditions saved by any automatic checkpoint. If needed, the user code is also free to invoke an explicit checkpoint as often as desired.

5.2. The Language Compiler

According to the previous discussion, the language compiler had to be changed to accommodate the new runtime environment, as mentioned in Section 4.2. The three components that had to be changed are the Code Generation module, the Runtime Support and the Standard Libraries. A description of the modifications applied to those modules will follow.

5.2.1. Code Generation

The component that produces the output code from the compiler (in the form of C source) is built using lex and yacc, with C fragments. That implies that the crucial part of the code generation is performed by C code embedded into the productions LALR(1) used by the parser[ASU86], in a syntax-directed way. Since the running environment is now different, the semantic actions inside the rules have to be changed accordingly; however, the overall structure of the rules can remain largely unchanged. The differences in the new code that must be produced, with respect to that produced by the old compiler, are mainly the following:

- the language stack was previously implemented using the C stack; this has to be changed to implement recoverability (see below)
- consequently, the method calling sequence must be modified

- a program counter and a jump table have to be added to allow the execution flow to restart from an internal point in the user code following a resumption
- the object allocations must be changed to use the new routines, and the virtualisation layer must be informed about the structure of the classes used in the program

The previous version of the compiler worked by transforming every method of the language in a C function, so that the handling of parameter passing and recursive calls was handled by the C compiler. That kind of approach is no longer possible, mainly because the stack may need to be saved together with all the other objects, and possibly restored following a recovery. Extracting the data from the native C stack would be quite complex and dependent on the hardware platform, therefore the obvious choice is to transfer the stack into system objects. In this way, the stack state is saved automatically whenever a checkpoint takes place. The handling of the stack frames must be made explicit, in that, as in any ordinary compiler implementation, the parameters are pushed on the stack just before the message dispatching, and the return value has to be extracted from the stack at the end. The local variables are now kept in this new implementation of the stack, and allocated/deallocated at the beginning/end of every method. The stack is maintained as permanent objects which, as mentioned, also work as persistent roots. This implies that the objects which are preserved in the store are all those reachable, directly or indirectly, from any of the stack frames.

The code produced is now divided (from a logical point of view) in basic steps, following the structure described in Section 5.1.9, and a jump table is built during the code generation. The logical program counter maintained by the user code is stored in an additional permanent object, named “control block” together with some other system critical infomation (notably, the stack pointer). The initialisation of all the classes related to the user program is performed by an appropriate function, which is invoked once during the first run of the code.

5.2.2. Runtime Support

The runtime support offers a few support routines needed during the execution of the user code, the most important of which is the message dispatcher. Only minor modifications were required by this module, the most relevant being the introduction of a specific initialization routine necessary to set up properly the virtualisation layer before the actual user code execution. If the program is running for the first time, the virtualisation layer is opened requesting the creation of a new store, the standard classes are added to it and the system objects allocated; if, on the other hand, the program is resuming a previously interrupted execution, the existing store is reopened and the runtime context restored, after which the execution can restart from the entry point corresponding to last saved program counter.

5.2.3. Standard Libraries

The standard libraries offer a set of predefined classes and methods, ranging from functions needed for string manipulations and arithmetic operations to basic input/output. The code implements those basic functions using hand-written code that respects the calling conventions imposed by the compiler. Being tightly coupled with the implementation aspects, this module had to be rewritten almost entirely, mainly to conform to the new structure of the stack and the new conventions concerning the parameters and the return value. To avoid many repetitions of similar code, especially in the arithmetic routines, and the introduction of possible inconsistencies, the implementation of the standard libraries relies massively on macros, which allow the code to remain reasonably manageable without impacting on the execution speed. In this case, the structure based on macros helped to reduce the time necessary to rewrite the necessary routines.

6. Conclusions

6.1. Tests

The result of the programming activity described in Chapter 5 has been a software system composed by language compiler and persistent runtime environment, structured in accordance with the design guidelines discussed in the previous chapters. To verify that the system concretely implements a multimethod-based orthogonally persistent language, as expected, a certain number of tests have been performed on the platform. Some test programs are listed in Appendix A.2.

6.1.1. Language Tests

The system does not implement in every aspect the language definition of Chapter 3; nonetheless, a rather large set of features has been implemented and verified using test programs. In particular, tests have been used to verify the functionality of classes, methods, messages, inheritance, multimethods and dynamic dispatching, static type checking and the set of rules used to avoid ambiguities described in Sections 2.3.2 and 3.14.1, constructors, control structures, type tunnels, arithmetic functions, multidimensional arrays, user defined operators and the other syntactic aspects.

6.1.2. Recoverability

To check the ability to continue in conditions of exhaustion of virtual memory space, the virtualisation layer has been recompiled allowing only a very small portion of the addressing range to be used. Some test programs have been then run against such an intentionally reduced memory space. The result, as expected, has been a rapid exhaustion of the available addressing space and the generation of an exception from the virtualisation layer. Following the technique described in Section 5.1.1, the running programs have then automatically performed an emergency checkpoint and a reset of the system structures, followed by an automatic resumption of the program execution. As expected, the user code is therefore

capable of continued execution even when the virtual memory is scarce.

The other crucial test about recoverability consisted in running arbitrary programs and interrupting their execution at random times. In all the tests performed, without exception, the system has been capable of resuming the execution of the running program from the latest automatic checkpoint, showing therefore rather effectively how the recoverability aspect of the system implemented is fully functional. Some examples of interrupted and resumed executions are listed in Appendix A.2.

6.2. Possible Developments

The system developed in the context of this project is essentially an instrument to verify the applicability of some ideas and techniques. As such, its current implementation is not optimised under several aspects. For instance, the mapping between objects and pages is too simplistic and inefficient, and a better approach would be advisable.

An important improvement in the allocation of objects and their transfer in the store would be to implement a policy for temporary objects. While currently all objects, including temporary ones, are preallocated and saved in the persistent store, delegating the garbage collection to the store itself, a much better performance could be obtained by allocating newly created objects exclusively in memory. During the eviction stage, a scan of the objects that were already stabilised and have been modified since the last eviction stage could be used to determine which transient objects are now reachable from already persistent ones, either directly or through other transient objects, and therefore need to be promoted to persistent by allocating and writing them in the store. Obviously, those objects that are not reachable from persistent ones can be safely discarded, therefore reducing the number of objects actually transferred to the store and increasing the overall efficiency. Some improvement in performance should be obtainable just by introducing this last technique, and replacing the simple data structures used in the present implementation (arrays and linked lists) with more proper structures like hash tables.

Another area of possible improvement is the language compiler, which still lacks some interesting aspects of the language, the most important being multiple inheritance. The language itself could be redesigned on a more formal ground, possibly including advanced features like parametric polymorphism, functions as first order objects etc.

The system, as it is at present, relies on a number of different software components, each of which has different needs in terms of configuration of the host environment.[1] This, of

[1] For example, Sphere is compiled using the Sun Workshop Compiler while the language compiler relies on features offered by the Gnu C Compiler, some scripts depend on csh and some others on bash and so on.

course, makes the task of assembling a distribution for general use rather challenging, and a prepackaged installation kit is not, at the moment, available. For the same reason, a complete user manual of the system is not included in this report. The non orthogonally persistent version of the language compiler, however, is available for both Linux and Solaris and can be installed with relatively little effort. If any reader would like to experiment with that version, or would like to go through the task of setting up the environment for the orthogonally persistent version, the author will be more than pleased to offer all the software and assistance necessary. More information can be obtained by writing to cuneia@dcs.gla.ac.uk.

6.3. Evaluation and Conclusions

All the tests suggest that the system is fully functional with respect to the features that have been implemented, and in particular for what concerns multimethods and static type checking, orthogonal persistence and recoverability. While the present implementation does not meet the levels of performance and usability that would be necessary for a production tool, the system appears nontheless to be remarkably stable and functional, and can be considered a good indicator of the validity of the ideas and the techniques proposed. In particular, being a working implementation of a compiler for a multimethod-based, strongly and statically typed orthogonally persistent language, it shows, with a practical example, how multimethods can be used in the context of an orthogonally persistent system.

6. Conclusions

A. Appendix

A.1. The Virtualisation Layer: User Interface

The Virtualisation Layer can be accessed using its user interface, which is briefly described in this appendix. The codename used during the development is "Ahio",[1] therefore the same name is recurrent in the interface. All the constants and the function prototypes are available including the header "Ahio.h".

A.1.1. Types

The relevant types exported are `ptr`, `ref`, `classID`, `permID` and `classSize`.

1. ptr is a generic pointer.

2. ref is a reference to another object. The workspace used into an object to manipulate a pointer could be slightly larger than the pointer alone. Therefore, whenever a pointer has to be stored into an object, the space for a ref must be allocated instead.

3. classID is a scalar number which identifies a class. The first class returned will have classID equal to 1 and the following 2,3, etc. The classID value is guaranteed not to change across restarts.

4. permID is a scalar number used to identify persistent roots. The first object selected as permanent will have its id set to 1, and the following 2,3, etc. The permID value is guaranteed not to change across restarts.

5. classSize is a generic class size. It is guaranteed to be at least as large as an int.

[1] Biblical name, which means Brotherly or Fraternal. Being a layer whose purpose is to bring together two different software parts (the language compiler and the object store), the name seemed appropriate.

A.1.2. Functions

The interface is composed by the following functions:

- `int OpenAhio(int restart,void *arg);`

  Used to initialise the layer.

`restart`	must be false (zero) if a new store has to be created. if the value is true (different from zero) an attempt is made to recover an existing store.
arg	is a store-dependant parameter which is passed unaltered to the underlying store.

- `void CloseAhio();`

  Used to close the store and flush all the buffers.

- `classID NewClass(classSize numRefs,classSize numBytes);`

  Creation of a new class. Returns the identifier to the newly created class.

numRefs	is the number of references to other objects desired for instances of that class
numBytes	is the number of bytes of data (non-pointers) desired

- `ptr NewObj(classID cl);`

  Creates a new object. Returns the pointer to the new object.

cl	is the classID of the class to which the new object must be an instance

- `permID MakePermanent(ptr p);`

  An existing, already allocated object is made permanent, and works as a persistent root. Returns the identifier to the new root.

p	is the object which has to be made permanent

- `void Restart();`

  All the buffers are flushed and the store is stabilised. All objects present in memory are discarded and the system is reinitialised.

- `void Checkpoint();`

  Performs an explicit checkpoint. When the function returns, the store is stable.

Nothing else is required to use the layer. The interface is intentionally essential and extremely simple to use. The new objects created are currently composed by numRefs references at the beginning of the objects, followed by numBytes freely usable bytes of scalar values. All pointers stored into objects must currently refer to the starting locations of other objects.

A.1.3. Errors

It is possible to intercept the exceptions generated by the layer. The functionality can be used to bypass the default behaviour, which consists in printing an error message and exiting. An user defined error handler must have the prototype:

```
void myHandler(errorDesc e);
```

The prototype is define with the type "`crashHandler`". The new handler can be installed using:

```
crashHandler InstallCrashHandler(crashHandler newHandler);
```

which returns the address of the handler previously active. The personalised crash handler can obtain a textual description of the error with the function:

```
char *GetErrMsg(errorDesc e);
```

If the error "`e`" equals the constant `ERR_HostResourcesExhausted`, it is safe to call `Restart()` and resume the execution.

A.1.4. Customisation

In the current implementation, it is possible to change some default values by defining the following symbols while recompiling:

MAX_PHYSICAL	max physical memory, in kB
MAX_CLASSES	max number of classes
MAX_RESIDENT	max number of objects in memory
MAX_PERMANENT	max number of persistent roots
MAX_FIRST_WRITE	an automatic checkpoint if performed when this amount of new objects have been created
MAX_MAPSIZE	max number of objects mapped in virtual memory

A.2. Test Programs

A.2.1. File: BOH/test/test.first

```
//
// test.first - Antonio Cunei
//
first_example :  uses system
{
 !main()
 {
  println("Hello, world!");
 }
}
```

Output:

```
...  execution begins at PC: 0
Hello, world!
```

A.2.2. File: BOH/test/test.second

```
//
// test.second - Antonio Cunei
//
second_package:  uses system
{
//
// first class definition:  glass
//

 !glass :  super object
 {
  !glass():  super object()
  {
  }

  !break(a:glass)
  {
   println("Crash!");
  }
 }
//----------
//
// second class definition:  mattress
//

 !mattress :  super object
 {
  springs:long;

  !mattress(n:long):  super object()
  {
   mattress.springs:=n;
  }

  !break(m:mattress)
  {
   for a:=0; a<m.springs; a:=a+1;
   {
    println("Sproingg!!");
   }
  }
 }
```

```
 !main()
 {
  a:=glass();
  b:=mattress(3);
  break(a);
  break(b);
 }
}
```

Output:

```
...  execution begins at PC: 0
Crash!
Sproingg!!
Sproingg!!
Sproingg!!
```

A.2.3. File: BOH/test/test.ONE

```
//
// test.ONE - Antonio Cunei
//
basic_system :  uses system {

/*
One comment /* nested?  */
// on a line is fine as well
*/
// outside the block

!  mytestKclass :  super object {

 !  zipp()
 {
  "zipp!".println;
 }

 !  main()
 {
  nl();
  print("  The length of the string ""Machiavelli"" is:  ");
  println(len("Machiavelli"));
  print("  While 4+4*(5+3)-7/2 is ");
  println(4+4*(5+3)-7/2);
  println("  See how smart I am?");
  println(2(4));
  /*
  println(5e3(4));
  println(51df(4));
  println(51f-(6)(4));
  */
  nl();
  println("  Let's try something more difficult, now:");
  "  The value of (2*3+7.-(4)).*(4+3.*(5)) is :  ".print;
  ((2*3+7.-(4)).*(4+3.*(5))).println;
  .nl;
  "-------------".println;
  .nl;
  "Interaction test!".println;
```

```
.nl;
" Type something!  :  ".print;
" You typed :  """.+(.readText).+(""""").println;
.nl;
"-------------".println;
"  Examples of different syntax styles:".println;
"    5.+(6).+(7).+(8).*(2).println  :  ".print;
     5.+(6).+(7).+(8).*(2).println;
"    println(*(+(+(+(5,6),7),8),2)) :  ".print;
     println(*(+(+(+(5,6),7),8),2));
"    println((5+6+7+8)*2)           :  ".print;
     println((5+6+7+8)*2);
.nl;
"-------------".println;
" Let's if I can print special characters!  :  ""%\""".println;
.nl;
" And now some assorted numbers :  ".print;
0.print; " ".print;
1.print; " ".print;
1.-.print; " ".print;
2000000000ul.print; " ".print;
4000000000ul.print; " ".print;
2147483647.print; " ".print;
2147483648.-.println;
true.and(false).println;
false.and(false).println;
.nl;
true.or(false).println;
false.or(false).println;
.nl;

.zipp;
"Done!".println;
.nl;

/*
{
 b:=34+8*4-2;
 c:=4;
 a:=b+c;
 a.println;
 .nl;
}
*/
```

```
  if (4<5) {
  .nl;
  puf:=3;
  } elsif (6>3) {
  .nl;
  puf:=65;
  puf.println;
  } elsif (7<>9) {
  .nl;
  } else {
  .nl;
  }



  {
   c:=0;

   while c<10 {
    c.print;
    " ".print;
    c:=c+1;
   }
   .nl; .nl;
  }

  for j:=0; j<10; j:=j+1; {
   j.print;
   " :  ".print;
   for i:=0; i<10; i:=i+1; {
    i.print; " ".print;
   }
   .nl;
  }

  c:="The demonstration is really terminated, now.";
  c.println;
 }
}
}
```

Output:

```
...  execution begins at PC: 0
```

```
  The length of the string "Machiavelli" is:  11
  While 4+4*(5+3)-7/2 is 33
  See how smart I am?
4.2

  Let’s try something more difficult, now:
  The value of (2*3+7.-(4)).*(4+3.*(5)) is :  171

-------------

Interaction test!

 Type something!  :  67w ei9w6 84w8o
 You typed :  "67w ei9w6 84w8o"

-------------
  Examples of different syntax styles:
    5.+(6).+(7).+(8).*(2).println  :  52
    println(*(+(+(+(5,6),7),8),2)) :  52
    println((5+6+7+8)*2)           :  52

-------------
 Let’s if I can print special characters!  :  "%\"

 And now some assorted numbers :  0 1 -1 2000000000 4000000000 2147483647 -2147483648
false
false

true
false

zipp!
Done!



0 1 2 3 4 5 6 7 8 9

0 :  0 1 2 3 4 5 6 7 8 9
1 :  0 1 2 3 4 5 6 7 8 9
2 :  0 1 2 3 4 5 6 7 8 9
3 :  0 1 2 3 4 5 6 7 8 9
```

```
4 :  0 1 2 3 4 5 6 7 8 9
5 :  0 1 2 3 4 5 6 7 8 9
6 :  0 1 2 3 4 5 6 7 8 9
7 :  0 1 2 3 4 5 6 7 8 9
8 :  0 1 2 3 4 5 6 7 8 9
9 :  0 1 2 3 4 5 6 7 8 9
The demonstration is really terminated, now.
```

A.2.4. File: BOH/test/test.TWO

```
//
// test.TWO - Antonio Cunei
// Checks numeric syntax styles
//
basic_system :  uses {

!  mytestKclass :  super object {
 !  main()
 {
println(2(4));
println(5e3(4));
println(51f6(4));
51f-(6)(4).println;

println(4.2);
println(4.5e3);
println(4.51f6);
println(4.51f-(6));

println(4.2);
println(4.5e3);
println(4.51f6);
println(4.51f-6);

println(18);
println(-18);
println(18w);
println(-18w);


999e.println;
999e99.println;
999.9.println;
999.9e.println;
999e-99.println;
999.9e99.println;
999.9e-99.println;
999.9e-(99).println;
9e-99(999).println;
9(999).println;
9e(999).println;
999e-(99).println;
```

```
9e99(999).println;
9e-(99)(999).println;

println(-999e);
println(-999e99);
println(-999.9);
println(-999.9e);
println(-999e-99);
println(-999.9e99);
println(-999.9e-99);
println(-999.9e-(99));
println(-9e-99(999));
println(-9(999));
println(-9e(999));
println(-999e-(99));
println(-9e99(999));
println(-9e-(99)(999));

-999e.println;
-999e99.println;
-999.9.println;
-999.9e.println;
-999e-99.println;
-999.9e99.println;
-999.9e-99.println;
-999.9e-(99).println;
-9e-99(999).println;
-9(999).println;
-9e(999).println;
-999e-(99).println;
-9e99(999).println;
-9e-(99)(999).println;

999e.-.println;
999e99.-.println;
999.9.-.println;
999.9e.-.println;
999e-99.-.println;
999.9e99.-.println;
999.9e-99.-.println;
999.9e-(99).-.println;
9e-99(999).-.println;
9(999).-.println;
9e(999).-.println;
999e-(99).-.println;
```

```
9e99(999).-.println;
9e-(99)(999).-.println;

12873e.println;
3e11.println;
971.1.println;
119.1878237683463768732e.println;
7623f-2.println;
567.4e12.println;
0.98765f-99.println;
712.9e-(18).println;
0e-001(91872133).println;
563(98675).println;
652e(1672518).println;
98675e-(3).println;
788e000013(2).println;
11112e-(5)(8761).println;

a:=2e;

for c:=0; c<10; c:=c+1; {
 a.println;
 a:=a*a;
}

 }
}
}
```

Output:

```
...  execution begins at PC: 0
4.2
4500
4510000
4.51e-06
4.2
4500
4510000
4.51e-06
4.2
4500
4510000
4.51e-06
```

```
18
-18
18
-18
999
9.99e+101
999.9
999.9
9.99e-97
9.999e+101
9.999e-97
9.999e-97
9.999e-97
999.9
999.9
9.99e-97
9.999e+101
9.999e-97
-999
-9.99e+101
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
-9.999e-97
-9.999e-97
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
-999
-9.99e+101
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
-9.999e-97
-9.999e-97
-999.9
-999.9
-9.99e-97
-9.999e+101
```

```
-9.999e-97
-999
-9.99e+101
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
-9.999e-97
-9.999e-97
-999.9
-999.9
-9.99e-97
-9.999e+101
-9.999e-97
12873
3e+11
971.1
119.1878
76.23
5.674e+14
0
7.129e-16
9187213
98675.56
1672519
98.675
2.788e+13
0.08761111
2
4
16
256
65536
4.294967e+09
1.844674e+19
3.402824e+38
1.157921e+77
1.340781e+154
```

A.2.5. File: BOH/test/test.THREE

```
//
// test.THREE - Antonio Cunei
//
basic_system :  uses {

operator x +j x 710;
operator x -j x 710;

//---------

!  complex :  super num
{
 re,im:double;

 !  +j(r,i:double):  super numZero()
 {
  +j.re:= r;
  +j.im:= i;
 }

 !  println(c:complex)
 {
  c.re.print;
  if c.im<0.0 {
   "-j".print;   c.im.-.println;
  } else {
   "+j".print;   c.im.println;
  }
 }
}

//---------

!  list :  super object
{
 !  emptyList():  super object() {}
 !  scanPrint(l:list) {}
}

!  nonulllist :  super list
{
 it:object;
```

```
 next:list;

 !  add(l:list,x:object):  super emptylist()
 {
  add.next:=l;
  add.it:=x;
 }

 !  scanPrint(l:nonulllist)
 {
  println(l.it);
  scanPrint(l.next);
 }
}

//---------
// just as a test:  subclass of complex

!  plaf :  super complex
{
 vec:  long[-8..-3];

 !  printme(x:plaf)
 {
  for a:=-8; a<-3; a:=a+1; {
   x.vec[a].println;
  }
 }

 !  fillme() :  super 2.4+j5.1
 {
  for a:=-8; a<-3; a:=a+1; {
   fillme.vec[a]:=(10-a)*8;
  }
 }
}

//---------

!  main()
{

 a:=2e;

 for c:=0; c<10; {a:=a*a;c:=c+1;} {
```

```
  a.println;
 }

 .nl;

 5.0+j3.4.println;

 .nl;
//

 "Now a list of assorted elements:".println;
 .nl;
 .emptyList.add(4).add(1.2+j4.7).add("hello").add(912.45*7e91).scanPrint;
 .nl;
//

 "Finally, a nice array (subclass of complex!?)  filled and immediately printed:"
 .println;
 .nl;
 fillme().printme;
 "--".println;
 fillme().println;
}
}
```

Output:

```
...  execution begins at PC: 0
2
4
16
256
65536
4.294967e+09
1.844674e+19
3.402824e+38
1.157921e+77
1.340781e+154

5+j3.4

Now a list of assorted elements:

6.38715e+94
```

```
hello
1.2+j4.7
4

Finally, a nice array (subclass of complex!?)  filled and immediately printed:

144
136
128
120
112
--
2.4+j5.1
```

A.2.6. File: BOH/test/test.complex

```
//
// test.complex - Antonio Cunei
//
basic_system :  uses system {

operator x +j x  710;
operator x -j x  710;

//---------

!  complex :  super num
{
 re,im:double;

 !  -j(r,i:double):  super numZero()
 {
  -j.re := r;
  -j.im := -i;
 }

 !  +j(r,i:double):  super numZero()
 {
  +j.re:= r;
  +j.im:= i;
 }

 !  +(a,b:complex):  super numZero()
 {
  +.re:=a.re+b.re;
  +.im:=a.im+b.im;
 }

 !  -(a:complex):  complex {
  -:=(-(a.re))-j(a.im);
 }

 !  *(a,b:complex):  complex {
  *:=(a.re*b.re-a.im*b.im)+j(a.im*b.re+a.re*b.im);
 }

 !  complex(d:double):  complex {
  complex:=d+j0e;
```

```
 }

 !  /(a,b:complex):  complex {
  d:=b.re*b.re+b.im*b.im;
  if d=0.0 {
   "Division by zero".println;
   /:=0.0+j0.0;
  } else {
   /:=((a.re*b.re+a.im*b.im)/d)+j((a.im*b.re-a.re*b.im)/d);
  }
 }

 !  =(a,b:complex):  bool {
  if (a.re=b.re and a.im=b.im) {
   =:=true;
  } else {
   =:=false;
  }
 }

 !  println(c:complex)
 {
  c.re.print;
  if c.im<0.0 {
   "-j".print; -(c.im).println;
  } else {
   "+j".print;   c.im.println;
  }
 }

 !complexTest()
 {
  "The real      part of 5.0+j3.4 is :  ".print;  5.0+j3.4 .re .println;
  "The imaginary part of 5.0+j3.4 is :  ".print;  5.0+j3.4 .im .println;
 }
}

//---------

!  mytest_class :  super object
{
 !  main()
 {
  a:=4.0+j5.0;
  " a = ".print;
```

```
    a   .println; .nl;
  b:=3.91-j4.2;
  " b = ".print;
    b   .println; .nl;
  c:=a*b;
  " c = a*b = ".print;
    c         .println;
  .nl;
  "c+b/a + 1.0-j9.3 = ".print;
  (c+b/a + 1.0-j9.3)   .println;
  .nl;
  "(b/c).+(a) + 3.2-j4.3 = ".print;
  ((b/c).+(a) + 3.2-j4.3)  .println;
  .nl;
  "b.+(a/c) = ".print;
  (b.+(a/c))   .println;
  .nl;
  "b.+(a/c) + 1.0-j9.3 = ".print;
  (b.+(a/c) + 1.0-j9.3)   .println;
  .nl;
  "12.3+j4.7 * 3.2-j8.25 = ".print;
  (12.3+j4.7 * 3.2-j8.25)   .println;

  .nl;
  .complexTest;
  .nl;
 }
}

}
```

Output:

```
...  execution begins at PC: 0
 a = 4+j5

 b = 3.91-j4.2

 c = a*b = 36.64+j2.75

c+b/a + 1.0-j9.3 = 37.50927-j7.436585

(b/c).+(a) + 3.2-j4.3 = 7.297561+j0.5780488
```

```
b.+(a/c) = 4.028744-j4.072449

b.+(a/c) + 1.0-j9.3 = 5.028744-j13.37245

12.3+j4.7 * 3.2-j8.25 = 78.135-j86.435

The real      part of 5.0+j3.4 is :  5
The imaginary part of 5.0+j3.4 is :  3.4
```

A.2.7. File: BOH/test/test.set

```
//
// test.set - Antonio Cunei
// Generic Set.
//
set :  uses system
{
 !set:  super object
 {
  first:object;
  extra:set;

  !+(a,b:set):set {
   while not(empty(a)) {
    b:=b+a.first;
    a:=a.extra;
   }
   +:=b;
  }

  !in(a:set,obj:object):bool
  {
   in:=false;
   while not(in) and not(empty(a)) {
    if a.first=obj {
     in:=true;
    }
    a:=a.extra;
   }
  }

  !+(a:set,obj:object):set
  {
   if in(a,obj) {
    +:=a;
   } else {
    c:=set(obj);
    c.extra:=a;
    +:=c;
   }
  }

  !-(a,b:set):set
```

```
{
 while not(empty(b)) {
  a:=a-b.first;
  b:=b.extra;
 }
 -:=a;
}

!-(a:set,obj:object):set
{
 b:=emptySet();
 while not(empty(a)) {
  if a.first=obj {
   b:=b+a.extra;
   a:=emptySet();
  } else {
   b:=b+a.first;
   a:=a.extra;
  }
 }
 -:=b;
}

!pick(a:set):object
{
 pick:=a.first;
}

!emptySet():  super object()
{
 emptySet.extra:=emptySet;
 emptySet.first:=object(); // ignored
}

!set(obj:object):  super object()
{
 set.first:=obj;
 set.extra:=emptySet();
}

!empty(a:set):bool
{
 empty:=(a=a.extra);
}
```

```
}

main()
{
 s:=.emptySet;

 s:=s+4+5+"hello"+4.5;
 c:=.emptySet+13+"cheese"+4+99+"hello"+4.5+4.6;
 d:=s-4-"hello";
 s:=s+c;
 s:=s-d;
 s:=s+919-99;
 "And the final content of the set is:".println;
 while not(empty(s)) {
  x:=pick(s);
  x.println;
  s:=s-x;
 }
}

}
```

Output:

```
...  execution begins at PC: 0
And the final content of the set is:
919
4.6
hello
4
cheese
13
```

A.2.8. File: BOH/test/test.rule

```
//
// test.rule - Antonio Cunei
//
rule :  uses system {

//---------

!  cmdr :  super object
{
 !newCmdr():  super numZero() {}
 !ops(a,b:cmdr):  super numZero() {}
}

!  taco:  super cmdr {
 x:long;

 !  ops(a,b:taco):taco {
  ops:=fit();
 }

 !  fit():  super newCmdr() {
  x:=19;
 }
}

!  mytest_class :  super object
{
 !  main(){}
}

}
```

During the compilation:

```
Phase 3 successfully completed.
Class hierarchy and method headers checked.

Phase4:  processing...
The method ops(taco,taco) returns a value of type taco;
which is incompatible with the constructor:
ops(cmdr,cmdr) which returns a value of type cmdr:
A method and a constructor cannot have parameters
```

```
one subclasses of the other.
```

A.2.9. File: BOH/test/test.tunnel

```
//
// test.tunnel - Antonio Cunei
//
tunnel:  uses system {

!set:  super object {
 !set():  super object() {}
 !inner(x:set,a:set):a
 {
  step:=a;
  inner:=step;
 }
 !gruppolo(a:set,b:a,xx:set,yy:xx):a
 {
  epsilon:=b;
  gruppolo:=inner(xx,epsilon);
 }
}

!gluz:  super set {
 !gluz():  super set() {}
 !special(c:gluz) {}
}

!main()
{
 .gluz.special;

 s:=.set;
 g:=.gluz;

// this invocation can succeed
// only if the static type
// return is the most specific one.

 gruppolo(g,g,s,s).special;

 "Success!".println;
}

}
```

Output:

```
...  execution begins at PC: 0
Success!
```

A.2.10. File: BOH/test/test.recovery

```
basic_system :  uses system {
!  mytest_class :  super object {
 !  main()
 {

  total:=0;

  for j:=0; j<100000; j:=j+1; {
   j.print;
   " :  ".print;
   for i:=0; i<10; i:=i+1; {
    i.print; " ".print;
   }
   total:=total+j;
   ":  The sum is now :".print;
   total.println;
  }

 }
}
}
```

Output:

```
...  execution begins at PC: 0
0 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :0
1 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1
2 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :3
3 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :6
4 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :10
5 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :15
6 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :21
7 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :28
8 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :36
9 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :45
10 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :55
11 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :66
12 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :78
13 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :91
14 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :105
15 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :120
16 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :136
```

```
17 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :153
18 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :171
19 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :190
20 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :210
^C21 :  0 1 2 3 4 [cuneia@neptune test]$ ./result 1
## Sphere Recovery ::  Detected Inconsistent Store
## Sphere Recovery ::  Redoing lost updates ...
## Sphere Recovery ::  Undoing loser histories ...
## Sphere Recovery ::  Store OK
...  execution resumes at PC: 109
:  The sum is now :210
21 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :231
22 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :253
23 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :276
24 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :300
25 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :325
26 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :351
27 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :378
28 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :406
29 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :435
30 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :465
31 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :496
32 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :528
33 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :561
34 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :595
35 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :630
36 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :666
37 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :703
38 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :741
39 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :780
40 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :820
41 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :861
42 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :903
43 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :946
44 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :990
45 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1035
46 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1081
47 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1128
48 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1176
49 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1225
50 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1275
51 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1326
52 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1378
53 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1431
54 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1485
```

```
55 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1540
^C56 :  0 1 2 3[cuneia@neptune test]$ ./result 1
## Sphere Recovery ::  Detected Inconsistent Store
## Sphere Recovery ::  Redoing lost updates ...
## Sphere Recovery ::  Undoing loser histories ...
## Sphere Recovery ::  Store OK
...  execution resumes at PC: 99
9 :  The sum is now :861
42 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :903
43 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :946
44 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :990
45 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1035
46 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1081
47 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1128
48 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1176
49 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1225
50 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1275
51 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1326
52 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1378
53 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1431
54 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1485
55 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1540
56 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1596
57 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1653
58 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1711
59 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1770
60 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1830
61 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1891
^C62 :  0 1 2 3 4 5 6 7[cuneia@neptune test]$ ./result 1
## Sphere Recovery ::  Detected Inconsistent Store
## Sphere Recovery ::  Redoing lost updates ...
## Sphere Recovery ::  Undoing loser histories ...
## Sphere Recovery ::  Store OK
...  execution resumes at PC: 99
9 :  The sum is now :861
42 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :903
43 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :946
44 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :990
45 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1035
46 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1081
47 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1128
48 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1176
49 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1225
50 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1275
51 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1326
```

```
52 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1378
53 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1431
54 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1485
55 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1540
56 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1596
57 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1653
58 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1711
59 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1770
60 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1830
61 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1891
62 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :1953
63 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2016
64 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2080
65 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2145
66 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2211
67 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2278
68 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2346
69 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2415
70 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2485
71 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2556
72 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2628
73 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2701
74 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2775
75 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2850
76 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2926
77 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :3003
78 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :3081
79 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :3160
^C80 :  0 1 2 3 4 [cuneia@neptune test]$ ./result 1
## Sphere Recovery ::  Detected Inconsistent Store
## Sphere Recovery ::  Redoing lost updates ...
## Sphere Recovery ::  Undoing loser histories ...
## Sphere Recovery ::  Store OK
...  execution resumes at PC: 269
 8 9 :  The sum is now :1953
63 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2016
64 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2080
65 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2145
66 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2211
67 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2278
68 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2346
69 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2415
70 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2485
71 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2556
72 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2628
```

```
73 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2701
74 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2775
75 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2850
76 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :2926
77 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :3003
78 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :3081
79 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :3160
80 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :3240
81 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :3321
82 :  0 1 2 3 4 5 6 7 8 9 :  The sum is now :3403
^C83 :  0 1 2 3 [cuneia@neptune test]$
```

A.2.11. Compiled Code

Sample program:

```
basic_system :  uses system {
!  mytest_class :  super object {
 !  main()
 {

  total:=0;

  for j:=0; j<100000; j:=j+1; {
   j.print;
   " :  ".print;
   for i:=0; i<10; i:=i+1; {
    i.print; " ".print;
   }
   total:=total+j;
   ":  The sum is now :".print;
   total.println;
  }

 }
}
}
```

Resulting code:

```
//
// no return value
//
struct _f_main {
_d_long *tmp0;
_d_long *total_2;
_d_long *tmp1;
_d_long *j_2;
_d_long *tmp3;
_d_bool *tmp4;
_d_long *tmp5;
_d_text *tmp6;
_d_long *tmp7;
_d_long *i_3;
_d_long *tmp9;
_d_bool *tmp10;
_d_long *tmp11;
_d_text *tmp12;
```

```
_d_text *tmp13;
};
_m_main:
{
_f_main *f;

allocFrame((sizeof(struct _f_main)/sizeof(ptr))-0);
//-------------------------------------------------------------
pc244:  *pcPtr=244;
                                                                //  tmp0:=0  // long
((_f_main*)topRf)->tmp0=new_long(0x00000000);
//-------------------------------------------------------------
pc245:  *pcPtr=245;
                                                                //  total_2:=tmp0  // long
((_f_main*)topRf)->total_2=(_d_long*)((_f_main*)topRf)->tmp0;
//-------------------------------------------------------------
pc246:  *pcPtr=246;
                                                                //  tmp1:=0  // long
((_f_main*)topRf)->tmp1=new_long(0x00000000);
//-------------------------------------------------------------
pc247:  *pcPtr=247;
                                                                //  j_2:=tmp1  // long
((_f_main*)topRf)->j_2=(_d_long*)((_f_main*)topRf)->tmp1;
                                                                //  FOR LOOP :
toptmp2:
//-------------------------------------------------------------
pc248:  *pcPtr=248;
                                                                //  tmp3:=100000  // long
((_f_main*)topRf)->tmp3=new_long(0x000186a0);
//-------------------------------------------------------------
pc249:  *pcPtr=249;
                                                                //  tmp4:=H(j_2,tmp3)  // bool
  f=(_f_main*)topRf;
  theLabel=dispatchMTCmessage("H",f->j_2,f->tmp3);
  allocFrame(1);
  pushArg((ptr)(f->tmp3));
  pushArg((ptr)(f->j_2));
  pushPC(250);
  goto *theLabel;
  pc250:  *pcPtr=250;
  ret=popArg();
  ((_f_main*)topRf)->tmp4=(_d_bool*)ret; // Return Value
                                                                //  FOR tmp4
if (!get_bool(((_f_main*)topRf)->tmp4)) goto endtmp2;
goto fortmp2;
looptmp2:
//-------------------------------------------------------------
pc251:  *pcPtr=251;
                                                                //  tmp5:=1  // long
((_f_main*)topRf)->tmp5=new_long(0x00000001);
//-------------------------------------------------------------
pc252:  *pcPtr=252;
```

```
                                                                //  j_2:=I(j_2,tmp5)  // long
  f=(_f_main*)topRf;
  theLabel=dispatchMTCmessage("I",f->j_2,f->tmp5);
  allocFrame(1);
  pushArg((ptr)(f->tmp5));
  pushArg((ptr)(f->j_2));
  pushPC(253);
  goto *theLabel;
  pc253:  *pcPtr=253;
  ret=popArg();
  ((_f_main*)topRf)->j_2=(_d_long*)ret; // Return Value
goto toptmp2;
fortmp2:
//-------------------------------------------------------------
pc254:  *pcPtr=254;
                                                                //  print(j_2)
  f=(_f_main*)topRf;
  theLabel=dispatchMTCmessage("print",f->j_2);
  pushArg((ptr)(f->j_2));
  pushPC(255);
  goto *theLabel;
  pc255:  *pcPtr=255;
//-------------------------------------------------------------
pc256:  *pcPtr=256;
                                                                //  tmp6:=" :  "  // text
((_f_main*)topRf)->tmp6=new_text(" :  ");
//-------------------------------------------------------------
pc257:  *pcPtr=257;
                                                                //  print(tmp6)
  f=(_f_main*)topRf;
  theLabel=dispatchMTCmessage("print",f->tmp6);
  pushArg((ptr)(f->tmp6));
  pushPC(258);
  goto *theLabel;
  pc258:  *pcPtr=258;
//-------------------------------------------------------------
pc259:  *pcPtr=259;
                                                                //  tmp7:=0  // long
((_f_main*)topRf)->tmp7=new_long(0x00000000);
//-------------------------------------------------------------
pc260:  *pcPtr=260;
                                                                //  i_3:=tmp7  // long
((_f_main*)topRf)->i_3=(_d_long*)((_f_main*)topRf)->tmp7;
                                                                //  FOR LOOP :
toptmp8:
//-------------------------------------------------------------
pc261:  *pcPtr=261;
                                                                //  tmp9:=10  // long
((_f_main*)topRf)->tmp9=new_long(0x0000000a);
//-------------------------------------------------------------
pc262:  *pcPtr=262;
                                                                //  tmp10:=H(i_3,tmp9)  // bool
```

```
  f=(_f_main*)topRf;
  theLabel=dispatchMTCmessage("H",f->i_3,f->tmp9);
  allocFrame(1);
  pushArg((ptr)(f->tmp9));
  pushArg((ptr)(f->i_3));
  pushPC(263);
  goto *theLabel;
  pc263:  *pcPtr=263;
  ret=popArg();
  ((_f_main*)topRf)->tmp10=(_d_bool*)ret; // Return Value
                                                                  //  FOR tmp10
if (!get_bool(((_f_main*)topRf)->tmp10)) goto endtmp8;
goto fortmp8;
looptmp8:
//-------------------------------------------------------------
pc264:  *pcPtr=264;
                                                                  //  tmp11:=1  // long
((_f_main*)topRf)->tmp11=new_long(0x00000001);
//-------------------------------------------------------------
pc265:  *pcPtr=265;
                                                                  //  i_3:=I(i_3,tmp11)  // long
  f=(_f_main*)topRf;
  theLabel=dispatchMTCmessage("I",f->i_3,f->tmp11);
  allocFrame(1);
  pushArg((ptr)(f->tmp11));
  pushArg((ptr)(f->i_3));
  pushPC(266);
  goto *theLabel;
  pc266:  *pcPtr=266;
  ret=popArg();
  ((_f_main*)topRf)->i_3=(_d_long*)ret; // Return Value
goto toptmp8;
fortmp8:
//-------------------------------------------------------------
pc267:  *pcPtr=267;
                                                                  //  print(i_3)
  f=(_f_main*)topRf;
  theLabel=dispatchMTCmessage("print",f->i_3);
  pushArg((ptr)(f->i_3));
  pushPC(268);
  goto *theLabel;
  pc268:  *pcPtr=268;
//-------------------------------------------------------------
pc269:  *pcPtr=269;
                                                                  //  tmp12:=" "  // text
((_f_main*)topRf)->tmp12=new_text(" ");
//-------------------------------------------------------------
pc270:  *pcPtr=270;
                                                                  //  print(tmp12)
  f=(_f_main*)topRf;
  theLabel=dispatchMTCmessage("print",f->tmp12);
  pushArg((ptr)(f->tmp12));
```

```
      pushPC(271);
      goto *theLabel;
      pc271:  *pcPtr=271;
    goto looptmp8;
                                                                  //  ENDFOR
    endtmp8:
    //-------------------------------------------------------------
    pc272:  *pcPtr=272;
                                                                  //  total_2:=I(total_2,j_2)  // long
      f=(_f_main*)topRf;
      theLabel=dispatchMTCmessage("I",f->total_2,f->j_2);
      allocFrame(1);
      pushArg((ptr)(f->j_2));
      pushArg((ptr)(f->total_2));
      pushPC(273);
      goto *theLabel;
      pc273:  *pcPtr=273;
      ret=popArg();
      ((_f_main*)topRf)->total_2=(_d_long*)ret; // Return Value
    //-------------------------------------------------------------
    pc274:  *pcPtr=274;
                                                                  //  tmp13:=":  The sum is now :"  //
text
    ((_f_main*)topRf)->tmp13=new_text(":  The sum is now :");
    //-------------------------------------------------------------
    pc275:  *pcPtr=275;
                                                                  //  print(tmp13)
      f=(_f_main*)topRf;
      theLabel=dispatchMTCmessage("print",f->tmp13);
      pushArg((ptr)(f->tmp13));
      pushPC(276);
      goto *theLabel;
      pc276:  *pcPtr=276;
    //-------------------------------------------------------------
    pc277:  *pcPtr=277;
                                                                  //  println(total_2)
      f=(_f_main*)topRf;
      theLabel=dispatchMTCmessage("println",f->total_2);
      pushArg((ptr)(f->total_2));
      pushPC(278);
      goto *theLabel;
      pc278:  *pcPtr=278;
    goto looptmp2;
                                                                  //  ENDFOR
    endtmp2:
    //-------------------------------------------------------------
    pc279:  *pcPtr=279;
                                                                  //  nothing to return.
    releaseFrame((uint32)(sizeof(struct _f_main)/sizeof(ptr)));
    goto *jumpTable[popPC()];

    }
```

A.3. Language comparison

A.3.1. SideEffect in Smalltalk

```
"
""
""  Side effect in Smalltalk.
""  The procedure "dangerous" discovers that one of its
""  parameters changes unexpectedly.
""
"

Object  subclass:  #sideEffect
        instanceVariableNames:  'n'
        classVariableNames:  ''
        poolDictionaries:  ''
        category:  nil !

!sideEffect class methodsFor:  ''!
new:  v
 ^((super new) init:  v)
!!

!sideEffect methodsFor:  ''!
init:  v
 n:=v
!
val
 ^n
!
dangerous:  b
 'b is ' xprint.
 (b val) printNl.

 n:=n+20.

 ((b val) > 20) ifTrue:  [
   'Ehi!  b has changed!  Now it is:  '  xprint.
   (b val) printNl
 ]
!!
"
"" A custom printing function
"
```

```
!String methodsFor:  'customPrinting'!
xprint
 self do:  [ :char | stdout nextPut:  char ]
!!

"
"" Verification of the side-effect!
"
 |a|
  a:=sideEffect new:  5.
  a dangerous:  a
!
```

Output:

```
Execution begins...
b is 5
Ehi!  b has changed!  Now it is:  25
2587 byte codes executed
```

A.3.2. SideEffect in Java

```
/*
  Side effect in Java.
  The procedure "dangerous" discovers that one of its
  parameters changes unexpectedly.
*/

class sideEffect
{
 long n;

 sideEffect(long v) { n=v; }

 static void dangerous(sideEffect a,sideEffect b)
 {
  System.out.println("b is "+b.n);

  a.n+=20;

  if (b.n>10)
   System.out.println("Ehi!  b has changed!  Now it is:  "+b.n);
 }

 public static void main(String av[])
 {
  sideEffect a;
  a=new sideEffect(5);
  dangerous(a,a);
 }
}
```

Output:

```
b is 5
Ehi!  b has changed!  Now it is:  25
```

A.3.3. SideEffect in BOH

```
/*
  Side effect in BOH.
  The procedure "dangerous" discovers that one of its
  parameters changes unexpectedly.
*/
side:  uses system {

!sideEffect:super object
{
 n:long;

 !sideEffect(v:long):  super object() { sideEffect.n:=v; }

 !dangerous(a,b:sideEffect)
 {
  "b is ".print;
  b.n.println;

  a.n:=a.n+20;

  if b.n>10 {
   print("Ehi!  b has changed!  Now it is:  ");
   b.n.println;
  }
 }
}

!main()
{
 a:=sideEffect(5);
 dangerous(a,a);
}
}
```

Output:

```
...  execution begins at PC: 0
b is 5
Ehi!  b has changed!  Now it is:  25
```

A.3.4. Identifiers in BOH: extended character set

```
lexicon :  uses system
{

!  ~B%#@@$?  ():long
{
 ~B%#@@$?  := 15;
}

main()
{
 println(~B%#@@$?());
}

}
```

Output:

```
...  execution begins at PC: 0
15
```

A.3.5. Overloading vs. generic functions: Java

```
class child extends parent {}

class parent
{
 void direct(parent b) {
  System.out.println(":  Called parent - parent");
 }

 void direct(child b) {
  System.out.println(":  Called parent - child");
 }

 void indirect(parent b) {
  direct(b);
 }

 public static void main(String av[])
 {
  parent p=new parent();
  child c=new child();

  System.out.print(p); System.out.print(p); p.direct(p);
  System.out.print(p); System.out.print(c); p.direct(c);
  System.out.print(p); System.out.print(c); p.indirect(c);
 }
}
```

Output:

```
parent@80caf4a parent@80caf4a :  Called parent - parent
parent@80caf4a child@80caf4c :  Called parent - child
parent@80caf4a child@80caf4c:  Called parent - parent
```

A.3.6. Overloading vs. generic functions: BOH

```
GenericVsOverload:  uses system
{
!parent:  super object {!parent():  super object() {}}
!child:  super parent {!child():  super parent() {}}

direct(a:parent,b:parent)
{ println(" :  Called parent - parent"); }

direct(a:parent,b:child)
{ println(" :  Called parent - child"); }

indirect(a:parent,b:parent)
{ direct(a,b); }

main()
{
 p:=parent();
 c:=child();

 p.print; p.print; p.direct(p);
 p.print; c.print; p.direct(c);
 p.print; c.print; p.indirect(c);
}
}
```

Output:

```
...  execution begins at PC: 0
<parent><parent> :  Called parent - parent
<parent><child> :  Called parent - child
<parent><child> :  Called parent - child
```

A.3.7. Methods with different return types in Java

```
class child extends parent
{
 figlia msg() {
  System.out.println(" Called method in child");
  return (new child());
 }
}

class parent
{
 parent msg() {
  System.out.println(" Called method in parent");
  return (new parent());
 }

 public static void main(String av[])
 {
  parent p=new parent();
  child f=new child();
  parent k;
  k=p.msg();
  k=f.msg();
 }
}
```

Compiler Output:

```
retVal.java:2:  Method redefined with different return type:  child msg() was parent
msg()
 child msg() {
        ^
1 error
```

A.3.8. Methods with different return types in BOH

```
retVal:  uses system
{
 !parent:  super object {!parent():  super object() {}}
 !child:  super parent {!child():  super parent() {}}

msg(a:parent):parent {
 println(" Called method in parent");
 msg:=a;
}

msg(a:child):child {
 println(" Called method in child");
 msg:=a;
}

parent_or_child(a:parent) {
 print (" Parent_or_child, applied to ");
 println (a);
}

special_for_child(a:child) {
 print (" Special_for_child, applied to ");
 println (a);
}

main()
{
 p:=parent();
 f:=child();

 p.print;
 p.msg.parent_or_child;      // static type of p.msg:  parent
 .nl;
 f.print;
 f.msg.special_for_child; // static type of f.msg:  child
}
}
```

Output:

```
<parent> Called method in parent
 Parent_or_child, applied to <parent>
```

```
<child> Called method in child
 Special_for_child, applied to <child>
```

A.3.9. Ambiguity in C++

```
class ambig {};
class ambig2:ambig {};

void one(ambig a,ambig2 b) {}
void one(ambig2 a,ambig b) {}

main(){
  ambig2 a=*(new ambig2);
  one(a,a);
}
```

Compiler Output: (no suggestions, overloading)

```
ambig.cpp:  In function ‘int main()’:
ambig.cpp:9:  call of overloaded ‘one’ is ambiguous
ambig.cpp:4:  candidates are:  one(ambig, ambig2)
ambig.cpp:5:                  one(ambig2, ambig)
```

A.3.10. Ambiguity in Java

```
class ambig {}
class ambig2 extends ambig {

 static void one(ambig a,ambig2 b) {}
 static void one(ambig2 a,ambig b) {}

 static void main(){
  ambig2 a=new ambig2();
  one(a,a);
 }
}
```

Compiler Output: (no suggestions, overloading)

```
ambig.java:9:  Reference to one is ambiguous.  It is defined in
     void one(ambig2, ambig) and void one(ambig, ambig2).
  one(a,a);
     ^
1 error
```

A.3.11. Ambiguity in BOH

```
ambigPack:  uses system {

 !ambig:  super object {!ambig():  super object(){}}
 !ambig2:  super ambig {!ambig2():  super ambig(){}}

 one(a:ambig,b:ambig2) {}
 one(a:ambig2,b:ambig) {}

 main(){
  a:=ambig2();
  one(a,a);
 }
}
```

Compiler Output: (solution proposed, multiple dispatching)

```
...
Phase4:
  processing...
Uhm..  missing definition:  one(ambig2,ambig2)..
There were errors during the verification of messages.

Error during compilation.
Bailing out.
```

Bibliography

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983, `http://www-ppg.dcs.st-and.ac.uk/Publications/1983.html#approach.persist%ence`.

[ADJ+96] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *ACM SIGMOD Record*, 25(4):68–75, December 1996.

[ADL91] Rakesh Agrawal, Lindga G. DeMichiel, and Bruce G. Lindsay. Static type checking of multi-methods. *ACM SIGPLAN Notices*, 26(11):113–128, November 1991. OOPSLA '91 Conference Proceedings, Andreas Paepcke (editor), October 1991, Phoenix, Arizona.

[AH98] Alan Au and Gernot Heiser. L4 user manual. Technical Report UNSW-CSE-TR-9801, School of Computer Science and Engineering, University of New South Wales, Australia, 1998, `ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/9801.ps.Z`.

[AJ99] M. Atkinson and M. Jordan. Issues raised by three years of developing PJama: An orthogonally persistent, platform for Java[TM]. *Lecture Notes in Computer Science*, 1540:1–30, 1999.

[AM95] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995, `http://www-ppg.dcs.st-and.ac.uk/Publications/1995.html`.

[AMB95] M. Atkinson, D. Maier, and V. Benzaken, editors. *Persistent Object Systems*, Berlin, 1995.

[Ano94] Anonymous. Inheritance or delegation? *Byte Magazine*, 19(5):60, May 1994.

[App95] Apple Computer. *Dylan Reference Manual*, October 1995. (Draft).

[App96] Apple Computer. *Newton Programmer's Guide (For Newton 2.0)*. Addison-Wesley, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bal95] Stoney Ballard. *Dylan Competitive Analysis*. Apple Computer, February 1995.

[Bea94] Michel Beaudouin. *Object-Oriented Languages : Basic Principles and Programming Techniques*. Chapman & Hall, 1994.

[BFH+92] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS(R) nanokernel architecture. In USENIX Association, editor, *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures: 27–28 April, 1992, Seattle, WA, USA*, pages 95–112, Berkeley, CA, USA, April 1992. USENIX.

[BLNR96] Eva Z. Bem, Anders Linström, Stephen Norris, and John Rosenberg. Hoppix - an implementation of a unix server on a persistent operating system. In Luis-Felipe Cabrera and Nayeem Islam, editors, *Proceedings of 5th International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, pages 112–116, Washington, DC, 1996. IEEE Computer Society.

[BM97] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In ACM, editor, *Conference record of POPL '97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Paris, France, 15–17 January 1997*, pages 302–315, New York, NY, USA, 1997. ACM Press, `http://www.acm.org:80/pubs/citations/proceedings/plan/263699/p302-bourd%oncle/`.

[BMP92] Doug Bell, Ian Morrey, and John Pugh. *Software Engineering, a programming approach*. Prentice-Hall, 2nd edition, 1992.

[Boe93] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 197–206, Albuquerque, NM, June 1993. ACM Press, `http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html`.

[BOL95] Gruruduth Banavar, Douglas Orr, and Gary Lindstrom. Layered, server-based support for object-oriented application development. Technical Report UUCS-95-007, University of Utah, Department of Computer Science, April 1995.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, 1991.

[BP94] Andrew Black and Jens Palsberg. Foundations of object-oriented languages: Workshop report. *ACM SIGPLAN Notices*, 29(3):3–11, March 1994, `ftp://crl.dec.com/pub/DEC/sigplan94.ps.Z`. The bibliography was truncated in the published version. obtain the full report by anonymous ftp from crl.dec.com in pub/DEC/sigplan94.ps.Z.

[CC99] Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In Loren Meissner, editor, *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 238–255, N. Y., 1999. ACM Press, `http://www.cs.washington.edu/research/projects/cecil/www/Papers/dispatc%hing.html`.

[Cha] Jane Chandler. *Introduction to Object-Oriented Programming Languages*. Department of Information Systems, University of Portsmouth, `http://www.sis.port.ac.uk/~chandler/OOLectures/oopl/oopl.htm`. (based on [Bea94]).

[Cha92] Craig Chambers. Object-oriented multi-methods in cecil. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56, Berlin, Heidelberg, New York, Tokyo, June 1992. Springer-Verlag.

[Cha93] Craig Chambers. The Cecil language. Technical Report 93-03-05, March 1993.

[CIL93] Jeffrey S. Chase, Valérie Issarny, and Henry M. Levy. Distribution in a single address space operating system. *ACM Operating Systems Review*, 27(2):61–65, April 1993.

[CIM92] Roy H. Campbell, Nayeem Islam, and Peter Madany. Choices, frameworks and refinement. In USENIX Association, editor, *Computing Systems, Sum-*

mer, 1992., volume 5, pages 217–257, Berkeley, CA, USA, Summer 1992. USENIX.

[CIRM93] R. H. Campbell, N. Islam, D. Raila, and P. Madany. Designing and implementing choices: An object-oriented system in C++. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):117–126, 1993.

[CL95] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[CL96] Craig Chambers and Gary T. Leavens. BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing. In *The Fourth International Workshop on Foundations of Object-Oriented Languages, FOOL 4, Paris, France*, December 1996, `http://www.cs.indiana.edu/hyplan/pierce/fool/chambers.ps.gz`. The proceedings are on-line at the URL http://www.cs.williams.edu/~kim/FOOL/FOOL4.html.

[CLBHL93] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 80–85, 1993.

[Clo] *The Common Lisp Object System*, `http://iahost.dis.ulpgc.es/clos.html`. (based on [Gra96]).

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, Berlin, 1981.

[CM91] Antonio Cunei and Marino Miculan. *OOPLog, progetto per l'Esame di Linguaggi di Programmazione (Prof. Carlo Tasso)*. Università degli Studi di Udine, 1991.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACMCS*, 17(4):471–522, December 1985, `http://www.research.digital.com/SRC/personal/Luca_Cardelli/Papers/OnUnd%erstanding.A4.ps`. Model of typed, polymorphic programming languages. Existential and bounded quantification. Lambda calculus based model of type systems. The language Fun. 44 references.

[dAJK] Jecel Mattos de Assumpcao Jr and Sergio Takeo Kufuji. *Bootstrapping the Object Oriented Operating System Merlin: Just Add Reflection*, http://www.lsi.usp.br/~jecel/jpaper7.ps.gz.

[Dal97] Jeff Dalton. *Brief Guide to CLOS*. University of Edinburgh, August 1997, http://www.tnt.uni-hannover.de/data/www/soft/case/lang/lisp/clos.html.

[DAS98] Eric Dujardin, Eric Amiel, and Eric Simon. Fast algorithms for compressed multimethod dispatch table generation. *ACM Transactions on Programming Languages and Systems*, 20(1):116–165, January 1998, http://www.acm.org:80/pubs/citations/journals/toplas/1998-20-1/p116-duj%ardin/.

[dBDF+94] Rex di Bona, Alan Dearle, James Farrow, Frans Henskens, Anders Lindstrom, John Rosenberg, and Francis Vaughan. Generic interface for configurable disk i/o systems. In Gopal Gupta, editor, *Proceedings of the Seventeenth Annual Computer Science Conference, ACSC-17, Part B*, pages 355–362, Christchurch, New Zealand, January 1994. http://docs.dcs.napier.ac.uk/DOCS/GET/bona94a/document.html.

[DdBF+94a] Alan Dearle, Rex di Bona, James Farrow, Frans Kenskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, Summer 1994, http://docs.dcs.napier.ac.uk/DOCS/GET/dearle92b/document.ps.gz.

[DdBF+94b] Alan Dearle, Rex di Bona, James Farrow, Frans Kenskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, Summer 1994.

[DdBF+96] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, David Hulse, Anders Lindstrom, Stephen Norris, John Rosenberg, and Francis Vaughan. Protection in the Grasshopper operating system. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, Tarascon, France, September 1996. http://docs.dcs.napier.ac.uk/DOCS/GET/dearle94a/document.html.

[DdBL+94] Alan Dearle, Rex di Bona, Anders Lindstrom, John Rosenberg, and Francis Vaughan. User-level management of persistent data in the Grasshopper op-

erating system. Technical Report GH-08, University of Sydney, Computer Science, N.S.W 2006, Australia, 1994, `http://docs.dcs.napier.ac.uk/DOCS/GET/dearle94b/document.html`.

[Deb95] Clive Debenham. *An Introduction To TAOS*. Tantric Technologies, March 1995, `tantric@cix.compulink.co.uk`.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 156–168. ACM Press, 1995, `ftp://ftp.pmg.lcs.mit.edu/pub/thor/where-clauses.ps.gz`.

[DH95] A. Dearle and D. Hulse. On page-based optimistic process checkpointing. In *Proc. of the Fourth Int'l Workshop on Object Orientation in Operating Systems (IWOOOS'95)*, pages 24–32, August 1995.

[DRH+92] Alan Dearle, John Rosenberg, Frans Henskens, Francis Vaughan, and Kevin Maciunas. An examination of operating system support for persistent object systems. In *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences*, pages 779–789, 1992, `http://docs.dcs.napier.ac.uk/DOCS/GET/dearle92a/document.ps.gz`.

[DV66] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966. Originally presented at the Proceedings of the ACM Programming Language and Pragmatics Conference, August 8–12, 1965.

[Eck] Bruce Eckel. *Thinking in C++*, 2.0 edition, `http://www.bruceeckel.com/`. (to be published by Prentice-Hall; downloadable in RTF format).

[Edi86] Edia Borland s.r.l., V. Cirene, 11 - 20135 Milano. *Turbo Pascal - Versione 3.0*, December 1986.

[EK95] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 78–83, Orcas Island, Washington, May 1995. IEEE Computer Society, `http://www.pdos.lcs.mit.edu/papers/hotos-jeremiad.ps`.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In Eric Jul, editor, *ECOOP '98–object-oriented programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211. Springer, July 1998, `http://www.cs.washington.edu/research/projects/cecil/www/Papers/gud.htm%l`.

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, volume 29, 1995, `http://www.pdos.lcs.mit.edu/papers/exokernel-sosp95.ps`.

[EKO94] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. The exokernel approach to extensibility (panel statement). In *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation (OSDI '94)*, page 198, Monterey, California, November 1994. `http://www.pdos.lcs.mit.edu/papers/exo-abstract.ps`.

[Elp99] Kevin John Elphinstone. *Virtual memory in a 64-bit microkernel*. PhD thesis, University of New South Wales, August 1999, `ftp://ftp.cse.unsw.edu.au/pub/users/disy/papers/Elphinstone:phd.ps.gz`.

[Fla97] D. Flanagan. *Java In A Nutshell*. A Nutshell Handbook. O'Reilly, 2nd edition, 1997.

[FSB+98] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, Joâo Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Roberts, Fadi Sandakly, George Coulouris, Jean Dollimore, Paulo Guedes, Daniel Hagimont, and Sacha Krakowiak. PerDiS: design, implementation, and use of a PERsistent DIstributed Store. Technical Report QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98, QMW, CSTB, INRIA and INESC, October 1998, `http://www-sor.inria.fr/publi/PDIUPDS_rr3525.html`.

[Geh84] Narain Gehani. *Ada, an advanced introduction (including reference manual for the Ada programming language*. Prentice-Hall software series. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[GFM+91] Carlo Ghezzi, Alfonso Fuggetta, Sandro Morasca, Angelo Morzenti, and Mauro Pezzi. *Ingegneria del software: progettazione, sviluppo e verifica*. Mondadori informatica, Milano, 1991.

[GM96] A. Gawecki and F. Matthes. Integrating subtyping, matching and type quantification: A practical perspective. In Pierre Cointe, editor, *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96*, volume 1098, pages 26–47, Linz, Austria, July 1996. Springer-Verlag, `http://www.sts.tu-harburg.de/papers/1996/GaMa96b`.

[Gol84] Adele Goldberg. *Smalltalk-80: the interactive programming environment*. Addison-Wesley computer science. Addison-Wesley, Reading (Mass.), 1984.

[Gra96] Paul Graham. *ANSI Common Lisp*. Prentice Hall series in artificial intelligence. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1996.

[Gro92] Peter Grognono. *Programmare in Pascal e Turbo Pascal*. Franco Muzzio Editore, 1992.

[HB87] D. M. Harland and B. Beloff. OBJEKT: A persistent object store with an integrated garbage collector. *sigplan*, 22(4):70–79, April 1987, `http://www.saqnet.co.uk/users/beloff/computing/sigplan.html`.

[HD96] David Hulse and Alan Dearle. A log-structured persistent store. In *Proceedings of the 19th Australasian Computer Science Conference*, 1996, `http://docs.dcs.napier.ac.uk/DOCS/GET/hulse96a/document.html`.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Trans. Softw. Eng.*, SE-16(12):1344–1351, October 1990.

[HKR92] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems*, 14(4):490–520, October 1992.

[HLR98] Gernot Heiser, Fondy Lam, and Stephen Russell. Resource management in the mungi single-address-space operating system. In *Proceedings of the 21 st Australasian Computer Science Conference*, Perth, Australia, February 1998. `ftp://ftp.cse.unsw.edu.au/pub/users/disy/papers/Heiser_LR_98.ps.gz`.

[HRB+87] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The emerald programing lanuage report. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, October 1987.

[Hul96] David Hulse. A flexible persistent architecture permitting trade-off between snapshot and recovery times. Technical Report GH-16, University of Sydney, Computer Science, N.S.W 2006, Australia, 1996, `http://docs.dcs.napier.ac.uk/DOCS/GET/hulse96b/document.html`.

[Hut87] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, University of Washington, Computer Science Department, January 1987.

[HVER97] Gernot Heiser, Jerry Vochteloo, Kevin Elphinstone, and Stephen Russell. The mungi kernel api, release 1.0. Technical Report UNSW-CSE-TR-9701, University of New South Wales, Department of Computer Systems, April 1997, `ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/9701.ps.Z`.

[IBM] IBM Corporation, Object Technology Products Group, Austin, Texas. *The System Object Model (SOM) and the Component Object Model (COM): A comparison of technologies from a developer's perspective*.

[Ins] Institut für Computersysteme - ETH Zürich. *The Official Oberon Home Page*, `http://www.oberon.ethz.ch`.

[IT94] Jun-Ichiro Itoh and Mario Tokoro. Object-oriented device driver programming. In *WOOC94, session 9-2*, Biwa lake, Shiga, March 1994. `ftp://ftp.itojun.org/pub/paper/itojun-wooc94-handout.ps.Z`.

[Jon] Richard Jones. *The Garbage Collection Page*, `http://stork.ukc.ac.uk/computer_science/Html/Jones/gc.html`.

[JRH88] Eric Jul, Rajendra K. Raj, and Norman C. Hutchinson. The emerald system user's guide. Technical Report Ver. 1.3, Department of Computer Science, University of Washington, Seattle, November 1988.

[Jul88] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, University of Washington, Computer Science Department, December 1988.

[KCC+97] G.N.C. Kirby, R.C.H. Connor, Q.I. Cutts, R. Morrison, D.S. Munro, and S. Scheuerl. Flask: An architecture supporting concurrent distributed persistent applications. Technical Report CS/97/4, University of St Andrews, Scotland, 1997.

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 52–65, New York, October 5–8 1997. ACM Press, `http://www.pdos.lcs.mit.edu/papers/exo-sosp97/exo-sosp97.ps`.

[KM] G.N.C. Kirby and R. Morrison. *A Persistent View of Encapsulation*. St Andrews, Fife KY16 9SS, Scotland.

[KM97] G.N.C. Kirby and R. Morrison. Orthogonal persistence as an implementation platform for software development environments. Technical Report CS/97/6, University of St Andrews, Scotland, 1997.

[KR89] Brian W. Kernigan and Dennis M. Ritchie. *Il Linguaggio C*. Jackson, 1989.

[Kur96] Kurt Nørmark. *Hooks and Open Points*, 1996, `http://www.cs.auc.dk/~normark/hooks/hypertext/hooks.html`.

[Lan92] Charles R. Landau. The checkpoint mechanism in keykos. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE Computer Society, September 1992, `http://docs.dcs.napier.ac.uk/DOCS/GET/lindstrom95a/document.html`.

[LDdB+94] Anders Lindstrom, Alan Dearle, Rex di Bona, J. Matthew Farrow, Frans Henskens, John Rosenberg, and Francis Vaughan. A model for user-level memory management in a persistent distributed environment. In Gopal Gupta, editor, *Proceedings of the Seventeenth Annual Computer Science Conference, ACSC-17, Part B*, pages 343–354, Christchurch, New Zealand, January 1994. `http://docs.dcs.napier.ac.uk/DOCS/GET/lindstrom94a/document.html`.

[LDdB+95] Anders Lindstrom, Alan Dearle, Rex di Bona, Stephen Norris, John Rosenberg, and Francis Vaughan. Persistence in the Grasshopper kernel. In Ramamohanarao Kotagiri, editor, *Proceedings of the Eighteenth Australasian Computer Science Conference, ACSC-18*, pages 329–338, Glenelg, South Australia, February 1995. IEEE Computer Society, `http://docs.dcs.napier.ac.uk/DOCS/GET/lindstrom95a/document.html`.

[Leo99] Yuri Leontiev. *Type System for an Object-Oriented Database Programming Language*. PhD thesis, Dept. of Computing Science, University of Alberta, Edmonton, Alberta, Canada, October 1999, `ftp://ftp.cs.ualberta.ca/pub/TechReports/1999/TR99-02/`.

[LRD95] Anders Lindstrom, John Rosenberg, and Alan Dearle. The grand unified theory of address spaces. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 66–71, Orcas Island, Washington, May 1995. `http://docs.dcs.napier.ac.uk/DOCS/GET/lindstrom95b/document.html`.

[Mar93] Dave Mark. *Learn C++ on the Macintosh*. Addison-Wesley, 1993.

[Mau96] Rainer Mauth. A better foundation: Development frameworks let you build an application with reusable objects. *Byte Magazine*, 21(9), September 1996. (International Features Section).

[MBCD89] R. Morrison, A. L. Brown, R. C. H. Connor, and A. Dearle. The napier88 reference manual. Technical Report PPRR-77-89, Universities of Glasgow and St Andrews, 1989, `http://www-ppg.dcs.st-and.ac.uk/Publications/1989.html#napier.reference%.manual`.

[MC99] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 279–303, Lisbon, Portugal, June 1999. Springer-Verlag.

[McA95] Jeff McAffer. Meta-level architecture support for distributed objects. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 232–241, Lund, Sweden, November 1995. IEEE Computer Society Press.

[Met] Metrowerks Inc. *CodeWarrior Pascal: Language Reference*, `http://www.jstream.com/java11jpds/pascal/pascalbook.html`.

[MHH91] Warwick B. Mugridge, John Hamer, and John G. Hosking. Multi-methods in a statically-typed programming language. In Pierre America, editor, *ECOOP '91: European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 307–324. Springer-Verlag, 1991.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[MIT] MIT. *Exokernel Operating System*, `http://www.pdos.lcs.mit.edu/exo.html`.

[MMPN93] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, 1993.

[Mod] *Modula-3 Home Page*, `http://www.research.digital.com/SRC/modula-3/html/home.html`.

[Mot92] Motorola Inc. *M68000 Family Programmer's Reference Manual*, 1992. (Ref: M68000PM/AD).

[Mot93] Motorola Inc. *PowerPC^{TM} 601 RISC Microprocessor User's Manual*, 1993. (Ref: MPC601UM/AD).

[Mot94] Motorola Inc. *DSP96002 32-bit Digital Signal Processor User's Manual*, 1994. (Ref: DSP96002UM/AD).

[MS97] Leonid Mikhajlov and Emil Sekerinski. The fragile base class problem and its solution. Technical Report TUCS Technical Report No 117, Turku Centre for Computer Science, June 1997.

[New] NewMonics Inc. *Discussions on Real-time Garbage Collection*, `http://www.newmonics.com/webroot/technologies/gc/`.

[Obea] *Oberon V3 Pages*, `http://caesar.ics.uci.edu/oberon`.

[Obeb] *Oberon V4 Pages*, `http://www.ssw.uni-linz.ac.at/Oberon.html`.

[Obec] Oberon microsystems. *Component Software: A Case Study Using BlackBox Components*. Preliminary version.

[Orn96] David Ornstein. *Garbage Collection in Smalltalk/V*, June 1996, `http://www.parcplace.com/support/vsesupp/TIPS/note2481.htm`.

[OW99] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, 1999.

[PAD98] T. Printezis, M. P. Atkinson, and L. Daynès. The implementation of Sphere: a Scalable, Flexible, and Extensible Persistent Object Store. Technical Report TR-1998-46, Department of Computing Science, University of Glasgow, May 1998.

[PC93] Jill Nicola Peter Coad. *Object-oriented programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[Pes95a] Carlo Pescio. *C++, Manuale di Stile*. Infomedia, Pisa, 1995.

[Pes95b] Carlo Pescio. Il problema della "fragile base class" in c++. *Computer Programming*, 41, September 1995.

[PHLS99] Candy Pang, Wade Holst, Yuri Leontiev, and Duane Szaforon. Multi-method dispatch using multiple row displacement. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 304–328. Springer-Verlag, New York, NY, June 1999, `http://web.cs.ualberta.ca/~yuri/ecoop99/body.ps`.

[Pri00] T. Printezis. *Management of Long-Running High-Performance Persistent Object Stores*. PhD thesis, Department of Computing Science, University of Glasgow, Scotland, May 2000.

[Proa] *IC Prolog][, online manual*, `http://laotzu.doc.ic.ac.uk/Localinfo/icprolog.html`.

[Prob] *SICStus Prolog User's Manual*, `http://www.sics.se/isl/sicstus`.

[PS94] Dick Pountain and Clemens Szyperski. Extensible software systems: New programming tools are needed to develop software systems tha can be easily extended with new modules. *Byte Magazine*, 19(5):57, May 1994.

[PvE96] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean, Language Report (Version 1.1)*, March 1996.

[Pyl81] I.C. Pyle. *The ADA programming language: a guide for programmers*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1981.

[R31] *Introduction to KeyKOS Concepts - Online documentation*, `http://www.cis.upenn.edu/~KeyKOS/agorics/KeyKos/Concepts/welcome.html`.

[R32] *A Programmer's Introduction to EROS - Online documentation*, `http://www.eros-os.org/devel/intro/ProgrmrIntro.html`.

[R34] *The TUNES Project - Home Page*, `http://www.tunes.org`.

[RDH+96] John Rosenberg, Alan Dearle, David Hulse, Anders Lindström, and Stephen Norris. Operating system support for persistent and recoverable computations. *Communications of the ACM*, 39(9):62–69, September 1996, `http://www.acm.org/pubs/toc/Abstracts/cacm/234472.html`.

[RTL+] Rajindra K. Raj, Evan Tampero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. *Emerald: A General-Purpose Programming Language*. Seattle.

[SB86] M. Stefik and D. Bobrow. Object oriented programming: Themes and variations. *AI Magazine*, 6(4), 1986.

[Sch96] Herbert Schildt. *Guida al Linguaggio C++*. McGraw-Hill, 1996.

[SFS96] Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. State caching in the EROS kernel. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, Cape May, N.J., 1996. `http://www.eros-os.org/papers/pos96.ps`.

[Sha97] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[SKW92] K. Singhal, S. Kakkad, and P. Wilson. Texas: An efficient, portable persistent store. In *Proc. Fifth International Workshop on Persistent Object Systems*, pages 13–28, San Miniato Pisa (Italy), September 1992. `ftp://ftp.cs.utexas.edu/pub/garbage/texaspstore.ps`.

[SM98a] Alan Skousen and Donald Miller. Operating system structure and processor architecture for a large distributed single address space. In *10th IASTED Parallel and Distributed Computing Conference (PDCS98)*, pages 631–634, October 1998, `ftp://ftp.eas.asu.edu/pub/cse/sasos/pdcs98.pdf`.

[SM98b] Alan Skousen and Donald Miller. The sombrero distributed single address space operating system project. In *Proceedings of the 2nd USENIX Windows NT Symposium*, page 168, Berkeley, August 1998. USENIX Association, `ftp://ftp.eas.asu.edu/pub/cse/sasos/usenix_nt.pdf`.

[SM99a] Alan Skousen and Donald Miller. Using a distributed single address space operating system to support modern cluster computing. In *Proceedings of the 32nd Hawaii Interntional Conference on System Sciences (HICSS-32)*, January 1999, `ftp://ftp.eas.asu.edu/pub/cse/sasos/hicss-32.pdf`.

[SM99b] Alan Skousen and Donald Miller. Using a single address space operating system for distributed computing and high performance. In *18th IEEE International Performance, Computing, and Communications Conference*, pages 8–14, February 1999, `ftp://ftp.eas.asu.edu/pub/cse/sasos/ipccc99.pdf`.

[SMR89] A. Straw, F. Mellender, and S. Riegel. Object management in a persistent smalltalk system. *Software–Practice and Experience*, 19(8):719–737, August 1989.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *17th ACM Symposium on Operating Systems Principles (SOSP '99)*, 1999, `http://www.eros-os.org/papers/sosp99-eros-preprint.ps`.

[Str94] Bjarne Stroustrup. *The design and evolution of C++*. Addison-Wesley, Reading, 1994.

[Sun] Sun Microsystem. *Java OS*, `http://java.sun.com/products/javaos/`.

[Sun95] Sun Microsystems. *The Java Virtual Machine Specification*, 1.0 beta edition, August 1995, `http://java.sun.com/doc/vmspec/VMSpec.ps`.

[SW] Jonathan Shapiro and Sam Weber. A family of securable protection systems - draft, `http://www.cis.upenn.edu/~shap/EROS/MS-CIS-98-18.ps.gz`.

[TRC95] See-Mong Tan, David K. Raila, and Roy H. Campbell. An object-oriented nano-kernel for operating system hardware support. In *Proceedings of 4th International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, Lund, Sweden, August 1995. IEEE Computer Society.

[Unia] University of California at Riverside. *Home Page for DYLAN Language*, `http://cuda.ucr.edu/Page_lang/inet_links/dylan.html`.

[Unib] University of Oviedo, Spain. *Oviedo 3 - An Object Oriented Integral System, Home Page*, `http://www.uniovi.es/~oviedo3/principal/e-frames.htm`.

[Val] Andrew Valencia. *A Tutorial for GNU Smalltalk*. Valencia Consulting, `http://www.smalltalk.org/versions/GNUSmalltalk.html`. (online documentation of GNU Smalltalk 1.1.5).

[VD92] Francis Vaughan and Alan Dearle. Supporting large persistent stores using conventional hardware. In *Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992. Springer-Verlag, `http://docs.dcs.napier.ac.uk/DOCS/GET/vaughan92a/document.ps.gz`. available online only, in ftp.gh.cs.su.oz.au as GH-02.

[VDC+94] Francis Vaughan, Alan Dearle, Jiannong Cao, Rex di Bona, Matthew Farrow, Frans Henskens, Anders Lindstrom, and John Rosenberg. Causality considerations in distributed persistent operating systems. In Gopal Gupta, editor, *Proceedings of the Seventeenth Annual Computer Science Conference, ACSC-17, Part B*, pages 409–420, Christchurch, New Zealand, January 1994. `http://docs.dcs.napier.ac.uk/DOCS/GET/vaughan94a/document.html`.

[Voc98] Jerry Vochteloo. *Design, implementation and performance of protection in the Mungi single-address-space operating system*. PhD thesis, University of New South Wales, 1998, `ftp://ftp.cse.unsw.edu.au/pub/users/disy/papers/Vochteloo:phd.ps.gz`.

[VRH93] Jerry Vochteloo, Stephen Russell, and Gernot Heiser. Capability-based protection in the mungi operating system. In *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems*, pages 108–115, Asheville, NC, USA, December 1993. IEEE Computer Society, `ftp://ftp.cse.unsw.edu.au/pub/users/disy/papers/Vochteloo_RH_93.ps.gz`.

[Way94] Peter Wayner. Objects on the march: The trend is toward an object-oriented approach to the design of operating systems. *Byte Magazine*, 19(1):139, January 1994.

[WG98] Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(5):980–1013, September 1998, `http://www.acm.org:80/pubs/citations/journals/toplas/1998-20-5/p980-wag%ner/`.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop*

on Memory Management, volume 637 of *Lecture Notes in Computer Science*, University of Texas, USA, 16–18 September 1992. Springer-Verlag, `ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps`.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing series. MIT Press, Cambridge, Massachusetts, February 1993.

[Wir] Niklaus Wirth. *A Brief History of Modula and Lilith*, `http://www.modulaware.com/mdlt52.htm`.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Texts and monographs in computer science. Springer, Berlin, 3rd edition, 1985.

[Wir96] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996.

[WK92] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge addresses on standard hardware. In *International Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, September 1992. IEEE Press.

[WMR+95] Tim Wilkinson, Kevin Murray, Stephen Russell, Gernot Heiser, and Jochen Liedtke. Single address space operating systems. Technical Report UNSW-CSE-TR-9504, School of Computer Science and Engineering, University of New South Wales, Australia, 1995, `ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/9504.ps.Z`.

[Wya94] Geoff Wyant. Introducing modula-3. *Linux Journal*, 8, December 1994, `ftp://ftp.gte.com/pub/m3/linux-journal.html`.

[Yok92] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 414–434, New York, NY, October 1992. ACM Press, `ftp://ftp.csl.sony.co.jp/CSL/CSL-Papers/92/SCSL-TR-92-014.ps.Z`.

[Yok93] Yasuhiko Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. In *Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 145–162. First JSSST

International Symposium, November 1993, `ftp://ftp.csl.sony.co.jp/CSL/CSL-Papers/93/SCSL-TR-93-014.ps.Z`.

[YTM+92] Yasuhiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. *The Muse Object Architecture: A new operating system structuring concept*, pages 27–51. May 1992.

[Zan91] Fausto Zanasi. *Teoria e pratica del linguaggio Prolog*. Calderini, 1991.