Computer Science
University of Manchester

From Types to Dataflow: Code Analysis for an Object-Oriented Language

Andrew J. Barnard

Technical Report UMCS–93–10–2

FROM TYPES TO DATAFLOW
Code Analysis for an Object-Oriented Language

By
Andrew J. Barnard
Department of Computer Science

October 1993

Contents

List of Figures

Abstract

Of all the object-oriented programming languages and systems to appear over the past decade, Smalltalk-80 has remained one of the more popular ‘pure’ object-oriented languages. However Smalltalk-80 has suffered criticism due to its relatively poor run-time efficiency, together with the lack of compile-time feedback for the programmer. In the past, type systems have been nominated as potential solutions to both these criticisms.

The aim of this thesis is to compare and contrast what a type inference system may offer a language like Smalltalk-80, with what is possible from an alternative form of code analysis, namely dataflow analysis. To this end, a type inference system and a dataflow analysis system are both formally defined for a Smalltalk-like language, and their relative properties examined.

While both systems are revealed to be capable of detecting potential ‘does-not-understand’ errors, the type inference system is shown to conflict with the semantics of the original untyped language. In contrast, the dataflow analysis system is shown to have no such problems. In addition, the dataflow analysis system is shown to be capable of extending the possibilities of analysing control structures beyond what is feasible with a type inference system.

For Keith, Gisela and Nic.

Acknowledgements

Half past ten and not a word written. Scannell despises himself. Just one sentence, he tells himself, just one little trickle of type to start the flow, unlock the flood…

from Keeping it Clean *by Andrew Davies*

I am deeply indebted to Mario Wolczko, who has supervised the work contained in this thesis. Without his time, patience and good guidance none of the work presented herein would have been possible. In addition, I must thank Trevor Hopkins for providing the official backing for this work. Again, without Trevor's signature in all the right places, none of this work would have been possible. As far as official bodies go, I wish to thank both the SERC and the DSS for their financial support during the execution of the work reported in this thesis, and during the tortuous writing-up phase. Thanks must also be directed to the Department of Computer Science at the University of Manchester for the use of their facilities, and in particular, to the late MUSHROOM group (The MUSHROOM is dead! Long live the MUSHROOM!). To all the various people who I have come to know during my time in Manchester, thanks for making my time here so enjoyable. Finally, thanks must go to Ian Piumarta for providing accommodation above and beyond the call of duty, especially during the crucial, final days in the preparation of this thesis.

The author spent a total of seven years at Manchester University, originally graduating with a B.Sc. in Computer Science and Mathematics in 1987. He stayed on as a research student and graduated with an M.Sc. in Computer Science in 1989. After leaving Manchester in 1991, he returned to his birthplace, Leeds, and subsequently took up employment at The University, where he is currently employed as a Research Assistant. In addition to his work-related activities as an "object engineer", during his spare time the author enjoys watching football, taking an interest in film studies, and honing his deadline-missing skills to a fine art.

Colophon: This thesis was typeset using the psLaTeX document preparation system. The main body of text was typeset in Times Roman, while the chapter quotes were typeset in Bookman Light Italic and Avant Garde Oblique.

Chapter 1

Introduction

The last thing one knows in constructing a work is what to put first.

Blaise Pascal

Efficiency — the watchword for any new programming language. Any new language, or programming technique will eventually be evaluated against some form of efficiency criterion, whether it be efficiency of execution or efficiency of coding time. In this thesis, the object-oriented programming methodology will be looked at from this perspective. Already acknowledged as a major advance in the area of speedy construction of prototype applications (e.g. [Cox86]), pure object-oriented languages have long been criticised for their apparent lack of execution speed. In addition, for the archetypal popular object-oriented language, Smalltalk-80 [GR89], there have been criticisms of the lack of programmer feedback during coding, not so much during the rapid-prototyping phase of application development, but rather during the creation of a finished product. It is from this basis that the work presented in this thesis proceeds.

Specifically, this thesis will look into the possibilities of adding code-analysis mechanisms to extract information from the program code, with a view to using this information to improve either of the efficiency criteria mentioned above. The two code-analysis mechanisms chosen for examination are a type inference system and a dataflow analysis system, both applied to a Smalltalk-like object-oriented language.

1.1 Background

During the past decade a form of programming, known as object-oriented programming, has developed from being a little-known research curiosity at the start of the 1980s, to being a major force in the world of computing, hailed as the 'next big thing' after the revolution of structured programming in the 1970s. In fact Rentsch's prediction of an explosion of independent and seemingly incompatible interpretations of 'object oriented programming' [Ren82] has come true, with the number and variety of 'object-oriented' languages and utilities increasing all the time, each with their own individual interpretations of what constitutes 'object-orientedness'. There are now numerous languages which claim to be object-oriented, from derivative, hybrid languages (C++ [Str91] and Objective-C [Cox86], both based on C [KR88]), through Lisp-derived languages (Common Objects [Sny86], CLOS [DG87] and Flavors [WM80]), to 'pure' object-oriented languages (Smalltalk-80 [GR89], Eiffel [Mey88] and SELF [US87]). In addition there have been several languages which have used object-oriented techniques for more specialised purposes, for example Emerald [HRB+87], which is aimed at distributed systems, ABCL/1 [YBS86], which is aimed at concurrent systems, and Gemstone [MSOP86], which is an object-oriented database system closely modelled on Smalltalk-80. However, for the purposes of this thesis we will return to basics, back to the founding principles as laid down by the most well known of the early languages: Smalltalk-80.

Smalltalk-80 was developed at the Xerox Palo Alto Research Centre (Xerox PARC), and was the culmination of a series of languages based on principles originally found in the simulation language Simula [BDMN73], the ancestor of all object-oriented programming languages. During the 1970s the Learning Research Group at Xerox PARC, led by Alan Kay, designed a series of languages based on the notions of objects and classes originally found in Simula.

Smalltalk went through several manifestations, from Smalltalk-72 [Sho79] and Smalltalk-76 [Ing78], through to its present incarnation, Smalltalk-80. The present language was widely publicised by the release of a group of commercial implementations and the publication of a set of technical books covering various aspects of the language, its novel graphical interface and its underlying implementation [GR83, Gol84, Kra83, GR89].[1]

Since its original commercial release by Xerox PARC, and its later support by the commercial spin-off company, ParcPlace Systems, Smalltalk-80 has gone through several transformations (especially with Release 4), but the underlying principles have remained, by and large, unchanged.

1.2 Motivation

The motivation behind the work presented in this thesis can be summed up as "the desire for greater efficiency". But the question then arises, "what kind of efficiency?" Within the context of this thesis, the 'efficiency' in question, applied to a language based on Smalltalk-80, will be interpreted in two ways:

- the desire for increased run-time speed (i.e. the efficiency of the run-time system), and
- the desire for greater compile-time feedback, leading to the early detection of coding errors (i.e. the efficiency of programmer coding).

[1] The first three books are known informally as the Blue Book, the Orange Book and the Green Book. The fourth book, purple in colour, is a revised version of the blue book, omitting the sections dealing with the implementation of the language. The introduction of Smalltalk to a wider community was also marked by a special issue of the magazine BYTE, which gave over a large part of its contents to the 'new' language [Byt81].

Naturally, tied up with both these interpretations is the position of the compiler in determining:

- how efficient the compiled code is, and
- how much the time spent compiling the code intrudes on time which the programmer could otherwise spend producing further code or testing the existing code.

The latter point is especially relevant given Smalltalk-80's incremental, method-by-method coding and compiling strategy.

Returning to the first of the two desires listed above, in terms of raw execution speed, for example "number crunching", it is true that most implementations of Smalltalk-80 are significantly slower than a more conventional language such as C. In [CU91], Chambers and Ungar write:

> For example, the fastest commercial implementation of a pure dynamically-typed object-oriented language, ParcPlace Smalltalk-80, runs a set of small C-style benchmarks at only 10% the speed of optimized C; this implementation contains techniques developed by Deutsch and Schiffman [DS84] that are widely regarded to be the state of the art in software techniques for building fast implementations of pure object-oriented languages.

However this statistic does not make any allowance for the nature of programming in Smalltalk-80, where the power of the language appears to lie in the ability to construct prototype applications quickly, and to test out design ideas before embarking on a more considered implementation phase from which the final product, in a state acceptable to hand over to a user, will be derived.

With respect to the second point, the desire for greater compile-time feedback, the ParcPlace Smalltalk-80 compiler is currently only able to detect undeclared variables and message selectors which do not have a corresponding implementation defined in any of the class definitions. One of the motivating factors behind research into adding type systems to Smalltalk-like object-oriented languages is to improve such programmer feedback. By associating type information with parameters and variables in the program code, the argument goes, such information could be used to discover mismatches between what a potential receiving object is expected to understand and the messages to be sent to it (the infamous 'does-not-understand' error). How such type information is added to the program code would seem to lie at the heart of the acceptability of this 'solution', as shown by moves away from type declarations, and towards type inference [Joh86, Gra89, PS91a].

To summarise, programmers want more from their systems — better speed, better feedback. It is proposed that the fulfillment of one or other of these desires, or even both, can be aided either by information provided by the programmers themselves — annotations, for example types, which the compiler can use either to check for consistency within the program code, or to help introduce optimisations into the compiled code — or by information derived from the code itself, as is the case for both type inference and dataflow analysis. In keeping with the shift in interest from type declaration systems to type inference systems in recent years, the work presented in this thesis deals with the latter of these two sources, namely information derived directly from the program code. To this end, this thesis will present two contrasting code analysis techniques for a simple Smalltalk-like object-oriented language: type inference and dataflow analysis.

1.3 Practicalities

This thesis is organised as follows. The rest of this introductory chapter consists of an overview of the basic principles underlying the object-oriented language used in this thesis. The following chapter looks more closely at some of the issues motivating this thesis, and reviews some of the work that has been carried out in the field of adding types to Smalltalk.

Chapter 3 presents a detailed formal description of a type inference system, based closely on the system designed by Justin Graver for the language Typed Smalltalk (TS). The type inference system defined in Chapter 3 follows Graver's design in its basic structure, but has been cleaned-up and re-modelled for a much simpler language. This chapter finishes with critiques of both the type inference system defined, as well as type systems in general as applied to a previously untyped object-oriented language. Readers more interested in dataflow analysis may wish to skip Chapter 3 on first reading.

Chapters 4, 5 and 6 are taken up with an extended description of a dataflow analysis system for a simple object-oriented language. The language used in these chapters is a modified version of that used in Chapter 3. Chapter 4 presents the basic core of the analysis process, introducing the major elements. Chapter 5 extends the process to handle primitive methods, while Chapter 6 introduces the analysis of Smalltalk blocks. Chapter 6 finishes with a look at how the dataflow analysis system might be extended further, a description of how information derived from the system could be used, and finally, a discussion of how this form of analysis compares with systems based on types.

Chapter 7 finishes off the thesis by drawing together the main conclusions from each of the code analysis systems, and speculates on how the results might be applied to future work.

Throughout this thesis the reader is assumed to be familiar with the VDM notation described in [Jon86, Jon89], and the programming language Smalltalk-80 [GR89] (although the following section does introduce some of the basic properties of object-oriented languages). A glossary covering the notation used in this thesis can be found in Appendix B. Interested readers may also wish to consult [Wol88] which contains the denotational semantics of the simple object-oriented language used as the basis for the languages used in this thesis.

1.4 What is object-oriented programming?

At the heart of object-oriented programming is the *object*, a structure containing state and functionality, encapsulated from the outside world and distinguishable by its unique identity. The state of an object is recorded by its internal structure, commonly known as its *instance variables*. These instance variables reference other objects, which in turn have instance variables which reference other objects, and so on, recording the state of the initial object. However the state of the instance variables for a given object are not directly accessible from outside that object. Instead, access and modification to the state of an object must be performed via the declared functionality of that object, commonly referred to as its *methods*. The execution of an object's method is carried out by sending that object a *message* with any required arguments. From the point of view of writing code, objects are accessible via variables, with each object assigned a *unique identity* to distinguish it from all other objects. The following code assigns the object 42 to the variable aVariable. The message isInteger is then sent to the object 42 via the variable.

```
aVariable := 42.
aVariable isInteger
```

Once an object has been assigned to a variable, and so long as that variable is not involved in a reassignment, the variable will always reference the same object, however much that object's state may be altered through message sends.[2] One important property of objects and variables is that, by traditional imperative standards, the variables are untyped. In Smalltalk-80, the only property guaranteed for a variable is that it will always be referencing an object — even 'uninitialised' variables reference an object, namely the 'undefined object', nil. As such there will always be an object to receive a message sent via a variable. The problem of whether the object will understand the message is a separate issue. In addition, because expressions are not statically typed, an expression sending a message is capable of being reused for any object able to understand that message.

In the definitions given above, the object is assumed to carry around its own functionality, in addition to its state. With the addition of the ability for an object to clone itself, this forms the basis of the language SELF [US87].[3] However for more common languages there is a different method of grouping together the functionality of separate objects, namely the *class*. A class provides a definition of both the internal structure and the functionality of objects that are derived from it. These objects are then *instances* of that class (hence instance variables), and, after copying their internal structure from the template provided by the class, are able to refer back to that class in order to access the method code to be executed for a given message. In this way objects of the same class are able to share the code representing their functionality while retaining their own private state. Returning to SELF, which has no notion of a class, a similar 'template' service is provided by prototype objects. Such prototype objects are cloned to produce new objects which in turn can be altered, either to change their state or to change their functionality. However the prototype objects themselves must not be altered. Such necessities introduce problems into the SELF language, where the lack of distinction between 'template' objects and working objects makes the accidental alteration of a key prototype object, instead of a clone of that object, a real possibility with potentially far-reaching consequences [Smi92].

[2] The exception to this rule in Smalltalk-80 is the become: primitive, which swaps two objects such that all the variables that were referencing one are now referencing the other, and vice versa. However it could be argued that this primitive violates the basic principles of object-oriented programming languages.

[3] "Probably the world's purest object-oriented programming language" to paraphrase a well-known beer commercial.

The final element in the world of object-oriented languages is *inheritance*, a convenient mechanism whereby code can be shared between objects of different overall functionality. In SELF inheritance occurs between objects, where one object can treat another object as its parent, passing on messages which it doesn't understand. In Smalltalk-80, inheritance is a relationship between classes. When one class inherits from another class, the former conceptually takes a copy of the latter's state template and methods. Instances of the child class then have the combined internal state of that class and all its ancestors (a parent can have a parent), as well as their combined functionality made up of the most recent definitions of each method (methods may be reimplemented in a child class, and hence this would be the most recent definition). In reality the code remains with parent class and the relationship is used in the same way as SELF, with the child class passing on messages for which it does not provide the method code. As with classes, inheritance would appear to be more of a convenient organising tool rather than an essential part of object-oriented programming languages. In fact, as both Wolczko [Wol88] and Graver [Gra89] have shown, it is possible to process out the simple, single inheritance described here, to produce a simple set of classes.

When compared to traditional imperative languages, for example Pascal, there are some features of object-oriented languages which stand out. All 'values' are objects, with equal rights, and can only be dealt with via message sends. Hence all operations must be carried out via message sends. However, because variables have no information regarding the objects which may be assigned to them, there is no way of knowing in advance which method should be invoked for a given message send. This is in stark contrast to the monomorphic world of the traditional imperative language where procedures and operations (e.g. integer addition, +) are normally bound at compile time. The final major difference is that with languages like Smalltalk-80 and SELF, the user interacts with the language through an interactive programming environment [Gol84]. As a result, unlike traditional imperative languages, there is no real 'main body' within the 'program' from which the rest of the code is invoked. This feature, or lack of a feature, has major consequences for both type inference and dataflow analysis.

In summary, the language to be used as the basis for the work presented in this thesis will be a simple object-oriented language, with the basic properties of unique object identities and encapsulation of state (instance variables) through an object's methods. In addition it will use classes as a means of organising the functionality of the objects. Finally, the chosen language will not use inheritance. This latter choice is made with the understanding that were inheritance to be used, it could be processed out prior to the application of either the type inference system or the dataflow analysis system. In fact, this is what happens in Graver's type inference system [Gra89].

Chapter 2

Why analyse programs?

There lies at the back of every creed something terrible and hard for which the worshipper may one day be required to suffer.

E. M. Forster

This chapter will look more closely at the two problems highlighted in the introduction. In particular it will look at the question of how 'efficiency' may be perceived, both by the programmer and by the end user of a Smalltalk application. After these problems have been discussed, it will go on to look at how types have been used in the past to attempt to solve these problems.

2.1 Perceptions of Efficiency

One of the problems in determining the effectiveness of a proposed solution to the question of improved efficiency lies in establishing that that solution does provide an *overall* improvement over not using it. This is particularly relevant for Smalltalk's incremental development environment, where the compiler is invoked after each method is coded. As such, any effort made to "improve efficiency" during an incremental compilation must be weighed up against any inconvenience introduced to the programmer as a result. In the following two subsections we will look at how the problems of 'efficiency' might be interpreted, and how a solution might be evaluated, from both the programmer's and the end user's perspectives.

2.1.1 Perceptions of Programmer Efficiency

From a programmer's point of view, improved efficiency could be interpreted as the overall reduction in time spent dealing with coding errors. From a code analysis perspective, this would involve the detection and elimination of ‘does-not-understand’ errors.[1] In turn, the acceptability of a solution to reduce the time spent on such errors may hinge on how much additional effort the programmer must invest in their detection (for example, through the addition of type declarations, or the overhead of a type inference system).

When coding in Smalltalk-80, with no real feedback from the compiler, the programmer will be unaware of any potential errors, including ‘does-not-understand’ errors, which may occur when the code is run. As a result, it is in the programmer’s best interest to test the code regularly as the software is developed, to be sure that it works as expected. It would seem beneficial to the programmer if potential ‘does-not-understand’ errors were detected at compile time. Given that, from a compile-time perspective, such a mismatch between a receiver object and a message selector would be seen as a mismatch between, for example, a variable and a message selector, a prime candidate for a solution to the problem would appear to be the introduction of a type system.

As was mentioned briefly in the introductory chapter, the information upon which such a type system could be based must be provided either by the programmer, or directly from the code through a type inference system. In the former case, where the information is provided by the programmer, it would come in the form of redundant type declarations — redundant because the code would work just as well without them. Naturally this would involve an additional, potentially unwanted burden on the programmer. In addition, depending on how the type system is defined, the type declarations could interfere with good code re-use practices. Consequently, as the software being coded is developed, the programmer may be called upon to re-type old code in order to ensure that the program as a whole remains consistent with respect to the declared types.

Clearly it would be preferable for the programmer if the type information were to come through a type inference system. The conclusion to be made here would seem to be that by removing the burden of type declarations from the programmer, the type inference system, if one could be designed, would be better suited to provide the error-detecting service required. However, the type inference system would not be without its own overheads, notably the time spent analysing the code to extract the type information. Assuming that the chosen system is to be incorporated as part of the incremental compile-time activities, it is clear that both systems would incur some form of overhead, whether it is the need for explicit type declarations or the time spent inferring types.

Unfortunately any type system which aims at detecting potential ‘does-not-understand’ errors will ultimately only ever be able to detect the manifestation of an error rather than its underlying cause.[2] In addition, there would be other coding errors, less amenable to automated detection but which would still have to be detected and corrected through programmer-driven testing. Given that the introduction of a technique for detecting coding errors at compile time will naturally introduce its own overhead, it seems sensible to ask whether such time or effort

[1] There is really only one basic error conducive to detection by a code analysis system, namely the ‘does-not-understand’ error — all other errors are up to the programmer to find and correct through thorough testing.

[2] This is because it is unlikely that the ‘does-not-understand’ error is an isolated error, rather than the manifestation of a coding error in some other part of the program. However, for a type system based on redundant declarations, the manifestation of the error may be closer to its source than it would be for a type inference system.

could be spent more profitably testing the code, since such testing ought to be part of the development process.

One factor in determining whether this latter question should be answered in the affirmative would seem to be the relative overhead of correcting an error once it has been detected. For some languages, correcting an error detected at compile time may be significantly quicker than detecting and correcting a similar error during testing. This would follow as a result of the overheads incurred by recompiling or relinking the whole program, or in any initialisations necessary before the suspect code can be invoked and tested. However this does not necessarily apply to a dynamically bound programming language like Smalltalk, where fast incremental compilation together with dynamic binding of method invocations means that code can also be tested, and debugged, incrementally. In fact, the overhead of checking for type errors during compilation may wipe out the advantages gained by Smalltalk's distinctively quick code/compile/test cycle, where the compiler is invoked briefly after each method is coded before the programmer moves on to code the next method or test the code already entered. Naturally if the overhead of the type system does not impinge noticeably on the programmer, this complaint would not hold.

To summarise, from a programmer's perspective, the acceptability of a compile-time technique to reduce the time spent dealing with 'does-not-understand' coding errors will be based as much on the negative impact such a technique might have on the programmer's practices, as on the positive benefits it might bring. Negative impacts might include the burden of adding redundant type declarations, or any excessive time overhead introduced by a type inference system on the speed of the compiler.

2.1.2 Perceptions of Speed

In contrast to determining whether the introduction of a particular technique will improve programmer efficiency, determining an improvement in the state of system efficiency can be done by straight-forward analysis of objective timings.[3] Given that this is possible, the question then arises, "where does all the time go?", for which the simple answer is: "message sends" [Kra83]. Because all computations are effectively performed by message passing, it follows that by speeding up the message sending process, or by eliminating it completely, it should be possible to gain a corresponding improvement in run-time efficiency.

In fact, this is what already happens with current commercial implementations of Smalltalk-80, where a variety of techniques are used to reduce the overhead of a message send. Such techniques [DS84, Kra83, Ung87] include:

- method caching, in which a hashed table of class/selector pairs is used to prevent the expense of a full method lookup, and
- in-line caching, in which a message send expression is pre-emptively bound to the last

[3]"There are lies, damn lies and Objective Timings": Naturally, objective timings are subject to the vagaries of interpretation in the same way as statistics — in particular to the choice of code from which the timings are obtained.

method invoked by it; a test is inserted to check that this is the correct method and if the test fails, a proper method search is carried out.

The latter of these two techniques is based on the statistical observation that for a given message send, the probability is high that it will invoke the same method that it invoked the last time it was executed.

Additional solutions include the special-casing of some message selectors. For example, ifTrue:ifFalse:, the Smalltalk equivalent of `if-then-else`, is treated by the compiler as if it is a message that will always be sent to an object of class True or class False, irrespective of how the programmer may wish to implement a corresponding method in another class. Such an optimisation, based on the popular use of such methods, is carried out at the expense of a degree of flexibility for the programmer.

Alternative techniques have been suggested to improve the efficiency of the compiled code by involving some form of type system. These techniques are based on the principle that if the class of the receiving object is known at compile time, then the method corresponding to the received message can either be bound statically (so reducing the cost of the method invocation) or in-lined into the compiled code at the point of the message send (so saving the cost of creating a context for the invoked method). In fact, with the method code in-lined, it is argued that additional optimisations could be performed, so improving the efficiency of the compiled code even further [Joh86].[4]

The significance of the type system lies in its provision of type information for variables. From the type information associated with a variable it should be possible to derive details about the kinds of objects which may assigned to that variable. Unfortunately, from an 'efficient-code' perspective, good quality type information tends to come in the form of explicit class names rather than more flexible typing schemes. Such type information is, in general, counter-productive with respect to effective code re-use. In addition, the usefulness of type information coming directly from the programmer will be greater than for that coming via a type inference system [Bar89]. As well as having possible consequences with respect to code re-use, such a system would impose a further burden on the programmer.

As a final comment, it would seem reasonable that if it is the run-time efficiency of the end-product that is important, then the efficiency of the compiler will be of secondary importance to the quality of code produced.

2.1.3 Summary

The main conclusion to be drawn at this point is simply a cautionary one. If the aim is to improve one aspect of the Smalltalk programming experience (e.g. the early detection of some programming errors through some form of type system) then it is important that another aspect is not hindered to the detriment of the overall system (e.g. a deterioration in the code/compile/test cycle whereby the programmer spends an unacceptable amount of time

[4]This also brings up the question of debugging optimised code, not to mention the problem of what to do when a method which has been in-lined needs to be altered. However such questions are beyond the scope of this thesis.

adding redundant type declarations or waiting for the compiler to finish). It may be that the needs of the programmer and the desire for a fast efficient finished product require two different solutions — perhaps two different compilers: a fast version to be used during the incremental development phase, and a second, slower version capable of producing a heavily optimised final product.

Given that the aim of this thesis is to look into code analysis techniques, and taking into account the manner of their examination, it would appear that more useful results may be obtained by concentrating on the quality of information derived by the analysis systems, rather than their efficiency in obtaining it.

In order to provide some background for the rest of this thesis, the following section will look at some past attempts at introducing types into the (statically) untyped language, Smalltalk-80.

2.2 Types in Smalltalk

Smalltalk-80 is untyped to the extent that variables and parameters do not have any properties assigned to them other than that they reference objects. As such Smalltalk is not statically typed.[5] A type system for Smalltalk would allow information, in the form of types, to be bound at compile time to individual variables, parameters and, ultimately, expressions. This type information could then be used to reason about the uses of the variables, parameters and expressions before the code is actually executed. It should be noted that the type of a variable (or parameter) is normally fixed for its lifetime, with the statically assigned information being compatible with any object assigned to that variable. As will be seen in subsequent chapters, the imposition of this restriction when adding a type system to Smalltalk will have significant repercussions (§3.8).

The problem of introducing a type system into an 'untyped' object-oriented programming language was addressed by the author in [Bar89]. In particular the structures of several type systems were described. The rest of this section will provide a brief review some of the type systems featured in [Bar89] in order to provide some background for the type inference system introduced in Chapter 3.

Suzuki [Suz81] made the first attempt to introduce a type inference system into Smalltalk, in this case Smalltalk-76, with the aim of providing feedback for the programmer. A detailed description of Suzuki's algorithm can be found in [Bar89]. Unfortunately due to the restrictions of the language implementation, Suzuki was unable to produce a fully functioning system. Subsequently he joined forces with Terada [ST84] with the aim of producing an efficient implementation of Smalltalk-80. To this end Suzuki abandoned the idea of using type inference to provide type information, and turned instead to type declarations. From an historical perspective, the paper which presented the pair's findings was published at the same time as Deutsch and Schiffman's seminal paper on an efficient implementation of Smalltalk-80 [DS84], and as a result Suzuki and Terada's work was put somewhat in the shade.

[5] Smalltalk can, however, be considered dynamically typed (see [Bar89], §2.3).

Borning and Ingalls [BI82] developed a type declaration system for Smalltalk-80 to provide programmer feedback. However their system was ultimately flawed due to a badly designed type structure. A description of this system may also be found in [Bar89]. The idea of adding redundant type declarations was taken up by Roberts [CR86, Rob86b, Rob86a, Rob90] who developed a version of Smalltalk, called Smalltype, which was also able to provide programmer feedback, in the form of warnings, if the type declarations proved inconsistent. Given that within this thesis we are interested in code analysis, none of these type systems are particularly relevant to the work presented here.

The idea of using type information to aid better code generation was picked up by Classy [LB83], a version of Smalltalk in which the programmer was able to add 'hints'. These 'hints' were then used to apply rewrite rules, effectly in-lining code. Atkinson's Hurricane [Atk86, Bar89] also used type 'hints', however these were used to construct control-flow graphs. By using the type hints, Atkinson was able to propagate type information out of message send points and into potential corresponding methods. The resulting control-flow and dataflow graphs could then be analysed and used to introduce code optimisations, with the optimised version of a method being compiled in addition to a non-optimised version. By being based on 'hints' rather that hard and fast type declarations, it was necessary to introduce run-time class tests into the code to ensure that the optimised version of a method was only used when the classes of the arguments matched the classes given by the programmer-provided hints. One of the interesting features of Hurricane was its use of call-site information in the propagation of type information. Instead of providing a general-purpose type for each method, methods were dealt with within the context of an individual message send expression.

Johnson's Typed Smalltalk [Joh86] was the first type system to address the problem of type safety, where the use of a variable or parameter is guaranteed to be compliant with its (declared) type. His original system used type declarations which could be checked to ensure that they were consistent. The need for type safety was paramount to Johnson's aim of developing a highly optimising compiler for his language. His argument was that if the compiler is able to trust the type declarations completely, then, for example, the kind of run-time class tests required by Hurricane could be avoided. A detailed analysis of Johnson's type system by the author can be found in [Bar89]. Subsequently a type inference system was added to Typed Smalltalk [Gra89] as a means of relieving the programmer from some of the burden of declaring types. This type inference system forms the main subject of Chapter 3.

In addition to the work on Smalltalk-80, there has been some work done on the general problem of typing imperative object-oriented languages. Palsberg and Schwartzbach have looked at the problems of both type inference [PS91a] and of type declarations in presence of inheritance [PS91b]. Although their language has some novel features which distinguish it from Smalltalk, the more general conclusions made in Chapter 3 can be applied to their system. Hense [Hen90, Hen91] has looked at the problem of incremental type inference and so has developed a way of defining 'extensible' types to aid typing in the presence of code re-use.

SELF [US87, CU89, CUL89, CU91], the language introduced in the first chapter, has had several generations of compilers developed for it over a period of time, each adopting various optimisation techniques, for example, compiling specialised versions where possible or practical, and applying a form of 'type analysis' to the contents of a control-flow graph. In fact the type analysis used by the SELF compiler exhibits some of the properties of the dataflow system described in this thesis, namely that the type of a variable is not fixed, but changes from node to node in the control-flow graph as type information is propagated. For example, the type of a variable will be different between those nodes preceding an assignment and those following it.

2.3 Summary

This chapter has looked at the problem of improving 'efficiency' from both the programmer's perspective and from the point of view of an end user. From the programmer's perspective, improved efficiency was interpreted as the reduction in time spent dealing with coding errors. However, it was noted that any improvement provided by a code analysis system would have to be weighed against any overhead such a system might introduce. From the perspective of an end user, efficiency was interpreted as the efficiency of the running code. To this end, it was noted that the efficiency of the compiler would be secondary to the quality of code produced. With these two areas in mind, it was decided to examine the chosen code analysis systems with respect to the quality of information produced, rather than the efficiency with which such information may be extracted.

The next chapter will deal with a type inference system. The chosen system is based on the type inference system designed for the language Typed Smalltalk, which has been implemented at the University of Illinois at Urbana-Champagne. Although the type inference method used by this system is not the only possible method, the system does exhibit most of the salient features a typed version of Smalltalk would possess.

Chapter 3

TS — A type system

The connection between the language in which we think/program and the problems and solutions we can imagine is very close. For this reason restricting language features with the intent of eliminating programmer errors is at best dangerous.

The C++ Programming Language, *Bjarne Stroustrup*

In the last chapter some of the ideas behind using types as an aid to program security and efficiency were introduced. In this chapter, one of the major exponents of the typing of Smalltalk will be described.[1]

The work done at The University of Illinois at Urbana-Champaign on the language *Typed Smalltalk* (TS) by the research group headed by Ralph Johnson [Joh86, JG88, JGZ88, Gra89, GJ90] was, until recently, the most advanced in this area. More recently other research groups have picked up on the general field of typing imperative object-oriented languages and some interesting results have emerged [Hen90, Hen91, PS91a, PS91b]. However the basic results of the work done on Typed Smalltalk apply to any proposed type system for the essentially untyped Smalltalk language. The most generally applicable results will be discussed in §3.8.

The structure of this chapter is as follows. In order to introduce the reader to the basic principle behind Graver's type inference system, the notion of *abstract interpretation* will be described. Following this, the description of the actual type inference system commences with an informal overview of how the system works. Section 3.3 introduces the abstract syntax of the language being typed. The language being used as the basis for this system is a subset of the full Smalltalk language, retaining enough of the original language's properties to convey the principles behind the type inference system together with the major consequences of such a system's use.

[1]Readers more interested in dataflow analysis may wish to skip this chapter on first reading.

Section 3.4 continues with the all-important definition of the *type* which forms the basis of Johnson and Graver's type system. Also within this section are descriptions of the *type environment*, which is required to keep track of the type information generated by the inference system, together with a few useful auxiliary functions.

The description of the type inference process itself begins in §3.5, and continues through §3.6. In these sections the processes of resolution (§3.5), transformation (§3.5.2) and reduction (§3.6.1 to §3.6.3) are described. Respectively these processes *resolve* all references to self, *transform* the code to a form suitable for interpretation over the domain of types, and finally *reduce* the transformed code through this interpretation to yield a typing for the code.

The rest of this chapter is taken up with an analysis of the product of this type inference system, together with a discussion of how some of the problems evident in the type system apply to any proposed typing scheme for Smalltalk.

3.1 Abstract interpretation

Lying at the heart of Graver's type inference system is a technique known as *abstract interpretation* [CC77, MJ81]. This technique is also known, more descriptively, as *non-standard interpretation*. This is also the technique used at the heart of the dataflow analysis system described in the following chapter.

In essence, abstract interpretation is the interpretation of code over a value domain other than the domain used at run time. Consider the following simple expression:

$$6 * -7$$

If the normal domain of interpretation is the set of integers, and the $*$ operation represents integer multiplication, then the result of a standard run-time interpretation would be the integer –42.

In order to illustrate a non-standard interpretation, consider the following domain:

$$\{\text{'}-\text{'}, \text{'0'}, \text{'+'}\}$$

where a value in the new domain is derived from a value in the standard domain according to whether the standard value is negative, zero, or positive. With respect to the new non-standard domain, the above expression would be:

$$\text{'+'} \; * \; \text{'}-\text{'}$$

and the new non-standard interpretation would be '–', the result of multiplying a positive and a negative number.

In the case of Graver's type inference system, the non-standard domain is the domain of types. For example if the above expression was to be interpreted over a type domain containing the type Integer, rewriting the expression over the domain of types would yield:

$$< Integer > * < Integer >$$

Assuming for the moment that this is a Smalltalk expression, this would be interpreted as sending the message $*$ to an instance of class Integer. The result of the expression would then be the type of the result of such a message send, bearing in mind that the associated argument is also an instance of class Integer.

By interpreting the code over the domain of types it is possible to set up a collection of constraint relations between the types of the various components of the code (variables, parameters, method return types, etc.), and from these constraints obtain a suitable typing of the code which, in turn, can guarantee that the code is type correct. What these constraints consist of, and how they are derived, will be made clear in due course.

3.2 Type inference in TS

Graver's type inference system can be divided into two stages. The first stage transforms the program code, originally in the form of a parse tree, into an equivalent 'type tree', representing an initial typing of the code. In the type tree, the structure of the original code is retained, but where there were originally identifiers and literals, there are now types. The actual structure of these types will be described in §3.4, but essentially there are two basic kinds, *object types* and *type variables*. Object types are derived from, and associated with, individual classes. In order to handle polymorphism, types, both object and variable, may be grouped together in a set to form a third, composite type, the *union type*. In the context of type inference, the type variables are the more important: where the type of a variable or parameter is unknown, it is assigned a type variable. In fact, as will be seen in §3.7, type variables are fundamental to the whole system.

Once the type tree has been constructed, the second stage attempts to 'reduce' it to a single type. It is this act of reduction which ultimately produces the final typing of the program, and it is here that the constraint relations between type variables and other types are derived. The act of reduction is actually the act of interpreting a given expression over the domain of types. It is here that non-standard interpretation is seen in action, and the transformation stage can now be seen as a transformation from one domain, the domain of identifiers and literals, to another, the domain of type variables and types.

The actual process of reduction depends on the form of expression being processed. As an example, consider the following expression:

```
var := object
```

When this is transformed into an equivalent type tree, the variable on the left-hand side will have been assigned a type variable, say t_{var}, and the simple expression on the right side will either already have an object type (for example, if it is a literal) or have also been assigned a type variable. Assuming that the type of the right-hand side is t_{object}, the transformed assignment expression will be:

$$t_{var} := t_{object}$$

The process of reduction, in general, tries to reduce an expression to a single type, namely the type of the result of that expression. In the case of an assignment, the result of the whole expression is the result of the right-hand side expression. Therefore the reduced type of the above assignment will be t_{object}.

However, this is not the only information derived during the reduction process. For the assignment to be type correct, the type of the right-hand expression must be 'included' in the type of the left-hand variable. Using the type-inclusion relation, $\sqsubseteq$, this inferred constraint on the left hand type can be represented thus:

$$t_{object} \sqsubseteq t_{var}$$

By processing a sequence of expressions, a set of constraint relations can be built up. These inclusion relations then provide an ordering between types, object types, type variables and union types, and by combining them, an ordered graph may be constructed. This ordered graph can then be used to derive a mapping from type variables to object and union types which will satisfy the original constraint relations. The inclusion relation, $\sqsubseteq$, is described in §3.4.1 on page 35.

That, in essence, is how type inference is performed. However the situation becomes slightly more complex when message sends are considered. Reduction in the case of a message send is achieved by 'looking-up' potential methods and interpreting them over the non-standard domain, i.e. types. The types returned by each potential method can then be combined to give the reduced type of the message send.

Message sends require at least one corresponding method in order to be considered 'correct'. In order to get this far the receiver of the message must have a type, either an explicit object type or a type variable. Alternatively it may have a union type, in which case each component of the set making up the union must be handled separately. In the case where the receiver's type is an object type, the simulated lookup can be achieved simply by extracting the class from the type. In the case where the type of the receiver is a type variable, simulated lookup is achieved by examining the bounds of the type variable.

Type variables are initially unbounded. That is, when a type variable is first introduced, it is considered to be able to take on any object type, or union type. As the transformed code is reduced through interpretation, information, in the form of constraint relations, provides the type variable with a gradually refined set of object types over which it may range. Constraint information such as the lower bound derived from the assignment example above is known as dataflow type information.[2] Upper bound information, which tends to come from explicit type declarations, is known as requirements-flow type information.

Although upper-bound information provides a limit to the object types a given type variable may range over, it tends to represent information imposed on the code — in other words, explicit declarations. Lower-bound information, on the other hand, represents information derived *from* the code — it represents information derived through assignments and argument passing. As a result, it is normally the lower bound of a type variable which is used during simulated lookup. The exception is when there is no lower bound, in which case the upper

[2]Dataflow type information should not be confused with dataflow information, the most important distinction being the domains over which they apply: types for the former, values and objects for the latter. The importance of this distinction will become clear in the following chapter.

bound is used as an approximation. In TS, if the type variable remains unbounded, no lookup is attempted.

When a suitable method has been found, the message send is reduced by interpreting the invoked method using constraints derived from passing the message arguments. For example, consider the following message send:

```
rcvr mess: arg1 age: arg2
```

and the following method declaration:

```
mess: par1 age: par2
    "Example method."
    ...
```

Assuming that the corresponding types are t_{rcvr}, t_{arg1}, t_{arg2}, t_{par1} and t_{par2}, with t_{self} representing the type of the receiver within the method, the act of invoking the method would then provide the following constraint relations:

$$\begin{aligned} t_{rcvr} &\sqsubseteq t_{self} \\ t_{arg1} &\sqsubseteq t_{par1} \\ t_{arg2} &\sqsubseteq t_{par2} \end{aligned}$$

These constraints may then be used in the reduction, through interpretation, of the method body, with the resulting 'return' type being the result of the method. If the resulting reduced type of the method is t_{method}, and the reduced type of the message send is $t_{message}$, then the following constraint may be derived from the entire message send:

$$t_{method} \sqsubseteq t_{message}$$

The formal description which makes up the bulk of this chapter is of a simplified version of Graver's type inference system. The reasoning behind this choice lies with the aims of this chapter, namely to show how information is extracted from the program code, as well as to show the type and quality of such information. To this end, the finer details of what happens to this information after it has been extracted from the code — the unification and inference system which lies at the heart Graver's system — is only covered informally (§3.6.3). The description of the system presented in this chapter, therefore, only covers the analysis of the code.

As a byproduct of this reasoning, inheritance has also been omitted. In fact in Graver's own formal description of his system [Gra89], inheritance was shown to be processed out at an early stage, by systematically copying down inherited methods and statically binding super message sends. A variation of this technique of removing inheritance was also demonstrated in [Wol88]. In addition, Graver's more complex handling of primitives, with the inclusion of primitive-failure code, has also been omitted for the sake of maintaining simplicity.

In addition, blocks have also been omitted. It was originally envisaged that blocks would be better suited to dataflow/control-flow analysis, and as a result they were omitted from the system described in this chapter. This decision came about specifically after it became clear that the type system alone could not cope with non-local return blocks. Section 3.9 describes this problem, together with the implications for the use of blocks with respect to the type inference system.

3.3 The language

As has been mentioned already, the language being analysed does not use inheritance. Consequently a program simply consists of a *class name* to *class definition* mapping.

$$Program = ClassMap$$

$$ClassMap = ClassName \xrightarrow{m} ClassBody$$

$$ClassName \ :: \ Id$$

There is also no 'main expression', a technique used to represent a starting point for the program code, and which can been found in papers describing the semantics of such a language [Nor88, Kam88]. In the context of code analysis, such a technique loses its importance, and a more truthful notion of a program — a collection of methods with no predetermined flow of execution — can be appreciated. Ultimately the lack of a starting point is dealt with by processing the program repeatedly until a stable state is achieved. As with the dataflow analysis system described in the following chapter, the problem of a lack of a 'starting point' is overcome by treating all methods as potential 'starting points'.

A class's declared structure essentially consists of a set of instance variables — which represent a state template for instances of that class — a set of methods — which represent the functionality of those instances — and finally a set of type variables.

$$\begin{aligned} ClassBody \ :: \ \ Inst\_vars \ &: \ InstVar \xrightarrow{m} Type \\ Methods \ &: \ Selector \xrightarrow{m} Method \\ Type\_vars \ &: \ BoundedTypeVar^{*} \end{aligned}$$

As can be seen, the actual structure of the class's 'body' is more complex than the informal description above. The instance variables are represented by a mapping between the variable identifiers and the variables' types, and the methods are represented by a map between the method selectors and their implementations. Finally the type variables help maintain a level of flexibility in a system in which everything from variables to expressions must be assigned a type. The type variables may be used in the declaration of the instance variables and in the typing of the methods, and implicitly represent a range of types, so avoiding the need to use explicit class-based object types.

One byproduct of this is that the class can then be considered to be parameterised by its type variables. In order to accommodate this, the type variables must be recorded, not as a set, but as a sequence whose order is fixed.

The class's type variables are represented by a pair made up of the type variable itself, together with a declared 'bound'. The bound represents the set of possible types that the type variable may be called upon to represent. The precise structure and rôle of the bound, which is actually just a type, will be explained in §3.4, together with the significance of parameterising the class with type variables.

$$\mathit{BoundedTypeVar} = \mathit{TypeVar} \times \mathit{Bound}$$

$$\mathit{Bound} = \mathit{Type}$$

Methods come in two varieties, the user-defined method, which makes up the visible part of the language's functionality, and the primitive method, which remains in the background but which is ultimately responsible for getting things done.

$$\mathit{Method} = \mathit{UserMethod} \ \cup \ \mathit{PrimitiveMethod}$$

The user-defined method consists of the parameters, temporary variables and method body, which make up the basis of the method definition. In addition it also has a return type and a set of type variables. The parameters are typed, as are the temporary variables. The type variables are there for ostensibly the same reason as for the class body. The object returned by the method is the result of evaluating the method body. This removes the need for a special *return* expression.[3] As a result, implicit '↑self' expressions (return reference to self object) found in existing Smalltalk-80 implementations are assumed to have been be inserted explicitly.

$$\begin{array}{lrcl} \mathit{UserMethod} :: & \mathit{Params} & : & \mathit{TypedParam}^* \\ & \mathit{Temps} & : & \mathit{TempVar} \xrightarrow{m} \mathit{Type} \\ & \mathit{Return} & : & \mathit{Type} \\ & \mathit{Body} & : & \mathit{Expression} \\ & \mathit{Type\_vars} & : & \mathit{BoundedTypeVar}^* \end{array}$$

Both the parameters and the type variables are represented as sequences for the same reason, namely that their ordering is significant and must be fixed. The ordering of the parameters is clearly necessary to enable proper matching with the message arguments passed during a message send. The significance of the ordering of the type variables lies in considering the type of the method. In the same way that a class is parameterised by its type variables, if a method was considered to have a type, made up of its parameter and return types, this type would be parameterised by the method's type variables.

The primitive method, within the current context, appears as a cut-down version of the user-defined method, with only the parameters and return type remaining. This can be seen as an abstracted view of the primitive method because it does not appear to record what action the primitive is supposed to perform. It can also be seen as a naïve view in that it does not make provision for the possible failure of the primitive. In this respect it moves away from Graver's model, which follows the Smalltalk-80 technique of executing user-defined Smalltalk code when faced with a primitive failure [Gra89], and instead moves towards the simplified assumption that error handling can be coded into the primitive itself [Wol88].

[3] The need for a special return expression comes from having return expressions within blocks. With respect to an abstract syntax, such a need can be removed simply by treating such blocks as special 'return' blocks.

$$\left.\begin{array}{r}\left.\begin{array}{r}\left.\begin{array}{r}\mathit{TempVar}\\ \mathit{InstVar}\end{array}\right\} = \mathit{AssVar}\\ \mathit{ParamId}\end{array}\right\} = \mathit{VarId}\\ \mathit{self}\end{array}\right\} = \mathit{ObjectName}$$

Figure 3.1: Object referent subgrouping

$$\begin{array}{lll} \mathit{PrimitiveMethod} :: & \mathit{Params} & : \mathit{TypedParam}^* \\ & \mathit{Return} & : \mathit{Type} \end{array}$$

Like the type variables, parameters come in the form of a pair, with one half representing the parameter's identity while the other represents its type.

$$\mathit{TypedParam} = \mathit{ParamId} \times \mathit{Type}$$

Naturally there is an implied constraint that in any one method's parameter sequence, a given parameter identity occurs only once.

The expressions which constitute the Smalltalk code itself make up only a handful of individual cases. There are the two 'active' expressions, the assignment and the message send, together with the more passive object creation ('new object' and 'literal') and object referencing ('object name') expressions. In addition there is the expression sequence to combine the other expressions.

$$\mathit{Expression} = \mathit{ExpressionList} \cup \mathit{Assignment} \cup \mathit{ObjectName} \cup \mathit{Message} \cup \mathit{NewObject} \cup \mathit{Literal}$$

$$\mathit{ExpressionList} = \mathit{Expression}^*$$

The assignment statement simply consists of an assignable variable — a temporary or instance variable — on the left-hand side, and an expression on the right-hand side. The result of evaluating the right-hand expression will then be assigned to the left-hand variable

$$\begin{array}{lll} \mathit{Assignment} :: & \mathit{LHS} & : \mathit{AssVar} \\ & \mathit{RHS} & : \mathit{Expression} \end{array}$$

The way in which the variables and parameters have been constructed allows subsets of object referents (*object names*) to be grouped together easily. Figure 3.1 shows how the subgroups are organised. In this way membership of a particular object name subgroup can be tested by simple set inclusion. The assignable variables have been grouped together as *AssVar*s:

$$\mathit{AssVar} = \mathit{TempVar} \cup \mathit{InstVar}$$

$$\mathit{TempVar} :: \mathit{Id}$$

$$\mathit{InstVar} :: \mathit{Id}$$

The 'variable' identifiers, made up of the proper variables together with the formal parameters, also form a group (*VarId*):

$$VarId = TempVar \cup InstVar \cup ParamId$$

$$ParamId :: Id$$

Finally all the object referents — variables, parameters and self — are grouped together within the 'object name' expression.

$$ObjectName = VarId \cup \{self\}$$

At this point it would be worth-while noting that all the identifiers — class names, variable names, method selectors etc. — have been defined in the same way, namely as a tagged instance of some non-specified domain *Id*. The reasoning behind this is to indicate that these names have some form of underlying textual representation which is common to all of them, but that by virtue of the way they have been constructed, or by the context of their occurrences, they can be distinguished and categorised according to the abstract syntax presented here.

The message-send expression consists of a receiver expression, a message/method selector, and a sequence of expressions constituting the arguments to be passed with the message send.

$$\begin{aligned} Message :: Rcvr &: Expression \\ Sel &: Selector \\ Args &: Expression^{*} \end{aligned}$$

The result of evaluating the receiver expression provides the object to which the message is to be sent. The selector can then be matched with the corresponding method provided by the receiving object. If the message send requires further arguments, these are provided by evaluating the sequence of argument expressions, in order. In the case of a unary message send, this sequence will, of course, be empty.

The creation of new objects is normally performed by the new primitive, invoked via a message send. In keeping with Graver, a separate expression will be used for this language, which will ultimately aid in the typing of such expressions. The new-object expression will create a new instance of the named class.

$$NewObject :: Class : ClassName$$

Finally, the generalised literal expression covers all acceptable literal expressions, returning a suitably initialised instance of the correct class. As a result of this 'catch all' approach the domain of the literal has not been specified.

$$\begin{aligned} Literal :: Value &: \ldots \\ Type &: Type \end{aligned}$$

In addition to the value of the literal there is also a field to record its type. This assumes that the literal is unambiguous enough to have a type inferred for it. The obvious case where this assumption might experience difficulties is when the literal is an array. For example, should the array

```
#(1 42 3)
```

be typed as an array of integers, an array of numbers or an array of anything?[4]

In this section we have looked at the syntax of the language to be analysed and typed. Throughout this section, references have been made to types, type variables and type variable bounds. Types have been associated with variables, parameters and method returns. In the next section the actual structure of these types will be introduced, and the use and significance of type variables will be explained.

3.4 Types, trees and environments

In this section we introduce the domain and environment over which the type inference takes place. As indicated earlier (§3.1) the inference mechanism works by interpreting the program code over a non-standard domain, in this case the domain of types. The structure of this domain is defined in the following section, §3.4.1.

The process of interpretation involves transforming the code in order to accommodate the new domain. The form of the new structure, the *type tree*, is defined in §3.4.2, and parallels the structure defined by the abstract syntax in §3.3.

In order to process the information gathered by the analysis and interpretation process, an environment is required. This type environment is defined in §3.4.3, and retains the same basic structure as used by Graver in his formal description of the type inference process.

The rest of this section is made up of various auxiliary functions. Section 3.4.4 defines two tests to see if a type tree has been reduced to a single type. Section 3.4.5 describes a set of general-purpose functions useful for interpreting a sequence of pairs as a sequence of maplets. These functions are necessary to help manipulate the typed parameters (*TypedParam*, page 30) and bounded type variables (*BoundedTypeVar*, page 29) introduced as part of the abstract syntax.

3.4.1 *Type*

Central to Graver's type inference system is the domain of types. This is what initially defines the level of detail that the system can resolve. However that is not the full picture, and, as will be shown, what ultimately defines the resolving power of the system is how this type information is treated. This issue can be seen as resulting from the traditional "one variable — one type" model, and will be discussed in §3.8.

First, it is necessary to understand what constitutes a type. A type can be any of the following:

---

[4]The answer is, it would seem, to make it an array of P, where P is a type variable. The typing of the array is completed by including an initial constraint on P, namely

$$Integer \sqsubseteq P$$

an object type: This is the basic building block of the type, and represents the finest level of detail. An object type is associated with a single class, and is parameterised by that class's type variables.

a type variable: Existential bounded polymorphism is achieved through the use of type variables to implicitly refer to a type, or set of types. It is the use of type variables which ultimately gives the type system its flexibility.

a union type: This, in essence, is a set made up of object types and type variables. It is a type constructor rather than a type in its own right, but, together with the type variable, it is essential for the typing of polymorphic expressions.

The formal definition follows:

$$Type = ObjectType \cup TypeVar \cup UnionType \cup \{\textsc{Self}, \textsc{Anything}, \textsc{Nothing}\}$$

The constants, SELF, ANYTHING and NOTHING are there for convenience. SELF represents, for the purposes of typing the variables and methods of a class, the object type associated with that class. ANYTHING and NOTHING represent, respectively, the union type ranging over all object types (one for each class), and the empty union type. The use of these constants is merely as an aid to the programmer, but their existence is essential because they represent the existence of a locally-accessible self type, and an initial (default) upper and lower bound for the type variables.

The object type consists of a class name, the class from which it has been derived, and a sequence of type parameters, derived from the type variables of the class in question.

$$\begin{array}{lrcl} ObjectType :: & Class & : & ClassName \\ & Params & : & Type^* \end{array}$$

In the general case, the parameters will consist of a sequence of type variables. In specific instances, though, some or all of the type variables may be replaced by single object types or union types, to produce a specialised version of that object type. For example, if the general type is

Array of: X

where X is a type variable, a specialised case of that type might be

Array of: Integer

or even

Array of: <Array of: Y>

In a specialised case such as this, when the object type is applied back to its originating class, for example as the type of the receiver of a message send, the type parameters will be instantiated in place of the corresponding type variables throughout that class. The effect of this would be to produce a specialised version of that class, as well propagating case-specific type information.

This will become clear when the interpretation/reduction process is described (§3.6, and in particular §3.6.2).

Like the variables and parameters of the language syntax, type variables are made up of tagged instances of the domain *Id*. Again, the reasoning behind this is that type variables have an underlying representation which is common to the syntax of the language, but are distinguishable from other identifiers by the context in which they are used.

TypeVar :: *Id*

By itself, the type variable appears to have no bounds. In fact it does have implicit lower and upper bounds, respectively NOTHING and ANYTHING. In the case of the bounded type variable introduced in §3.3 (page 29), the 'bound' part represents an upper bound declared by the user. Such a bound, normally in the form of a union type, will represent the largest set of object types over which the type variable may range. If no bound is provided, the default upper bound of ANYTHING is used.

The real bounds of the type variable are determined through the inference process by collecting together inclusion relations. The inclusion relation will be described in more detail shortly, but briefly it represents a relationship between two types, either of which may be a type variable. The relation is of the form:

$$type_1 \sqsubseteq type_2$$

This is interpreted as saying that $type_1$ has a smaller set of possible types than $type_2$, or the two types are equal. In effect, the declared bound of a bounded type variable implies the following *programmer imposed* relation must hold:

$$typevar \sqsubseteq bound$$

In contrast, the lower bound represents a type, or set of types, inferred from the code. As more information is derived, this bound will grow to include all the types associated with the corresponding type variable during its lifetime:

$$inferred\_types \sqsubseteq typevar$$

It follows that at no time must the following relation hold:

$$t_{upper} \sqsubseteq t_{lower}$$

The union type is, in principle, just a set of types. It is used to represent the set of possible types which may occur, for example when considering the types of the result of a message send.

UnionType = *Type*-set

When considering such a message send, the result of a given method invocation may be handled by a single object type, but when all possible method invocations are taken into account, the result will be taken as the union of all such individual results.

Subtyping, which is central to the type system, and the type inference system in particular, is defined by the $\sqsubseteq$ relation (the inclusion relation). The rules for the $\sqsubseteq$ relation are as follows. For relations involving a type variable, the relation is only valid if there exists a substitution for that type variable such that the relation still holds. Determining such a mapping from type variables to types is the task of Graver's type inference *algorithm*, described in detail in his thesis.

For relations involving union types, with the observation that non-union types can be coerced into singleton union types for the purposes of comparison, the situation is handled by a combination of set inclusion and 'pairing off'. For example the relation:

$$\{t_{x_1}, \ldots, t_{x_n}\} \sqsubseteq \{t_{y_1}, \ldots, t_{y_m}\}$$

is only valid if, for each type on the left-hand side, there is a type on the right-hand side such that the inclusion relations holds between them. In other words

$$\forall\ i \in 1 \ldots n \cdot \exists\ j \in 1 \ldots m \cdot t_{x_i} \sqsubseteq t_{y_j}$$

must hold. 'Pairing off' occurs for pairs of unbounded type variables, one from each side of the relationship, which are then able to cancel each other out.

Although the union type may be thought of as the most general case, it is the object-type case which forms the basis of the subtype/inclusion relation. All other combinations resolve down, through the union-type case, to this final case, namely that involving two object types. Put simply, the relation involving two object types

$$\textit{mk-ObjectType}(\textit{class}_x, \textit{params}_x) \sqsubseteq \textit{mk-ObjectType}(\textit{class}_y, \textit{params}_y)$$

where $\textit{params}_x$ and $\textit{params}_y$ are sequences of types of equal length, holds only if

$$\textit{class}_x = \textit{class}_y$$

and

$$\forall\ i \in 1 \ldots \mathsf{len}\, \textit{params}_x \cdot \textit{params}_x[i] = \textit{params}_y[i]$$

hold.

The reasoning behind this is based on the fact that the language being typed is imperative, not functional, and so has to be wary of assignments to objects passed by reference as arguments. As such, for example, a message argument which is typed as an array containing integers (type Array of: Integer) does not want to get involved with a formal parameter typed for an array of integers and characters (type Array of: <Integers + Character>) because it might end up with an unwanted character.

3.4.2 *TypeTree*

As part of the type inference process, the program code must be 'transformed' in order to accommodate the new domain of interpretation. The new structure is defined by the type tree,

$$\left.\begin{array}{r}\text{Object name}\\ \text{New object}\\ \text{Literal}\end{array}\right\} \rightarrow \textit{Type}$$

$$\begin{array}{rcl}\text{Expression list} & \rightarrow & \textit{TypeTreeList}\\ \text{Assignment} & \rightarrow & \textit{AssignmentType}\\ \text{Message send} & \rightarrow & \textit{MessageType}\end{array}$$

Figure 3.2: Transformation correspondence between expressions and type trees.

which closely parallels the parse tree structure of the language's abstract syntax.

The type tree is made up of four basic elements: the type, as defined above; the type-tree list, which corresponds to the expression list; the assignment type; and the message type. The simple type initially covers all object referencing expressions, from object names, through to object creation and literal expressions. Ultimately the aim is to reduce a given type-tree expression to a type. Figure 3.2 shows the correspondence between the various expression structures and type-tree structures.

$$\textit{TypeTree} = \textit{Type} \ \cup \ \textit{TypeTreeList} \ \cup \ \textit{AssignmentType} \ \cup \ \textit{MessageType}$$

The type-tree list is made up of a sequence of type trees, one for each expression in the corresponding expression list.

$$\textit{TypeTreeList} = \textit{TypeTree}^{*}$$

The assignment type retains the structure of the corresponding assignment expression, but with the left-hand variable and the right-hand expression replaced by type trees. In the case of the left-hand variable the type tree will in fact be a type.

$$\begin{array}{lcrcl}\textit{AssignmentType} & :: & \textit{LHS} & : & \textit{Type}\\ & & \textit{RHS} & : & \textit{TypeTree}\end{array}$$

As with the assignment type, the message type retains the structure of the message send expression, with the receiver and arguments replaced by type trees. The message selector, naturally, remains unchanged.

$$\begin{array}{lcrcl}\textit{MessageType} & :: & \textit{Rcvr} & : & \textit{TypeTree}\\ & & \textit{Sel} & : & \textit{Selector}\\ & & \textit{Args} & : & \textit{TypeTree}^{*}\end{array}$$

The actual process of transforming the parse trees, constructed from the abstract syntax, into type trees, constructed from the definitions above, will be defined and described in §3.5.2.

3.4.3 The Type environment

The 'raw' result of the type inference process is a simple map, taking a class name and a method selector, and returning either a method type or a 'no-def' indicator.

$$\textit{MethodTypeMap} = (\textit{ClassName} \times \textit{Selector}) \xrightarrow{m} (\textit{MethodType} \cup \{\textsc{no-def}\})$$

The purpose of the 'no-def' indicator is to allow for those methods which cannot be typed correctly. In the case of a language utilising inheritance, the 'no-def' indicator would be useful for the typing of methods provided by an abstract class.[5] Such methods may not possess a consistent typing in their originating class, but may be typable in those classes which inherit them.

The method type is the type equivalent of the method, in the same sense that the type tree was the type equivalent to the expression. Like the type-tree components of §3.4.2 it has the same basic structure as the user-defined method (page 29). The key elements are the self, parameter and return types. These are equivalent to the implicit receiver, the typed parameters and the return type of method definition. Replacing the method body is the type tree.

$$\begin{array}{lrcl} \textit{MethodType} :: & \textit{Self} & : & \textit{Type} \\ & \textit{Params} & : & \textit{Type}^* \\ & \textit{Return} & : & \textit{Type} \\ & \textit{Type\_env} & : & \textit{TypeEnv} \\ & \textit{Type\_tree} & : & \textit{TypeTree} \end{array}$$

In effect the method type represents the result of transforming a method from the syntactic domain into the domain of types. Those items which appear to be missing, the temporary and type variables, have been moved into the type environment, *TypeEnv*, and it is in this structure that most of the important information is held.

There is a copy of the type environment for each method being typed. The *TypeEnv* holds the type information extracted by the inference system, together with enough information to get back from the type domain to the state of a 'typed program'. To this end, the type environment records the type of the receiver, as well as a mapping from variables to their associated types. In addition it also records the type variables used in the typing of that method, together with their current 'estimates'. Before any information has been extracted from the code, the 'estimate' of a type variable will be either ANYTHING for those type variables without a declared upper bound, or the declared bound of that type variable. The final two elements are the method type map, which provides access to other method types for the purpose of typing message sends, and the set of inclusion relations, which provides the key to the whole type inference procedure. Throughout the rest of this chapter, instances of *TypeEnv* will be referred to by the greek letter τ when being used as a parameter.

[5] An abstract class is a class designed to be used only as a template for other classes to utilise. Classes which inherit from an abstract class are able to use the basic functionality provided by that class, but are required to 'fill in the blanks' before they are complete enough to be capable of having fully functional instances. As a consequence the abstract class does not require implementing to the extent of allowing working instances to be created.

$$\begin{array}{rrcl} \mathit{TypeEnv} :: & \mathit{Self\_type} & : & \mathit{Type} \\ & \mathit{Var\_types} & : & \mathit{VarId} \xrightarrow{m} \mathit{Type} \\ & \mathit{Type\_vars} & : & \mathit{TypeVar} \xrightarrow{m} \mathit{Type} \\ & \mathit{Method\_types} & : & \mathit{MethodTypeMap} \\ & \mathit{Incl\_relations} & : & \mathit{Incl\_rel}\text{-set} \end{array}$$

The method type map stored in the type environment is actually a copy of the map as it was at the end of the previous 'iteration'. This provides a stable way of accessing other methods during message sends, ensuring that a given method appears in the same state of typing to all other methods irrespective of the order in which the methods are processed.

The final element, the set of inclusion relations, represents the raw type information used by the type inference process. As the transformed code is 'interpreted' (§3.2), inclusion relations arise which must be checked against each other for consistency. It is these checks for consistency which provide the type information returned by the type inference system.

As has been mentioned earlier, the inclusion relation is of the form

$$\mathit{smaller} \sqsubseteq \mathit{greater}$$

where the set of object types represented directly (by a union type) or indirectly (by a type variable) by the *smaller* type is smaller or equal to the the set represented by *greater*. When dealing with type variables, this inclusion relation can also be thought of as a constraint relation, because it constrains the set of object types over which the variable may range. The inclusion relation is represented by

$$\begin{array}{rrcl} \mathit{Incl\_rel} :: & \mathit{Smaller} & : & \mathit{Type} \\ & \mathit{Greater} & : & \mathit{Type} \end{array}$$

where the *Smaller* and *Greater* fields correspond to the *smaller* and *greater* types in the example above.

As new relations are derived and recorded, the act of recording must also ensure that adding each new relation preserves the consistency of the whole set. This is achieved by performing a transitive closure on the relations, in other words:

$$\mathit{type}_1 \sqsubseteq \mathit{type}_2 \wedge \mathit{type}_2 \sqsubseteq \mathit{type}_3 \;\Rightarrow\; \mathit{type}_1 \sqsubseteq \mathit{type}_3$$

and then showing that a suitable substitution of object types exists for the type variables while maintaining consistency. The precise details of this algorithm is outside the scope of this thesis. However, the interested reader may wish to consult [Gra89] where the original formal definition may be found.

The manner in which these relations are derived has already been described informally (§3.2). A more precise description of how these inclusion relations are derived will be found as each relevant expression is encountered in the formal description which follows.

3.4.4 Auxiliary type functions

The goal of the type inference mechanism when 'interpreting' a transformed expression is to reduce it from a type tree to a single type — that is, an object type, a type variable or a union type (§3.6.2). To this end a test function, *isType*, is provided which will determine whether this has occurred.

$$isType : TypeTree \rightarrow \mathbb{B}$$
$$isType(type\_tree) \quad \triangleq \quad type\_tree \in Type$$

In addition a test function is also provided for sequences of type trees, in order to handle the results of argument lists and expression sequences.

$$isTypeSeq : TypeTreeList \rightarrow \mathbb{B}$$
$$isTypeSeq(type\_tree\_seq) \quad \triangleq$$
$$\forall\ i \in\ 1 \ldots \mathsf{len}\, type\_tree\_seq \cdot isType(type\_tree\_seq[i])$$

3.4.5 Miscellaneous auxiliary functions

The majority of the functions described in this section are designed to help manipulate a sequences of pairs as if each pair was a maplet. In other words, interpreting the pair

$$(a, b)$$

as the maplet:

$$\{a \mapsto b\}$$

The last function is used to convert a set into a sequence. The main use of this is to aid the manipulation of certain structures, and to aid the general readability of the specifications which follow. While only the function signatures are presented in this section, their definitions may be found in Appendix A.

The first pair of functions simply extract the first and second elements from a pair. These functions will later be used to extract the domain element (*fst*) and the range element (*snd*) from a pair posing as a maplet.

$$fst : X \times Y \rightarrow X$$
$$snd : X \times Y \rightarrow Y$$

The following function takes a sequence of pairs and returns the domain elements from each maplet. The domain elements are returned as a sequence, maintaining the same ordering as found in the original sequence.

$$dom\_seq : (X \times Y)^* \rightarrow X^*$$

The next function performs the same action as the previous one, only this time the sequence extracted is composed of the range elements of the original sequence. Once again the original order is preserved.

$$rng\_seq : (X \times Y)^* \rightarrow Y^*$$

This next function takes a sequence of pairs and converts it into a normal map. Each pair in the sequence is converted into a proper maplet, and the result added to the map. The major effect of this function is that the ordering, present in the sequence, is lost in the resulting map. Use of this function carries the assumption that the first items from each pair-element form a non-duplicating domain set.

$$seq\_as\_map : (X \times Y)^* \rightarrow X \xrightarrow{m} Y$$

The final function in this section is a simple conversion function. It takes a set and converts it into a sequence. The main reason for using such a function lies in the respective merits of the two structures. Sets are relatively easy to construct over an unordered domain, but when it comes to iterative manipulation, sequences are easier to process.

$$set\_to\_seq : X\text{-set} \rightarrow X^*$$

Now that both the language and the type system have been introduced, the next section will commence the description of the type inference process.

3.5 Resolution and transformation

The mechanism of type inference can be thought of as consisting of a sequence of processes. The key processes, briefly mentioned in the introduction to this chapter (page 24), are *resolution*, *transformation* and *reduction*.

resolution: In the simple language being dealt with in this thesis, this process amounts to instantiating implicit references to the type of self (SELF) with the correct object type. In a language utilising inheritance this process would involve copying down inherited methods and resolving references to self.

transformation: This is the process of converting the code being analysed into a form suitable for interpretation over the domain of types. This action, of converting 'parse trees' into 'type trees', also provides an initial typing for the code which can be refined through successive iterations of the third key process, *reduction*, to produce a final typing.

reduction: This, the third key process, performs the main work of the type inference system. By performing abstract interpretation over the transformed code, it is able to extract, as well as propagate, type information. The main aim of this process is to be able to reduce a given expression, in the form of a type tree, to the type of the result that that expression would return if evaluated. It is this process which constitutes abstract interpretation, and it is the information extracted during this process which provides the basis for the typing of the code.

This section will deal with the initial starting point of the type inference system, together with the resolution (§3.5.1) and transformation (§3.5.2) of the code being analysed.

To begin with the 'untyped' program is assumed to be in the proper form. This involves linking variables, parameters and return types with their declared types. In general these types will be type variables, which will act as 'hooks' onto which the inferred type information may be attached. Some of these declarations may be achieved by the inference system by automatically allocating type variables for method return types, for example. In addition to declaring the types of variables and methods using type variables, some declarations may involve explicit object and union types.

With the necessary 'hooks' in place, the structure of the code will correspond to the abstract syntax defined in §3.3. The following function sets the type inference process in motion.

$$Type\_program : Program \rightarrow MethodTypeMap$$

$$Type\_program\,[\![program]\!] \quad \triangleq$$
$$\quad \mathsf{let}\ initial\_typing = \bigcup\{instantiate\_self(class, program) \mid class \in \mathsf{dom}\,program\}$$
$$\quad\quad \mathsf{in}$$
$$\quad iterate(initial\_typing)$$

The initial typing is performed by the *instantiate_self* function, which performs both the resolution of the self type (initially assigned the special type constant SELF), and the transformation of the code. The result of applying *instantiate_self* to a particular class is a small method-type map, covering those methods implemented by that class. The initial typing of the whole program is obtained by taking the distributed union over the individual method-type maps returned by each class in the program.[6]

After the initial typing has been constructed, the main work of the type inference system is carried out under the control of the *iterate* function. This function invokes the reduction process repeatedly until a steady state is achieved, at which point the *iterate* function terminates and the inference process ends. The result at this point is a final method-type map, and from this, type information may be extracted and instantiated back into each class, variable and method to yield a fully typed program.

The following two subsections describe the processes of resolution, performed initially by *instant_self*, and transformation, performed initially by *type_method* (page 44). Both these processes are invoked via *instantiate_self*.

[6]The distributed union over a set of maps used in this function is assumed to have the obvious definition, treating each map as a set of domain/range maplet pairs. The distributed union can be defined in this case because each set of domain values from each application of *instantiate_self* will be disjoint from all other resulting domain sets. This arises specifically because each application of *instantiate_self* deals with a different class, and hence each separate map will have a unique *class* to make up its set of (*class, selector*) domain pairs.

3.5.1 Instantiation and initial typing

For a given call of *instantiate_self* on a given class, the function first calls *instant_self*, which returns both the resolved class definition (body) and the instantiated object type. The resolved class will have had all references to the special type constant SELF replaced by the correct object type. With this information *instantiate_self* then invokes *type_method* for each method implemented by that class. The resulting structure is a method-type map which covers all the methods in that class.

$$
\begin{array}{l}
\textit{instantiate\_self} : \textit{ClassName} \times \textit{Program} \to \textit{MethodTypeMap} \\
\textit{instantiate\_self}(\textit{class}, \textit{program}) \quad \triangleq \\
\qquad \mathsf{let}\ (\textit{class\_body}, \textit{self\_type}) = \textit{instant\_self}(\textit{class}, \textit{program}(\textit{class})) \\
\qquad\quad \mathsf{in} \\
\qquad \{(\textit{class}, \textit{sel}) \mapsto \textit{type\_method}(\textit{class}, \textit{sel}, \textit{class\_body}, \textit{self\_type}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \textit{sel} \in \mathsf{dom}\, \textit{Methods}(\textit{class\_body})\}
\end{array}
$$

Instant_self starts by creating the object type corresponding to the class. The object type is derived from both the class name and the declared type variables of the class definition. Note that only the type variables are required, hence the need to use the *dom_seq* function (page 39) to separate them from their ‘bounds’. Once created, this object type, the new *self_type*, is instantiated throughout the class in place of SELF. The new type is instantiated in the types of variables, parameters and even type variables, where SELF may have been used within a declared bound.

$$
\begin{array}{l}
\textit{instant\_self} : \textit{ClassName} \times \textit{ClassBody} \to \textit{ClassBody} \times \textit{Type} \\
\textit{instant\_self}(\textit{class\_name}, \textit{class\_body}) \quad \triangleq \\
\qquad \mathsf{let}\ \textit{self\_type} = \textit{mk-ObjectType}(\textit{class\_name}, \textit{dom\_seq}(\textit{Type\_vars}(\textit{class\_body}))) \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{let}\ \textit{inst\_vars} = \textit{instant\_self\_in\_vars}(\textit{Inst\_vars}(\textit{class\_body}), \textit{self\_type}) \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{let}\ \textit{methods} = \textit{instant\_self\_in\_methods}(\textit{Methods}(\textit{class\_body}), \textit{self\_type}) \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{let}\ \textit{type\_vars} = \textit{instant\_self\_in\_tvs}(\textit{Type\_vars}(\textit{class\_body}), \textit{self\_type}) \\
\qquad\quad \mathsf{in} \\
\qquad (\textit{mk-ClassBody}(\textit{inst\_vars}, \textit{methods}, \textit{type\_vars}), \textit{self\_type})
\end{array}
$$

The definitions of the various instantiating functions have been omitted from this chapter for the sake of brevity. However, their respective signatures follow, while their definitions may be found in Appendix A.

$instant\_self\_in\_vars : (AssVar \xrightarrow{m} Type) \times Type \rightarrow AssVar \xrightarrow{m} Type$
$instant\_self\_in\_methods : (Selector \xrightarrow{m} Method) \times Type \rightarrow Selector \xrightarrow{m} Method$
$instant\_self\_in\_tvs : (TypeVar \times Bound)^* \times Type \rightarrow (TypeVar \times Bound)^*$

As each of the above functions encounters a type it calls the function *subst_self* to perform the actual substitution. *Subst_self* takes the type encountered, together with the *self_type* to be substituted into it, and returns a freshly substituted type, ready to be placed back where the original was found. This function is defined by cases.

$subst\_self : Type \times Type \rightarrow Type$

The first three versions handle the basic types. For the object type it is necessary to go through the parameters of the object type. It is worth repeating at this point that although the *self_type* extracted from the class definition uses type variables exclusively for its parameters, it is valid to have an object type or union type in place of some, or all of the parameters.

$$subst\_self(mk\text{-}ObjectType(class, type\_vars), self\_type) \triangleq$$
$$mk\text{-}ObjectType(class, subst\_self\_in\_seq(type\_vars, self\_type))$$

The type variable is left as it is.

$$subst\_self(mk\text{-}TypeVar(id), self\_type) \triangleq mk\text{-}TypeVar(id)$$

The union type requires each constituent type to be dealt with separately.

$$subst\_self(type\_set, self\_type) \triangleq \qquad \text{for } type\_set \in UnionType$$
$$\{subst\_self(type, self\_type) \mid type \in type\_set\}$$

The rest of the cases deal with the special identifiers. SELF is the only one which results in a change, replacing the special identifier with the new *self_type*.

$$subst\_self(\textsc{Self}, self\_type) \triangleq self\_type$$

Both ANYTHING and NOTHING result in no change.

$$subst\_self(\textsc{Anything}, self\_type) \triangleq \textsc{Anything}$$

$$subst\_self(\textsc{Nothing}, self\_type) \triangleq \textsc{Nothing}$$

Subst_self_in_seq takes a sequence of types, in particular from the parameters of an object type, and invokes *subst_self* on each type in that sequence. As each type is dealt with, the result is concatenated back together to preserve their order.

$subst\_self\_in\_seq : Type^* \times Type \rightarrow Type^*$

$subst\_self\_in\_seq(type\_seq, self\_type) \quad \triangleq$

  if $type\_seq = [\,]$

  then $[\,]$

  else $[subst\_self(\mathsf{hd}\, type\_seq, self\_type)]$

  $\frown$

    $subst\_self\_in\_seq(\mathsf{tl}\, type\_seq, self\_type)$

Type_method creates the initial typing of the method passed to it. This involves extracting the various type elements for that method (parameter types, return types, variable types, etc.), as well as creating an initial type environment. This is the first stage of the transformation process, dealing with the overall structure of the method. The actual transformation of the method code is performed by the T function defined in the following section, §3.5.2.

Type_method distinguishes between user-defined methods and primitive methods, handing control over to their respective specialised functions, *type_user_method* and *type_prim_method*.

$type\_method : ClassName \times Selector \times ClassBody \times Type \rightarrow MethodType$

$type\_method(class, sel, class\_body, self\_type) \quad \triangleq$

  if $Methods(class\_body)(sel) \in UserMethod$

  then $type\_user\_method(class, sel, class\_body, self\_type)$

  else $type\_prim\_method(class, sel, class\_body, self\_type)$

Type_user_method starts by establishing the structure of the method by extracting all the components from which it is constructed. The next task is to arrange all the components needed for the type environment. Each type environment is unique for a given method, and is constructed from information derived from the defining class (the instance variables and class type variables) and from the method itself (the parameters and temporary variables). This information is stored in the first three fields of the type environment, the self type (*self_type*), the variable and parameter types (*var_types*) and the type variables (*type_vars*). In principle, these entries do not change throughout the type inference process. Instead they act as 'hooks', in the form of type variables, from which further type information may be 'hung' in the inclusion relations field. However their values do change temporarily during the process known as 'case analysis', in particular the type variable map (page 53).

The rest of the type environment is made up of the method type map, and the set of inclusion relations. The method type map is left empty to begin with, until all the methods have been given an initial type. The set of inclusion relations is the structure in which all the type information derived from the abstract interpretation process is stored. The initial contents of the set are derived from the declared bounds of the method and class type variables. The relationship between a type variable and its declared upper bound yields the following inclusion relation:

$$type\_variable \sqsubseteq bound$$

Once the type environment has been constructed it is possible to construct the method type. The majority of this structure is taken up by the 'signature' of the method — the receiver (*self_type*), parameter and return types — together with that method's type environment. The crucial final part is taken up by the transformation of the method body. This process of transformation is performed by the *T* function, described in §3.5.2.

$$type\_user\_method : ClassName \times Selector \times ClassBody \times Type \to MethodType$$

$$type\_user\_method(class, sel, class\_body, self\_type) \quad \triangleq$$

$$\begin{array}{l}
\textsf{let } mk\text{-}UserMethod(params, temps, return, body, m_type_vars) \\
\qquad\qquad\qquad\qquad\qquad\qquad = Methods(class_body)(sel) \\
\quad \textsf{in} \\
\textsf{let } var_types = Inst_vars(class_body) \cup seq_as_map(params) \cup temps \\
\quad \textsf{in} \\
\textsf{let } type_vars = seq_as_map(Type_vars(class_body)) \cup seq_as_map(m_type_vars) \\
\quad \textsf{in} \\
\textsf{let } incl_rels = \{mk\text{-}Incl_rel(tv, bound) \mid \{tv \mapsto bound\} \in type_vars\} \\
\quad \textsf{in} \\
\textsf{let } \tau = mk\text{-}TypeEnv(self_type, var_types, type_vars, \{\,\}, incl_rels) \\
\quad \textsf{in} \\
\textsf{let } param_types = rng_seq(params) \\
\quad \textsf{in} \\
\textsf{let } return_type = return \\
\quad \textsf{in} \\
\textsf{let } ttree = T[\![body]\!]\,\tau \\
\quad \textsf{in} \\
mk\text{-}MethodType(self_type, param_types, return, \tau, ttree)
\end{array}$$

Type_prim_method is almost identical to *type_user_method* in the order and actions it performs. The initial type environment is constructed from the instance variables and parameters related to that method, and the type variables of its defining class. Again the method type map is left empty, until the whole program has been initially typed, and the inclusion relation set is initialised with the declared bounds of the class type variables. The method type is constructed in the same way as for the user-defined method, but because the method has no 'body', the entry for the type tree is left empty. This act of leaving the transformed body empty ([ ] — the empty sequence) is used as a convention for indicating a totally typed method, where the type returned by invoking that method can be determined by propagating type information from the message send, through the parameters, and straight out to the return type.

$type\_prim\_method : ClassName \times Selector \times ClassBody \times Type \to MethodType$

$type\_prim\_method(class, sel, class\_body, self\_type) \quad \triangleq$
$\quad$ let $mk\text{-}PrimitiveMethod(params, return) = Methods(class\_body)(sel)$
$\quad\quad$ in
$\quad$ let $var\_types = Inst\_vars(class\_body) \cup seq\_as\_map(params)$
$\quad\quad$ in
$\quad$ let $type\_vars = seq\_as\_map(Type\_vars(class\_body))$
$\quad\quad$ in
$\quad$ let $incl\_rels = \{mk\text{-}Incl\_rel(tv, bound) \mid \{tv \mapsto bound\} \in type\_vars\}$
$\quad\quad$ in
$\quad$ let $\tau = mk\text{-}TypeEnv(self\_type, var\_types, type\_vars, \{\,\}, incl\_rels)$
$\quad\quad$ in
$\quad$ let $param\_types = rng\_seq(params)$
$\quad\quad$ in
$\quad$ let $return\_type = return$
$\quad\quad$ in
$\quad$ $mk\text{-}MethodType(self\_type, param\_types, return\_type, \tau, [\,])$

3.5.2 Transformation — from parse trees to type trees

The T function transforms 'parse tree' expressions into their equivalent 'type-tree' expressions. As was noted in §3.4.2, the type-tree expressions retain the same structure as their equivalent expressions defined by the syntax. It may be worthwhile referring to Figure 3.2 (page 36) when following the functions defined in this section. The order in which the cases are dealt with corresponds to both the ordering within of the definition of *TypeTree* (page 36) and the ordering in the figure.

The basic T function takes, as a starting point, a syntactic expression. In order to ensure that an expression is typed in the correct context, it is necessary to pass the relevant type environment for the method containing that expression. The result of the T function, as indicated earlier, is a 'type tree'.

$T : Expression \to TypeEnv \to TypeTree$

The first four cases deal with those expressions which transform to a single type (as opposed to a type tree). The first two cover object name expressions — references to self and other identifiers, variable or otherwise. The essential difference between these two cases is that for self the type is obtained from a separate *TypeEnv* field.

$T[\![self]\!]\,\tau \quad \triangleq \quad Self\_type(\tau)$

For all the other identifiers — variables and parameters — the type is stored in the general 'variable map' in the type environment.

$$T[\![id]\!]\,\tau \quad \triangleq \quad Var\_types(\tau)(id) \qquad\qquad \text{for } id \in VarId$$

The next two 'simple' cases, still returning a type, concern the two object creation expressions. First, the explicit object creation expression, which, when given a class name, creates an instance of that class. In transforming this expression it is necessary to construct an object type corresponding to the given class. This in turn requires extracting the type variables, minus their bounds, from that class — hence the use of the *dom_seq* function. With the type parameters extracted it is a simple step to create the required object type.

$$\begin{array}{l} T[\![NewObject(class)]\!]\,\tau \quad \triangleq \\ \quad \mathsf{let}\ type\_params = dom\_seq(Type\_vars(Program(\tau)(class))) \\ \quad\quad \mathsf{in} \\ \quad mk\text{-}ObjectType(class, type\_params) \end{array}$$

The final simple case is the literal expression. In the same way that the precise object type can be deduced for the object creation expression, it is assumed that the type of a literal may be statically inferred (see page 31). As a result it is merely necessary to extract the already-inferred type from the literal expression.

$$T[\![mk\text{-}Literal(val, type)]\!]\,\tau \quad \triangleq \quad type$$

The rest of the cases deal with expression constructs. For each of these there is a corresponding type-tree expression. The equivalent type-tree expression to the expression list is the type-tree list. Each element in the list is dealt with separately, and the result is concatenated together, preserving the original order.

$$\begin{array}{l} T[\![exprs]\!]\,\tau \quad \triangleq \qquad\qquad\qquad \text{for } exprs \in Expression^* \\ \quad \mathsf{if}\ exprs = [\,] \\ \quad \mathsf{then}\ [\,] \\ \quad \mathsf{else}\ [T[\![\mathsf{hd}\ exprs]\!]\,\tau] \frown T[\![\mathsf{tl}\ exprs]\!]\,\tau \end{array}$$

The assignment expression has, as its type-tree equivalent, the assignment type. The assignment type is constructed using the results obtained from transforming the left-hand and right-hand sides.

$$\begin{array}{l} T[\![mk\text{-}Assignment(lhs, rhs)]\!]\,\tau \quad \triangleq \\ \quad mk\text{-}AssignmentType(T[\![lhs]\!]\,\tau, T[\![rhs]\!]\,\tau) \end{array}$$

The final expression, the message send, with its type-tree equivalent, the message type, uses the same technique as the assignment expression. Each component of the original is transformed, and the results are used to construct the corresponding type-tree expression. The task is made slightly easier by noting that the message arguments are constructed from a sequence of expressions, and that this is the same form as the expression list. By invoking T on the argument sequence, this case is handled by the expression list case defined above, where the result is a type-tree list, equivalent to a sequence of type trees, which is the form required for constructing the message type.

$$
\begin{aligned}
&T[\![\mathit{mk\text{-}Message}(\mathit{rcvr}, \mathit{sel}, \mathit{args})]\!]\,\tau \quad \triangleq \\
&\qquad \mathsf{let}\ \mathit{rcvr\_type} = T[\![\mathit{rcvr}]\!]\,\tau \\
&\qquad\quad \mathsf{in} \\
&\qquad \mathsf{let}\ \mathit{arg\_types} = T[\![\mathit{args}]\!]\,\tau \\
&\qquad\quad \mathsf{in} \\
&\qquad \mathit{mk\text{-}MessageType}(\mathit{rcvr\_type}, \mathit{sel}, \mathit{arg\_types})
\end{aligned}
$$

This section has presented the transformation functions which convert the program code into a form suitable for interpretation over the domain of types. In the following section, the process of interpretation will be defined.

3.6 Iteration, reduction and type inference

Once the code has been suitably transformed, the process of interpretation, and typing, may commence. This is governed at the highest level by the *iterate* function, invoked by *Type_program* (page 41). This function will be defined in §3.6.1, but first a brief overview of the sequence in which the main inference functions are invoked will be given.

Iterate, defined recursively, invokes *infer_types* repeatedly until the method type map, the foundation structure of the whole inference process, achieves a stable state. *Infer_types* (page 49) controls the typing of each method, invoking *infer_method_type* on those methods requiring further typing. *Infer_method_type* (page 51) performs the low-level typing of an individual method, updating that method's type environment, invoking the *Reduce* function on the method body, and finally determining whether the reduction, and hence the typing, has been complete.

Finally, at the lowest level of all, the set of functions controlled from the *Reduce* function handle the typing of an individual expression. This group of functions, together with *Reduce* itself, are defined in §3.6.2 and §3.6.3.

3.6.1 Iteration

The process of type inference presented in this thesis is based on an iterative mechanism. During a single iteration, each method which still requires attention is processed. Once this has been performed, the result, in the form of a new method type map, is compared with the state of the type map as it was at the end of the previous iteration. If there has been a change, the process is repeated on the new map. If the state of the type map is unchanged then no further type information can be extracted and the inference process can terminate.

This mechanism is reflected in the definition of *iterate*. The function receives a method type map as its sole argument. This is either the result of the resolution and transformation process described earlier (the initial case), or the result of a previous iteration (the general case). *Infer_types* is invoked on this type map, and the result is recorded. The new type map is then compared with the old type map. If they are unchanged then the function can terminate, returning the final method type map. If a change has occurred, another iteration must be performed. This is achieved by invoking *iterate*, recursively, on the new method type map.

$$
\begin{array}{l}
\mathit{iterate} : \mathit{MethodTypeMap} \rightarrow \mathit{MethodTypeMap} \\
\mathit{iterate}(\mathit{method\_types}) \quad \triangleq \\
\quad \mathsf{let}\ \mathit{inferred\_method\_types} = \mathit{infer\_types}(\mathit{method\_types}) \\
\quad\quad \mathsf{in} \\
\quad \mathsf{if}\ \mathit{inferred\_method\_types} = \mathit{method\_types} \\
\quad \mathsf{then}\ \mathit{method\_types} \\
\quad \mathsf{else}\ \mathit{iterate}(\mathit{inferred\_method\_types})
\end{array}
$$

Infer_types controls which methods in the method type map go on to be processed. Each method is selected initially ($(\mathit{class}, \mathit{sel}) \in \mathsf{dom}\,\mathit{method\_types}$) but only those methods which still have something to type ($\mathit{method\_types}(\mathit{class}, \mathit{sel}) \neq$ NO-DEF) or a body to process ($\mathit{Type\_tree}(\mathit{method\_types}(\mathit{class}, \mathit{sel})) \neq [\,]$) are passed on to *infer_method_type*.

$$
\begin{array}{l}
\mathit{infer\_types} : \mathit{MethodTypeMap} \rightarrow \mathit{MethodTypeMap} \\
\mathit{infer\_types}(\mathit{method\_types}) \quad \triangleq \\
\quad \{(\mathit{class}, \mathit{sel}) \mapsto \mathit{infer\_method\_type}(\mathit{class}, \mathit{sel}, \mathit{method\_types}) \\
\qquad\qquad\qquad\qquad \mid (\mathit{class}, \mathit{sel}) \in \mathsf{dom}\,\mathit{method\_types} \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{method\_types}(\mathit{class}, \mathit{sel}) \neq \text{NO-DEF} \\
\qquad\qquad\qquad\qquad \wedge\ \mathit{Type\_tree}(\mathit{method\_types}(\mathit{class}, \mathit{sel})) \neq [\,]\}
\end{array}
$$

Infer_method_type deals with the low-level process of inferring an individual method type. The sequence of actions taken by this process on a given method is as follows. First the contents of the method's current type are extracted. The type environment is then updated to take account of the currently distributed method type map. This map is the result of the previous iteration, unless this is the first iteration, in which case it is the result of the transformation process.

With the correct type map in place, the next step is to interpret the transformed method body (*ttree*). This is carried out by invoking *Reduce* on the body in question. In the general case *Reduce* returns a reduced type tree and an updated type environment. In some cases, where the typing fails, *Reduce* returns a FAIL flag. In most cases this fail condition arises due to a contradictory state being reached when evaluating the inclusion relations of the type environment. Such failures have not been handled explicitly by the function definitions which follow, but are assumed to be handled implicitly by the relevant functions which deal directly with the set of inclusion relations, for example *add_incl* (page 66). The only other way a fail condition can arise is by a mismatch between a message send and a receiver — the 'does-not-understand' error. Again this problem is handled implicitly by the *Reduce_send* function (page 58). In all cases where *Reduce* returns FAIL, the method type is replaced by the NO-DEF tag.

Returning to the general *Reduce* case, it is now necessary to determine if the reduction of the method body has been complete. This will have occurred if the type tree returned by the *Reduce* function is a type rather than a just a type tree. If this is the case, it is only necessary to consult the parameter and return types in order to determine the actual return type for a given invocation. In such an instance, with the method body reduced to a type, this type must be linked to the declared return type of the method. This is performed by the *add_incl* function, described on page 66, which adds the following type inclusion to the set of type inclusions for that method:

$$\mathit{ttree}' \sqsubseteq \mathit{return}$$

where ttree' is the reduced type of the method body, and *return* is the declared return type of the method. There is a possible fail condition here which occurs when the inferred type (ttree') does not agree with the declared type (*return*). This situation is handled by replacing the method type with the NO-DEF tag.

Assuming that the inferred return type has been successfully handled, the updated method type may be returned as the result of the inference process. However, before this can take place, it must have its reduced method body replaced by an empty sequence — effectively an empty sequence of type trees — to indicate its successful and complete typing. Finally, in the case where the method body has *not* been completely reduced, the method type is returned with the partially reduced type tree replacing the previous state, together with an updated type environment.

$$\mathit{infer\_method\_type} : \mathit{ClassName} \times \mathit{Selector} \times \mathit{MethodTypeMap} \to (\mathit{MethodType} \cup \text{NO-DEF})$$

$\mathit{infer\_method\_type}(\mathit{class}, \mathit{sel}, \mathit{method\_types}) \quad \triangleq$

let $\mathit{mk\text{-}MethodType}(\mathit{self}, \mathit{params}, \mathit{return}, \tau, \mathit{ttree}) = \mathit{method\_types}(\mathit{class}, \mathit{sel})$
in
let $\tau' = \mu(\tau, \mathit{Method\_types} \mapsto \mathit{method\_types})$
in
let $\mathit{reduced\_method\_type} = \mathit{Reduce}\,[\![\mathit{ttree}]\!]\,\tau'$
in
if $\mathit{reduced\_method\_type} = \text{FAIL}$
then NO-DEF
else let $(\mathit{ttree}', \tau'') = \mathit{reduced\_method\_type}$
in
if $\mathit{isType}(\mathit{ttree}')$
then let $\tau''' = \mathit{add\_incl}(\mathit{ttree}', \mathit{return}, \tau'')$
in
if $\tau''' = \text{FAIL}$
then NO-DEF
else $\mathit{mk\text{-}MethodType}(\mathit{self}, \mathit{params}, \mathit{return}, \tau''', [\,])$
else $\mathit{mk\text{-}MethodType}(\mathit{self}, \mathit{params}, \mathit{return}, \tau'', \mathit{ttree}')$

It is interesting to note that none of the main types constituting the method type — self, parameters and return types — are altered during the inference process. In fact none of the types in either the method type or the type environment are altered directly, except during the breaking up and reconstitution of union types during 'case analysis' (page 53). Instead, all the type information extracted by the type inference process is contained in the inclusion relations. Inferred types are determined on an on-demand basis using the *evaluate_in* function, described on page 67, and transferred, when necessary, to the type variable map. This technique can be seen in action, for example, when determining the receiver type of a message send (*Reduce_send* when applied to a type variable, page 60).

3.6.2 Reduction — from type trees to types

After the initialisation, transformation and the general housekeeping work of the iteration function, we arrive at the heart of the type inference mechanism: the reduction functions. This set of functions perform the low level abstract interpretation on which the inference system is based. After the transformation functions, it is these reduction functions which are ultimately responsible for the way in which the type information is extracted from the code.

Like most of the system described so far, the underlying principle of the reduction functions is derived directly from Graver's formal description in [Gra89]. However, whereas most of the basic structure has been preserved throughout the previous stages, the actual process of the reduction functions has been revised. This revision has in part been done to rectify a basic problem is Graver's definition — that case analysis is only performed on sequences of expressions. This problem is discussed later, within the context of the function which rectifies it, namely *Reduce_expr*.

The *Reduce* function represents the entry point into the family of reduction functions. It also fixes the basic interface of the functions. *Reduce* takes as its arguments the type tree (expression) to be reduced and the type environment (context) within which the expression is to be reduced. In return, it yields the reduced expression and a type environment, updated with the type information derived during the reduction process. Alternatively, in accordance with the various explicit and implicit consistency tests performed, if, for example the reduction fails because of a message selector mismatch ('does-not-understand' error), the function returns FAIL.

$$Reduce : TypeTree \to TypeEnv \to ((TypeTree \times TypeEnv) \cup \{\text{FAIL}\})$$

Two basic cases are distinguished at this point in the process. The first case handles a sequence of type trees, calling *Reduce_expr_list* on the type-tree list. On return, if the sequence of type-trees has been successfully reduced to a sequence of types, the reduced sequence is replaced by its last element. This constitutes the fully reduced form of the expression sequence. If the reduction has not been complete, the partially reduced sequence is returned. In both cases an updated type environment is also returned.[7]

$$
\begin{array}{ll}
Reduce\,[\![ttrees]\!]\,\tau \;\;\triangleq & \text{for } ttrees \in TypeTreeList \\
\quad \mathsf{let}\ (ttrees', \tau') = Reduce\_expr\_list(ttrees, \tau) & \\
\quad\quad \mathsf{in} & \\
\quad \mathsf{if}\ isTypeSeq(ttrees') & \\
\quad \mathsf{then}\ (ttrees'[\mathsf{len}\ ttrees'], \tau') & \\
\quad \mathsf{else}\ (ttrees', \tau') &
\end{array}
$$

The second case handles all single expressions, and calls *Reduce_expr*, the single expression companion of the reduction function called by the first case.

$$
\begin{array}{ll}
Reduce\,[\![ttree]\!]\,\tau \;\;\triangleq & \text{for } ttree \notin TypeTreeList \\
\quad Reduce\_expr(ttree, \tau) &
\end{array}
$$

Reduce_expr and *Reduce_expr_list* form a pair of functions, with the latter providing a recursive structure to handle sequences while passing each individual expression over to *Reduce_expr* to perform the real work. *Reduce_expr_list* recurses over a sequence of type trees, calling

[7] The use of 'Strachey' brackets ($[\![\;]\!]$) for the *Reduce* function reflects the similarities between the analysis of the original abstract syntax and the analysis of the transformed type trees. Both domains are, in effect, different kinds of syntax.

Reduce_expr on each one, while propagating the type environment from the previous expression. The resulting reduced type trees are concatenated together, preserving their order, while the type environment returned is taken from the last expression dealt with.

$$\mathit{Reduce\_expr\_list} : \mathit{TypeTreeList} \times \mathit{TypeEnv} \to \mathit{TypeTreeList} \times \mathit{TypeEnv}$$

$\mathit{Reduce\_expr\_list}(\mathit{ttrees}, \tau) \quad \triangleq$
    if $\mathit{ttrees} = [\,]$
    then $([\,], \tau)$
    else let $(\mathit{ttree}', \tau') = \mathit{Reduce\_expr}(\mathsf{hd}\ \mathit{ttrees}, \tau)$
        in
      let $(\mathit{ttrees}', \tau'') = \mathit{Reduce\_expr\_list}(\mathsf{tl}\ \mathit{ttrees}, \tau')$
        in
      $([\mathit{ttree}']^\frown \mathit{ttrees}', \tau'')$

Reduce_expr handles individual expressions. It forms the central point from which all the lower functions — *R* and the message send reduction functions for example — are ultimately called. It also handles the case analysis mechanism, and, as a result, all expressions pass through this mechanism, unlike in Graver's definition, where the case analysis is restricted to expression sequences only ([Gra89], page 77).

The case analysis mechanism starts by creating 'cases' out of the type environment. From a single type environment, a set of type environments — cases — are created with the basic property that all the union types have been broken up into their constituent object types and type variables. The precise meaning of this is described in §3.6.3, but for now it is only necessary to accept that for a given variable associated with a union type in the original type environment, there is now a set of type environments in which the variable is associated with a different type in each, each type being a constituent of the original union type. For example, if the original type environment contains the following type variables bound by the following type unions:

$$\begin{aligned} \mathit{typevar}_1 &\mapsto \{\mathit{type}_1, \mathit{type}_2, \mathit{type}_3\} \\ \mathit{typevar}_2 &\mapsto \{\mathit{type}_3, \mathit{type}_4\} \end{aligned}$$

then six type environments (i.e. cases) will be generated directly from these type variables,[8] each with one of the following six combinations:

$$\begin{array}{ll} \mathit{typevar}_1 \mapsto \mathit{type}_1 & \mathit{typevar}_2 \mapsto \mathit{type}_3 \\ \mathit{typevar}_1 \mapsto \mathit{type}_2 & \mathit{typevar}_2 \mapsto \mathit{type}_4 \\ \mathit{typevar}_1 \mapsto \mathit{type}_3 & \mathit{typevar}_2 \mapsto \mathit{type}_4 \\ \mathit{typevar}_1 \mapsto \mathit{type}_1 & \mathit{typevar}_2 \mapsto \mathit{type}_4 \\ \mathit{typevar}_1 \mapsto \mathit{type}_2 & \mathit{typevar}_2 \mapsto \mathit{type}_3 \\ \mathit{typevar}_1 \mapsto \mathit{type}_3 & \mathit{typevar}_2 \mapsto \mathit{type}_3 \end{array}$$

[8]More cases would be generated as a result of other union types in the type environment.

One immediate effect of this is that a message can be typed separately for each potential type of its arguments. For example, if the following message send expression:

rcvr testMe ifTrue: [rcvr trueTestMsg]

has as the type of its receiver, $typevar_1$, then the expression can be typed separately and consistently for each possible case derived from the bounds of the type variables:

$type_1$ testMe ifTrue: [$type_1$ trueTestMsg]
$type_2$ testMe ifTrue: [$type_2$ trueTestMsg]
or $type_3$ testMe ifTrue: [$type_3$ trueTestMsg]

Note that the type of the receiver is automatically propagated through to its use within the block by virtue of the type variable.

The task of creating the 'set' of cases is performed by *expand_union*, which actually returns a sequence of type environments to aid processing.

With the 'cases' created, the case analysis can be performed. As a result of the various structures and pairs being handled, the actual reduction of each case has been passed over to another function, namely *Reduce_cases*. This function takes the expression to be reduced, together with the sequence of cases generated by *expand_union*. The result of this call is a sequence of reduced type trees and a corresponding sequence of updated type environments. The latter sequence represents the updated form of the original sequence of cases, while the former sequence represents the same expression reduced under the various contexts provided by these cases. If every case has produced a completely reduced type, the whole expression can be reduced to the union of the cases. This can be done by extracting the reduced types from the resulting case sequence, and combining them to yield a union type. Because each element in the sequence of cases is a fully reduced type, the required union type (a set of types) is created simply by taking all the elements from the case sequence (a sequence of types). The corresponding type environments are also 'reduced' by merging them into a single type environment. The manner of this merging is left rather ambiguous by Graver, but is dealt with in §3.6.3. In the case where the reduction has not been complete, the attempt is abandoned, and the original values are returned unaltered.

$Reduce\_expr : TypeTree \times TypeEnv \rightarrow TypeTree \times TypeEnv$

$Reduce\_expr(ttree, \tau) \quad \triangleq$

let $\tau\_cases = expand\_union(\tau)$
in
let $(ttree\_cases, \tau\_cases') = Reduce\_cases(ttree, \tau\_cases)$
in
if $isTypeSeq(ttree\_cases)$
then (elems $ttree\_cases, merge\_env\_cases(\tau\_cases')$)
else $(ttree, \tau)$

$$\left(\begin{array}{ccc} ttree & , & \begin{pmatrix} \tau\_case_1 \\ \vdots \\ \tau\_case_n \end{pmatrix} \end{array} \right) \text{becomes} \left(\begin{array}{ccc} \begin{pmatrix} red\_ttree_{\tau\_case_1} \\ \vdots \\ red\_ttree_{\tau\_case_n} \end{pmatrix} & , & \begin{pmatrix} \tau\_case_1{}' \\ \vdots \\ \tau\_case_n{}' \end{pmatrix} \end{array} \right)$$

Figure 3.3: *Reduce_cases* in action

The definition of *expand_union* can be found on page 62, but it signature is as follows:

$expand\_union : TypeEnv \to TypeEnv^*$

The *Reduce_cases* function is designed to handle the sets and sequences used by the reduction functions. In essence, a given reduction function returns a pair — a reduced type tree and an updated type environment. When working with sets or sequences of pairs it is necessary to be able to split these pairs up, maintaining separate sequences of type trees and type environments, rather than have a single sequence of type tree, type environment pairs. This problem can been seen in the definition of *Reduce_expr_list*, and is confronted once again by the *Reduce_cases* function. This function, though, only acts as an indirection between the *Reduce_expr* function and the low-level *R* function. It takes a type tree and a sequence of cases, calls the *R* function using the first case, and then recurses over the rest of the cases. The results of the first case are then combined with the rest of the remaining cases, while ensuring that the sequences of type trees and type environments are kept separate. Figure 3.3 shows the action of *Reduce_cases* on a single type tree (*ttree*) and a sequence of type environments ($[\tau\_case_1, \cdots, \tau\_case_n]$).

$Reduce\_cases : TypeTree \times TypeEnv^* \to TypeTree^* \times TypeEnv^*$

$Reduce\_cases(ttree, \tau\_cases) \quad \triangleq$

if $\tau\_cases = [\,]$

then $([\,], [\,])$

else let $(ttree', \tau) = R[\![ttree]\!]$ hd τ_cases

in

let $(ttree\_cases, \tau\_cases') = Reduce\_cases(ttree,$ tl $\tau\_cases)$

in

$([ttree'] \frown ttree\_cases, [\tau] \frown \tau\_cases')$

After the establishment of the case analysis, the next important stage of the inference process is the interpretation of each expression. This is performed, for the most part, by the *R* function. *R* retains the same basic interface as the top level *Reduce* function, taking an expression (type tree) to reduce together with a context (type environment) and returning a reduced expression and an updated context.

$$R : \mathit{TypeTree} \rightarrow \mathit{TypeEnv} \rightarrow \mathit{TypeTree} \times \mathit{TypeEnv}$$

There are four cases for the *R* function to handle, corresponding to the four different forms of the type tree. The first and simplest case is when the type-tree expression is just a type. For this it is only necessary to return the type and leave the type environment unchanged. In a way this reflects the 'passive' nature of those syntax expressions which transform straight to a type. With such expressions there is no new type information to be derived. In contrast, the two 'active' expressions, the assignment and the message send, do provide new type information.[9]

$$R[\![t]\!]\,\tau \quad \triangleq \quad (t, \tau) \qquad\qquad \text{for } t \in \mathit{Type}$$

The next simplest expression is the type-tree list. It is only necessary, for this case, to pass the expression sequence over to the *Reduce* function, where it can be suitably handled.

$$\begin{aligned} &R[\![\mathit{ttrees}]\!]\,\tau \quad \triangleq && \text{for } \mathit{ttrees} \in \mathit{TypeTreeList} \\ &\qquad \mathit{Reduce}[\![\mathit{ttrees}]\!]\,\tau \end{aligned}$$

The assignment expression is the first of the two 'active' expressions to be handled. The assignment expression itself is reducible if the two constituent expressions are. In other words, if the left-hand variable can be reduced (trivial) and the right-hand expression can be dealt with likewise (not trivial), then the whole expression can be reduced. The reduction process, therefore, rests on the reduction of the right-hand expression. Both sides are reduced through the *Reduce* function to ensure a rigorous treatment.

If the reduction of the right-hand side has been completed, the reduction of the assignment expression itself is made possible. It is at this point that new type information is derived, in the form of an inclusion relation indicating that the right-hand type is included in the left-hand type. This new information is added to the type environment by the *add_incl* function, which not only adds the inclusion relation, but also checks to ensure that the new information does not contradict earlier inclusion relations. In the case where the reduction of the right-hand side is unsuccessful, the assignment expression is reconstructed with the partially reduced components, and returned with the updated type environment, on the grounds that new information may have been derived while reducing the right-hand expression.

---

[9]The distinction between passive and active expressions relates to their effect on the object space during run time. For all the passive expressions — object name and object creation expressions — the object space does not change detectably, even for an object creation. In fact for an object creation to become detectable requires either an assignment to a variable or an involvement with a message send as an argument. In addition to assignments, message sends may also be considered active expressions, if only for their parameter passing mechanism.

$$
\begin{array}{l}
R[\![\mathit{mk\text{-}AssignmentType}(lhs, rhs)]\!]\,\tau \quad \triangleq \\
\quad \text{let } (lhs', \tau') = \mathit{Reduce}[\![lhs]\!]\,\tau \\
\quad\quad \text{in} \\
\quad \text{let } (rhs', \tau'') = \mathit{Reduce}[\![rhs]\!]\,\tau' \\
\quad\quad \text{in} \\
\quad \text{if } \mathit{isType}(rhs') \\
\quad \text{then } (rhs', \mathit{add\_incl}(rhs', lhs', \tau'')) \\
\quad \text{else } (\mathit{mk\text{-}AssignmentType}(lhs', rhs'), \tau'')
\end{array}
$$

The last case to handle is also the most complicated, namely the message send expression. As with the assignment expression, the reduction of the message send depends on the successful reduction of its components, in this case the message receiver and arguments. The reduction of the receiver and argument expressions is carried out by the *Reduce* and *Reduce_expr_list* functions respectively, with the type environment being propagated from component to component, from the left to right, as each component is processed. Again, as with the assignment, if the reduction of the components has been complete, the reduction of the message send can commence, courtesy of the *Reduce_send* function. Finally, as with the assignment expression, if the reduction of the receiver and argument expressions is unsuccessful, a reconstructed, partially reduced message expression, is returned, together with an updated type environment.

$$
\begin{array}{l}
R[\![\mathit{mk\text{-}MessageType}(rcvr, sel, args)]\!]\,\tau \quad \triangleq \\
\quad \text{let } (rcvr', \tau') = \mathit{Reduce}[\![rcvr]\!]\,\tau \\
\quad\quad \text{in} \\
\quad \text{let } (args', \tau'') = \mathit{Reduce\_expr\_list}(args, \tau') \\
\quad\quad \text{in} \\
\quad \text{if } \mathit{isTypeSeq}([rcvr'] \frown args') \\
\quad \text{then } \mathit{Reduce\_send}(rcvr', sel, args', \tau'') \\
\quad \text{else } (\mathit{mk\text{-}MessageType}(rcvr', sel, args'), \tau'')
\end{array}
$$

In reducing a message send expression, the act of abstract interpretation is taken to its limit, involving, as it does, the simulated lookup and invocation of a method or methods. The *Reduce_send* function takes as its arguments the receiver type, the message selector, the argument types (a sequence of types) and the context (type environment) of the message send. In return the function provides either a type — the result of the message send — or a type tree — the reconstructed message send, with reduced components — together with a type environment, updated, in the case of a successful reduction, or not, in the case of a failure.[10] Implicit in the function is a test to ensure that the message selector has a corresponding method definition. If this fails, a FAIL flag is passed straight back to the original method body being typed, that is,

[10] In effect, this means that an incomplete reduction returns the message expression and type environment as they were just before the attempted method lookup.

back to the originating *infer_method_type*, defined on page 51.

$$Reduce\_send : Type \times Selector \times Type^{*} \times TypeEnv \rightarrow TypeTree \times TypeEnv$$

The first case to consider is where the evaluated receiver is an object type. In this case, the class of the receiver can be determined directly, and a simulated lookup and method invocation can be performed. The method lookup can be observed in the first line of the definition, where the method type map from the calling type environment has been indexed by the receiver class and the message selector.

With the invoked method extracted, it is now necessary to set up the context for the interpretation of the called method's body. This is performed in several stages, the first of which is to record the original calling self for later re-instantiation. With the original self type safe, the 'calling' and 'called' type environments are merged. This is necessary in order to provide the inclusion relations relating to any type variables used as arguments to the message send. It is also of use in the case of a self send, where instance variables would still be accessible, and the calling context would provide important call-site specific type information.

Argument passing is achieved by adding inclusion relations to the type environment of the form:

$$message\_argument \sqsubseteq method\_parameter$$

Note that this also includes the receiver of the message, despite the fact that the relation should strictly be one of equality. However the receiver type is correctly passed during the merging of the two type environments (*merge_envs* page 64), taking up its place as the new self type of the merged environment. With the arguments passed it is now possible to interpret the method body. At this point a test is carried out to see if there is any method body to interpret. If there is no body — if it is either a primitive method or it has been successfully typed in its own context — the message send can be completed by evaluating the method's return type in the newly merged context (type environment). If there is a body, it is passed on to the *Reduce* function for reduction in the new context. If this is successful — if the body's type tree has been reduced to a type — then the return type is evaluated and returned. If, on the other hand, the reduction has been unsuccessful, the whole message send is returned intact, as it was before the reduction attempt.

$Reduce\_send(rcvr, sel, args, \tau) \quad \triangleq$ for $rcvr \in ObjectType$

let $mk\text{-}MethodType(self, params, return, \tau', ttree) =$
$Method\_types(\tau)(Class(rcvr), sel)$
in
let $call\_self = Self\_type(\tau)$
in
let $\tau'' = merge\_envs(\tau, \tau')$
in
let $\tau''' = add\_incl(rcvr, self, \tau'')$
in
let $\tau^{iv} = add\_incls(args, params, \tau''')$
in
if $ttree = [\,]$
then $(evaluate\_in(return, \tau^{iv}), \mu(\tau^{iv}, Self\_type \mapsto call\_self))$
else let $(ttree', \tau^{v}) = Reduce[\![ttree]\!]\,\tau^{iv}$
in
if $isType(ttree')$
then $(evaluate\_in(return, \tau^{v}), \mu(\tau^{v}, Self\_type \mapsto call\_self))$
else $(mk\text{-}MessageType(rcvr, sel, args), \tau)$

Note that in both cases where the return type can be evaluated in the context of the method invocation, the returned context has been re-instantiated with the original self type. It should also be noted that all type variables (*TypeVar*, page 34) are assumed to have been re-tagged for each simulated 'invocation' of a given method, making them distinguishable between separate analyses of the same method. In this way name clashes can be avoided. As such, old type information, left in a type environment, for example after a message send, is in no danger of interfering with further code analysis.

The object-type case is the only case which involves a simulated lookup and invocation. Both the remaining cases merely attempt to manipulate the receiver into a form which can be handled by this first case. For the case where the receiver is a type variable, the problem is whether or not it has a suitable 'estimate'. All type variables are given initial upper and lower bounds of ANYTHING and NOTHING respectively, unless declared otherwise (i.e. the type variable's bound). As type information is extracted from the code, a better estimate of the type implied by a type variable may be made. Of the two bounds, the lower bound is the best, implying as it does type information flowing into the type variable, through assignments and argument passing. This can be seen from the calls to *add_incl* from the assignment case of the *R* function, and from the argument passing during the object-type case of *Reduce_send*, given that the type variable is being assigned to and passed to respectively.

As a result of this reasoning, a call to *evaluate_in* will return the lower bound of the type variable if it has one. If this is not possible, then it will return the type variable's upper bound.

If neither of these cases is possible then the function simply returns the type variable. As a result, if the type variable has no upper or lower bound other than the defaults of ANYTHING and NOTHING, the test to see if the receiver type hasn't changed will succeed and the message send will be returned intact and unsuccessful. If, however, *evaluate_in* does provide an estimate for the type variable, then this is passed on to whichever *Reduce_send* case can handle the new established receiver type.

$$
\begin{array}{ll}
\mathit{Reduce\_send}(\mathit{rcvr}, \mathit{sel}, \mathit{args}, \tau) \;\triangleq & \text{for } \mathit{rcvr} \in \mathit{TypeVar} \\
\quad \textsf{let } \mathit{rcvr}' = \mathit{evaluate\_in}(\mathit{rcvr}, \tau) & \\
\quad\quad \textsf{in} & \\
\quad \textsf{if } \mathit{rcvr}' = \mathit{rcvr} & \\
\quad \textsf{then } (\mathit{mk\text{-}MessageType}(\mathit{rcvr}, \mathit{sel}, \mathit{args}), \tau) & \\
\quad \textsf{else } \mathit{Reduce\_send}(\mathit{rcvr}', \mathit{sel}, \mathit{args}, \tau) &
\end{array}
$$

The final case handles receiver types which are union types. The problem for the union-type case is to ensure that each constituent type is reduced, and then to combine the results. The case is defined recursively, with the general case extracting an individual constituent type from the union set, and calling *Reduce_send* on it. The recursion bottoms out when all cases have been attempted. At this point the recombining of the reduced type trees and updated type environments takes place. The type trees, returned by the individual cases are initially combined into a sequence. This sequence is constructed from *hd_ttree*, a single type tree derived from the extracted type (*head_type*), and *tl_ttrees*, a sequence of type trees (and hence a type tree itself) derived from the recursive call on *tail_union_type*. The resulting sequence can then be tested for successful reduction by using *isTypeSeq*. If each case has been successful, the cases can be merged to give a fully reduced expression, extracting the elements of the type sequence to yield a union type. If any of the cases fails, the original message expression is returned intact and, again, unsuccessful.

$Reduce\_send(rcvr, sel, args, \tau) \quad \triangleq$ for $rcvr \in UnionType$

if $rcvr = \{\ \}$

then $([\ ], \tau)$

else let $head\_type \in rcvr$

in

let $tail\_union\_type = rcvr - \{head\_type\}$

in

let $(hd\_ttree, hd\_\tau) = Reduce\_send(head\_type, sel, args, \tau)$

in

let $(tl\_ttrees, tl\_\tau) = Reduce\_send(tail\_union\_type, sel, args, \tau)$

in

let $ttree\_cases = [hd\_ttree] ^\frown tl\_ttrees$

in

if $isTypeSeq(ttree\_cases)$

then (elems $ttree\_cases, merge\_envs(hd\_\tau, tl\_\tau)$)

else $(mk\text{-}MessageType(rcvr, sel, args), \tau)$

As the recombination takes place as the function comes out of its recursion, there will only ever be at most two elements in *ttree_cases*, one of which will be a union case. Another effect of recombining one level at a time, is that there will only ever be two type environments to merge.

3.6.3 Case analysis, unification and type inference

At the end of the reduction functions we have established the following mechanisms:

Transformation: Abstraction from syntax (parse trees) to types (type trees).

Reduction: High-level type inference through abstract interpretation.

Up to now, the degree of resolution achievable by the system has been determined by the *Reduce_expr* function. All the work prior to that stage was determined by the structure of the domain of interpretation, in particular the structure of the type trees, which was, in turn, dictated by the structure of the original syntax. In other words, the process of transformation and interpretation, up to the level of individual expressions, was determined by structure of the original language.

It is at the point where individual expressions are processed — i.e. *Reduce_expr* — that the fine tuning of the system can take place. From the coarse resolution provided by the type environment and the type domain itself, *Reduce_expr* determines how that type information is handled at the finest level. In breaking down the coarseness of the union type to the fineness of the object type, *Reduce_expr* is where case analysis takes place, specifically through the application of *expand_union* and *merge_env_cases*.

Expand_union takes a type environment and derives from it a set of type environments. It is this set of type environments which constitutes the set of cases to be analysed. In each case, all union types have been removed and replaced by their constituent types, one per case. The cases are formed on a combinatorial basis, that is each union type in the original case generates as many cases as there are constituents within that union type. Therefore the set expands in such a way that adding the contents of a union type multiplies the number of current cases by the number of elements in the added union type.

Expand_union works in the following way. First it updates the type variable map to take account of any new type information derived from the inclusion relations. From this point the creation of the cases can commence. The initial set of cases is derived from the variable types.[11] This set is expanded further by deriving cases for the type variables. The function finishes by converting the set of cases into a sequence, in order to ease their manipulation by *Reduce_expr*.

$$
\begin{array}{l}
\mathit{expand\_union} : \mathit{TypeEnv} \to \mathit{TypeEnv}^{*} \\
\mathit{expand\_union}(\tau) \quad \triangleq \\
\qquad \mathsf{let}\ \tau' = \mathit{evaluate\_tvs}(\tau) \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{let}\ \tau set = \mathit{expand\_vars}(\tau') \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{let}\ \tau set' = \bigcup\{\mathit{expand\_tvs}(\tau'') \mid \tau'' \in \tau set\} \\
\qquad\quad \mathsf{in} \\
\qquad \mathit{set\_to\_seq}(\tau set')
\end{array}
$$

The type variable map is updated by calling the *evaluate_tvs* function, which takes the type variables and invokes *evaluate_in* on each of them. *Evaluate_in* is one of the functions which deals with the inclusion relation set, and which interfaces with the underlying type inference algorithm designed by Graver in his thesis.

$$
\begin{array}{l}
\mathit{evaluate\_tvs} : \mathit{TypeEnv} \to \mathit{TypeEnv} \\
\mathit{evaluate\_tvs}(\tau) \quad \triangleq \\
\qquad \mathsf{let}\ \mathit{type\_vars} = \mathit{Type\_vars}(\tau) \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{let}\ \mathit{type\_vars}' = \{\mathit{tv} \mapsto \mathit{evaluate\_in}(\mathit{tv}, \tau) \mid \mathit{tv} \in \mathsf{dom}\,\mathit{type\_vars}\} \\
\qquad\quad \mathsf{in} \\
\qquad \mu(\tau, \mathit{Type\_vars} \mapsto \mathit{type\_vars}')
\end{array}
$$

[11] Astute readers will notice immediately that this particular step is unnecessary because all references to the variables and parameters have been removed through the transformation process. This would be a correct observation. This issue, which is central to the wider issue of how Graver has defined his case analysis, will be discussed in the section on problems relating to the type inference system, §3.7.

Both *expand_vars* and *expand_tvs* are defined below, with both functions reconstructing their respective fields — the variable type map and the type variable map. The reconstruction of the two maps is such that each element in the respective domains now maps to a type which is not a union type, but instead an object type or type variable which is a subtype of the type found in the original type environment.

$$expand\_vars : TypeEnv \rightarrow TypeEnv\text{-set}$$

$$\begin{aligned} & expand\_vars(\tau) \quad \triangleq \\ & \quad \{\mu(\tau, Var\_types \mapsto exp\_v\_types) \mid \{var \mapsto type\} \in Var\_types(\tau) \\ & \qquad \wedge \{var \mapsto exp\_type\} \in exp\_v\_types \\ & \qquad \wedge exp\_type \sqsubseteq type \\ & \qquad\quad \wedge exp\_type \in (ObjectType \cup TypeVar)\} \end{aligned}$$

$$expand\_tvs : TypeEnv \rightarrow TypeEnv\text{-set}$$

$$\begin{aligned} & expand\_tvs(\tau) \quad \triangleq \\ & \quad \{\mu(\tau, Type\_vars \mapsto exp\_tv\_types) \mid \{tv \mapsto type\} \in Type\_vars(\tau) \\ & \qquad \wedge \{tv \mapsto exp\_type\} \in exp\_tv\_types \\ & \qquad \wedge exp\_type \sqsubseteq type \\ & \qquad\quad \wedge exp\_type \in (ObjectType \cup TypeVar)\} \end{aligned}$$

At the other end of the case analysis mechanism is the ‘merge’ function, for combining the results of the various cases analysed. This is performed principally by the *merge_envs* function which merges two type environments into one, though *Reduce_expr* calls it indirectly through *merge_env_cases*, which deals with a sequence of type environments. *Merge_env_cases* is defined recursively, combining the sequence one by one, using *merge_envs*, as it comes out of the recursion. Note that because there will always be at least one case to deal with, τseq is guaranteed to be non-empty when the function is first called from *Reduce_expr* (page 54).

$$merge\_env\_cases : TypeEnv^* \rightarrow TypeEnv$$

$$\begin{aligned} & merge\_env\_cases(\tau seq) \quad \triangleq \\ & \quad \text{if } \tau seq = [\tau] \\ & \quad \text{then } \tau \\ & \quad \text{else let } hd\_\tau = \text{hd } \tau seq \\ & \qquad\quad \text{in} \\ & \qquad \text{let } merged\_tl\_\tau = merge\_env\_cases(\text{tl } \tau seq) \\ & \qquad\quad \text{in} \\ & \qquad merge\_envs(hd\_\tau, merged\_tl\_\tau) \end{aligned}$$

The process of merging two type environments is treated rather vaguely by Graver, and so a real understanding of how it should be performed can only be estimated. The functions defined below are an attempt to 'infer' a more complete definition than Graver's 'one liner' and one line description ([Gra89], page 74).

$$merge\_envs : TypeEnv \times TypeEnv \rightarrow TypeEnv$$

$$\begin{array}{l} merge\_envs(\tau_1, \tau_2) \quad \triangleq \\ \quad mk\text{-}TypeEnv(Self\_type(\tau_2), \\ \qquad\qquad merge\_var\_types(Var\_types(\tau_1), Var\_types(\tau_2)), \\ \qquad\qquad merge\_type\_vars(Type\_vars(\tau_1), Type\_vars(\tau_2)), \\ \qquad\qquad Method\_types(\tau_1), \\ \qquad\qquad merge\_incl\_rels(Incl\_relations(\tau_1) \cup Incl\_relations(\tau_2), \{\,\})) \end{array}$$

The principles behind combining the various fields would seem to be as follows.

self type: Type taken from the second environment.
This would initially appear to be an arbitrary choice. The self type should be an object type and hence only one self type may selected when merging a set of cases, but because the self type does not change between cases originating from the same location, any one of these cases should be capable of providing a suitable type. However, *merge_envs* is also used by the *Reduce_send* function's object-type case (page 59), where it merges the call-site environment with the environment recorded for the 'invoked' method. For this case, the two self types will be different, and since it is the type of the receiver which needs to be recorded at this point, it is the self type from the second, method-specific environment which is used.

variable type map: For each variable occurring in both maps, the union type of the two types is taken for the merged map. All other entries are taken as they are. Because entries in the variable map do not change, it is possible for the map of a method to merged with itself during *Reduce_send* without any problems arising.

$$merge\_var\_types : (VarId \xrightarrow{m} Type) \times (VarId \xrightarrow{m} Type) \rightarrow VarId \xrightarrow{m} Type$$

$$\begin{array}{l} merge\_var\_types(var\_types_1, var\_types_2) \quad \triangleq \\ \quad \{var_1 \mapsto type_1 \mid (var_1 \mapsto type_1) \in var\_types_1 \wedge var_1 \notin \mathsf{dom}\, var\_types_2\} \\ \qquad \cup \\ \quad \{var_2 \mapsto type_2 \mid (var_2 \mapsto type_2) \in var\_types_2 \wedge var_2 \notin \mathsf{dom}\, var\_types_1\} \\ \qquad \cup \\ \quad \{var \mapsto (type_1 \cup type_2) \\ \qquad\qquad\qquad \mid (var \mapsto type_1) \in var\_types_1 \wedge (var \mapsto type_2) \in var\_types_2\} \end{array}$$

type variable map: The merging of two type variable maps follows the same pattern as for the merging of two the variable type maps, although it would also seem possible to use the merged set of inclusion relations to obtain an updated type variable map. When used in conjunction with *Reduce_send*, the re-tagging of any type variables between 'invocations' of the same method ensures that there are no clashes during the merging.

$$
\begin{array}{l}
\mathit{merge\_type\_vars} : (\mathit{TypeVar} \xrightarrow{m} \mathit{Type}) \times (\mathit{TypeVar} \xrightarrow{m} \mathit{Type}) \rightarrow \mathit{TypeVar} \xrightarrow{m} \mathit{Type} \\
\mathit{merge\_type\_vars}(\mathit{type\_vars}_1, \mathit{type\_vars}_2) \quad \triangleq \\
\quad \{\mathit{typvar}_1 \mapsto \mathit{type}_1 \mid (\mathit{typvar}_1 \mapsto \mathit{type}_1) \in \mathit{type\_vars}_1 \wedge \mathit{typvar}_1 \notin \mathsf{dom}\, \mathit{type\_vars}_2\} \\
\quad\quad \cup \\
\quad \{\mathit{typvar}_2 \mapsto \mathit{type}_2 \mid (\mathit{typvar}_2 \mapsto \mathit{type}_2) \in \mathit{type\_vars}_2 \wedge \mathit{typvar}_2 \notin \mathsf{dom}\, \mathit{type\_vars}_1\} \\
\quad\quad \cup \\
\quad \{\mathit{typvar} \mapsto (\mathit{type}_1 \cup \mathit{type}_2) \mid (\mathit{typvar} \mapsto \mathit{type}_1) \in \mathit{type\_vars}_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, (\mathit{typvar} \mapsto \mathit{type}_2) \in \mathit{type\_vars}_2\}
\end{array}
$$

method type map: Taken from the first environment for the benefit of *Reduce_send*, object-type case. This is necessary in order to propagate the latest method type map from the method being typed by *infer_method_type*, where the latest map is placed in the method's type environment, through to any methods invoked by it. For all other uses of *merge_envs*, the choice is arbitrary because both versions should be the same.

inclusion relations: It would seem that the two sets undergo a systematic 'add inclusion', to combine the two sets, and also to ensure that they are consistent.

$$
\begin{array}{l}
\mathit{merge\_incl\_rels} : \mathit{Incl\_rel}\text{-}\mathsf{set} \times \mathit{Incl\_rel}\text{-}\mathsf{set} \rightarrow \mathit{Incl\_rel}\text{-}\mathsf{set} \\
\mathit{merge\_incl\_rels}(\mathit{additional\_incl\_rels}, \mathit{main\_incl\_rels}) \quad \triangleq \\
\quad \mathsf{if}\ \mathit{additional\_incl\_rels} = \{\,\} \\
\quad \mathsf{then}\ \mathit{main\_incl\_rels} \\
\quad \mathsf{else}\ \mathsf{let}\ \mathit{incl\_rel} \in \mathit{additional\_incl\_rels} \\
\qquad\qquad \mathsf{in} \\
\qquad \mathsf{let}\ \mathit{additional\_incl\_rels}' = \mathit{additional\_incl\_rels} - \{\mathit{incl\_rel}\} \\
\qquad\qquad \mathsf{in} \\
\qquad \mathsf{let}\ \mathit{main\_incl\_rels}' = \mathit{add\_incl}(\mathit{Smaller}(\mathit{incl\_rel}), \mathit{Greater}(\mathit{incl\_rel}), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{main\_incl\_rels}) \\
\qquad\qquad \mathsf{in} \\
\qquad \mathit{merge\_incl\_rels}(\mathit{additional\_incl\_rels}', \mathit{main\_incl\_rels}')
\end{array}
$$

A key feature of merging two environments is that all the types split up through the *expand_union* process are recombined back into union types. The significance of this is that any results, from a given expression, which depend on a certain combination of types thrown up by the case analysis process are lost when the types are merged back together. In other words, the effect of a given case analysis process is restricted to an individual expression rather than propagated through a sequence of expressions.

The rest of the functions, described below, relate to the low level handling of the inclusion relations. They interface directly into the underlying type inference algorithm designed by Graver, and as such, they are outside the scope of this thesis. The purpose of the formal description presented in this chapter was to show how type information is extracted from the code, and how this information is treated. The lowest level analysed in this thesis, is the case analysis mechanism, because it is at this level that the resolution of the type inference system is determined. All such analysis presupposes the existence of the Graver's algorithm, described in detail in his thesis, together with a suitable mechanism for handling the inclusion relations. In principle, Graver's algorithm provides a mapping from type variables to types which is consistent with the inclusion relations in a given type environment.

The first two functions outlined below add inclusion relations to a type environment. The first function adds a single relation, while the second adds a sequence of them, the sequence being represented by two separate sequences to be paired-up index-wise.

$$\mathit{add\_incl} : \mathit{Type} \times \mathit{Type} \times \mathit{TypeEnv} \rightarrow \mathit{TypeEnv} \ \cup \ \{\text{FAIL}\}$$
$$\mathit{add\_incls} : \mathit{Type}^* \times \mathit{Type}^* \times \mathit{TypeEnv} \rightarrow \mathit{TypeEnv} \ \cup \ \{\text{FAIL}\}$$

Add_incl takes two types (e.g. $\mathit{add\_incl}(\mathit{sub}, \mathit{super}, \tau)$) and inserts an inclusion relation, in this example case $\mathit{sub} \sqsubseteq \mathit{super}$, into the type environment. Because this involves ensuring that the new relation is consistent with the old set of relations, the possibility of an inconsistency arising is handled by returning FAIL. If the relation can be added while retaining consistency, an updated copy of the type environment is returned. *Add_incls* performs the same task as *add_incl* for sequences of types of equal length, forming each inclusion relation by taking corresponding types from each sequence. For example $\mathit{add\_incls}(\mathit{subseq}, \mathit{superseq}, \tau)$ would add, for each i up to the length of the sequences, $\mathit{subseq}[i] \sqsubseteq \mathit{superseq}[i]$. Again, if the addition of one of the relations results in an inconsistency within the type environment, the function returns FAIL.

The test for consistency when adding inclusion relations to a type environment utilises Graver's inference algorithm. It ensures that a suitable mapping from type variables to types exists for the updated set of inclusion relations.[12]

The final function, *evaluate_in*, utilises the result of Graver's algorithm, namely the mapping returned. It uses this mapping to return a consistent type which can be used in place of the type variable it receives. There is a special case running throughout these low level functions, relating to type variables which are still bound by their default bounds, ANYTHING

[12]It would seem that Graver was more concerned with ensuring that such a mapping exists, rather than obtaining a suitably general mapping. The definition of his algorithm returns a set of possible mappings, each of which is correct and consistent with the inclusion relations. However the mapping returned is chosen arbitrarily from that set, with no attempt being made to see if there is a most general mapping.

and NOTHING. If the bounds haven't changed from their default values, the type variable is assumed to be unbound, no attempt is made to infer its type, and in the case of *evaluate_in*, the type variable is returned as the result.

$evaluate\_in : Type \times TypeEnv \rightarrow Type \ \cup \ \{\text{FAIL}\}$

As with *Reduce_send*, there is an implicit test in all three of the above functions. If any of these functions fails, for example if the inclusion relations are not consistent, then a FAIL flag is returned straight to the originating *infer_method_type* (page 51).

3.7 Problems with type inference

This section will deal with two problems associated directly with the type inference system. The following sections will then deal with problems associated with the underlying type system.

As far as the type inference system goes, everything is geared towards the 'reduction' functions, and it is these functions which control how the system behaves. The second problem we shall consider relates specifically to this part of the process. However, our first problem relates to the direct typing of variables and parameters. That is, to the typing of variables and parameters with non-variable types, as well as to the instantiation of object type parameters.

The type system developed by Johnson and Graver allows for the declaration of non-variable types by the programmer, as well as for the declaration or allocation of type variables to untyped variables and methods (i.e. parameter and return types). This presents the system with two ways of accessing the type of an expression:

$$variable \rightarrow declared\ type$$

for variables with declared non-variable types (i.e. object or union types), and

$$variable \rightarrow type\ variable \rightarrow inferred\ type$$

for variables (or parameters and method returns) which have been allocated a type variable.

The real problem starts during the transformation stage, where a variable, or other identifier, is replaced by its type. In the case of a variable which has been allocated a type variable, one variable is, in effect, replaced by another, and the pattern of usage of the original variable is replicated by the new type variable. For example

```
var msg: (var extract: tmp)
```

will be transformed into

t_{var} msg: (t_{var} extract: t_{tmp})

where t_{var} and t_{tmp} are the type variables associated with the variables var and tmp. If, on the other hand, a variable has a declared type, then all instances of that variable will be replaced with the declared type. If, in the example above, the type of var was some union type, $\{t_1, \ldots, t_n\}$, then the transformed expression would be:

$$\{t_1, \ldots, t_n\} \text{ msg: } (\{t_1, \ldots, t_n\} \text{ extract: } t_{tmp})$$

At this point the case analysis of the expression breaks down. Instead of there being n cases due to var, one for each constituent type of var, there would be n^2 cases, covering each combination of constituent types between the first and second occurrences of var:

$$\begin{array}{c}
t_1 \text{ msg: } (t_1 \text{ extract: } t_{tmp}) \\
t_1 \text{ msg: } (t_2 \text{ extract: } t_{tmp}) \\
\vdots \\
t_1 \text{ msg: } (t_n \text{ extract: } t_{tmp}) \\
\vdots \\
t_n \text{ msg: } (t_1 \text{ extract: } t_{tmp}) \\
t_n \text{ msg: } (t_2 \text{ extract: } t_{tmp}) \\
\vdots \\
t_n \text{ msg: } (t_n \text{ extract: } t_{tmp})
\end{array}$$

There is a clear possibility that a 'non-existent' error might be detected as a result of one of these spurious cases.

The problem is that during the transformation process, connections between uses of a given variable or other identifier are lost if they are not replaced by some other mechanism for linking these uses. This is why the use of type variables does not produce this problem, whereas the use of declared, non-variable types does. What is needed, therefore, is to extend the level of indirection provided by the type variables, over the whole system, using the declared bound of a given type variable to express the intended declared type.[13] It is obvious at this point that dealing with the 'variables map' of the type environment, during the expansion of union types, has no impact on the actual case analysis mechanism. Its inclusion (as part of a larger mapping in Graver's description) might have been to provide an indication that there is a link between the declared, non-variable union types and their associated variables, but as is clear from the description of the system, any such links are lost during the transformation process.

The second problem is related specifically to the case analysis mechanism, and how type information is propagated between expressions. The situation in Graver's system, as it stands after being 'tidied up', is that for an individual expression a set of cases is produced. So for example the message send

```
rcvr msg: arg
```

will generate a set of cases, and for each case the receiver, argument and finally the message send itself will be processed by the reduction functions. After all the cases have been dealt with, all the information derived from each one is merged into a single result.

There seem to be two basic consequences of this. Firstly, there is no propagation of case information between expressions, even between the expressions which make up the arguments

[13] Strictly speaking, the need for a level of indirection is only necessary for union types, because the problem occurs during the expansion of such union types for the purposes of case analysis.

of a message send.[14] For example, if a case produces a specific return type as a result of a particular receiver type, this pairing up is lost after the expression has been dealt with. This information disappears as the type of the receiver and the return type are merged into their respective union types. This problem most clearly shows itself when considering the type interdependency of two variables. Consider the following piece of code:

```
msg: par
    | temp |
    temp := par startup.
    par testOfClass ifTrue: [temp classDependentMessage]
```

During the typing of blocks, the contents are 'reduced' as for a normal expression, using the context of the defining method. If a type error is found, the block is given a special type, marking it as an 'inconsistent block'. Then, when it comes to passing the block to a method such as ifTrue:, for the false case, where the block is not executed, the method accepts arguments of any type, including the special type assigned to 'bad' blocks. In this way a block which is only type correct for the true case, for example, can also be type checked for the false case.

In the above example the type of temp depends on the type of par, assuming that the message send startup produces a suitable result. The testOfClass message determines the class of par such that the message sent to temp, classDependentMessage, is only sent if the correct class is encountered. For simplicity, consider the case where the type of par, via a type variable, is the union type $\{t_x, t_y\}$. The first expression would then produce, say, a typing for temp of $\{p_x, p_y\}$. At the point just before the merging of the types at the end of the first expression, there would be the following cases:

$$\{t_{par} \mapsto t_x, t_{temp} \mapsto p_x\}$$
$$\{t_{par} \mapsto t_y, t_{temp} \mapsto p_y\}$$

However after the merge, there would only be the following:

$$\{t_{par} \mapsto \{t_x, t_y\}, t_{temp} \mapsto \{p_x, p_y\}\}$$

The second expression relies on the correspondence between the types of par and temp. Assuming that the message classDependentMessage is only understood by one of the types of temp, and that the test on par (testOfClass) ensures that the correct case receives it, then it is clear that the code is alright. The following two cases will be dealt with correctly:

$$\{t_{par} \mapsto t_x, t_{temp} \mapsto p_x\}$$

[14]The problem concerning the lack of propagation between arguments in a message send is not as crucial as the problem of propagating information through a sequence of expressions. This is essentially because for the message send most of the necessary cases are created when the whole expression is first analysed, prior to 'reduction'. The only other cases which could arise would be due to some form of assignment, for example:

```
rcvr check: (temp := var extract) tick: (temp query)
```

In this case the assignment to temp may involve a union type, whose expansion when analysing the first argument should be consistent with its expansion when analysing the second argument.

Assuming this is the correct type for temp to receive the message classDependentMessage, sending the message testOfClass to par should return true. Since the block has been successfully dealt with for this type of temp, the block may be passed with the message ifTrue: to the receiver true. The type of the argument will match-up with the required type, and the whole expression will be accepted as correct code.

$$\{t_{par} \mapsto t_y, t_{temp} \mapsto p_y\}$$

For this case, the type of temp is not correct for the message classDependentMessage, and the reduction of the block will fail causing it to be assigned the special 'inconsistent block' type. However the message send to par will return false, so that when the block is passed with the message ifTrue:, the type of the block will be compatible with the expected argument type (ANYTHING), and again the expression will be accepted as correct.

Unfortunately, because the case information from the assignment expression has been lost, these are not the only two cases to be considered. Of the following two cases one will succeed, but the other will fail.

$$\{t_{par} \mapsto t_y, t_{temp} \mapsto p_x\}$$

The case above will succeed because the message send to par will return false. As a result the expression will be accepted irrespective of how the block is typed.

$$\{t_{par} \mapsto t_x, t_{temp} \mapsto p_y\}$$

This is the case which will fail, despite the fact that it will never occur when the code is executed. The message send to par will return true, so the typing of the expression will rest on the successful typing of the block. However this time the type of temp will be wrong for it to receive the message classDependentMessage. As a result the block will be assigned the special 'inconsistent block' type, and the ifTrue: message send will fail because of a bad argument type.

The problems of the type inference system, rather than those of the underlying type system, seem to boil down to the case analysis mechanism. Even the need for a level of indirection between variables and declared types only comes to light after considering case analysis. The real problem, though, is in the lack of case information being propagated between expressions. Clearly some form of propagation would be welcome, for example to help overcome the problem of interdependency shown in the example above. It would seem that the urge to merge stems from a fear of an explosion of cases, as each case produces more cases, and each of those cases produces even more cases. However, there should be a limit on the number of cases possible, based on the number of items being typed in the system. In the current set up cases are created, used and then merged back together, only for them to be split up for the next expression. The real question is, what is there to prevent the system from doing away with the case merging, and instead propagate case information properly between expressions?

3.8 Problems with the type system

In this section, the problems we are considering are not specific to type inference, but instead apply to any type system that is based on a "one variable — one type" approach.

3.8.1 Assignments

Consider once again the assignment example given in the informal overview in §3.2.

```
var := object
```

Say that, for example, the object on the right-hand-side is a 'number', of class Number and object type *Number*.[15] The derived constraint will then be

$$Number \sqsubseteq t_{var}$$

Now consider the following message send

```
var numberMsg
```

assuming that the corresponding method, numberMsg, is only implemented in class Number. By examining the lower bound of t_{var}, a simulated lookup of the corresponding Number method can be done, and, assuming everything else is in order, the message send can be 'completed' and the expression declared type correct.

Next, suppose that a few lines later there is an another assignment to the variable

```
var := aBoolean
```

and that the object on the right-hand side is an object of class True (object type *True*). This will result in an *additional* constraint on the type t_{var}

$$True \sqsubseteq t_{var}$$

yielding a new lower bound

$$\{True, Number\} \sqsubseteq t_{var}$$

If we now analyse the previous message send retrospectively, it could be argued that the expression in now no longer type correct. However, instead of arguing that point, consider the following message send, which comes next:

```
var booleanMsg
```

where booleanMsg is only defined in the classes True and False, together with a set of experimental classes implementing a form of fuzzy logic, but definitely not in class Number. Now,

[15]For simplicity let us assume the existence of a single Number class, with associated object type *Number*, rather than the complex structure of abstract classes and subclasses which exists in the Smalltalk programming environment.

when a simulated lookup is attempted, the two cases derived from the lower bound must be considered separately. In the case of type *True* the message send should check out correctly, but in the case of type *Number* the lookup will fail. The failure of the *Number* case should mean that the whole expression will fail, despite the fact that the code is correct.

It could be argued that since one of the cases succeeds, the expression as a whole should succeed. However if one considers the case where the second message is specific to class Number, rather than True, it is clear that this solution is unsatisfactory.

Looking at the dataflow through this piece of code, the solution seems obvious:

var := aNumber	Variable var refers to an object of class Number.
var numberMsg	Message send to object referenced by var.
	Dereferencing reveals receiver's class to be Number.
var := aBoolean	Variable var refers to an object of class True.
var booleanMsg	Message send to object referenced by var.
	Dereferencing reveals receiver's class to be True.

Notice that all information regarding the first assignment is discarded when the second assignment is encountered, so preventing any clash between the two assignments. By concentrating on the object rather than the variable, the potential problem of multiple assignments can be avoided.

It would seem that by keeping track of what has been assigned we are able to overcome our initial typing problem. In order to complete our solution, it must be recognised that the type of a variable is not totally active throughout its existence, but only that part indicated by the last assignment. Keeping track of such assignments will most certainly require some form of dataflow analysis.

Although the reassignment of a variable in such a way that the type system is unable to cope may seem like poor programming practice, it does raise an important issue — namely, whether the type system should be allowed to dictate to the language semantics. In fact the problem of reassignment is a real one, and manifests itself regularly in Smalltalk-80 in the guise of the 'undefined' variable. When a variable first comes into scope, and before it has been explicitly assigned an object, it will normally reference the 'undefined' object, nil. However once the variable has been properly initialised it will most likely reference an object quite unlike the 'undefined' object. The problem for the type system is how to handle this initial period in the variable's lifetime before it has been properly initialised, and provide a useful typing for the rest of the variable's lifetime. The problem is made somewhat more complex, especially for instance variables, by the uncertainty of when or how the variable will be explicitly initialised. Unfortunately this is a problem which most type systems choose to side-step.[16]

[16] The whole problem of dealing with uninitialised variables appears to be a vexed question with respect to type systems, where there is a real dilemma between 'special-casing' such variables — separating the notion of the undefined variable from the rest of the type system — or treating them on equal terms with respect to the rest of the type system and so risk weakening the type system. A novel example of the former treatment can be found in the language developed by Palsberg and Schwartzbach [PS91a, PS91b].

So, by including dataflow analysis it would appear that we can overcome this ‘fundamental’ limitation in the type system. By incorporating dataflow analysis into the type inference mechanism we can demarcate regions over which assignments are ‘active’. The question we now ask is: Is the solution to the problem merely to alter the type inference system to take account of dataflow?

The answer, it would seem, is: yes, but with some strong reservations. By seeming to vary the type of a variable depending on its current use, we are moving away from the basic premise of using a type system, namely the assignment of a single type to a variable, and that variable’s consistent usage with respect to that type throughout the variable’s lifetime. Also by relating the type of a variable to the object assigned to it — i.e. the variable’s type is the class of the object assigned to it, but only for the duration of that assignment — rather than the other way round — i.e. assigning an object to a variable results in the addition of the object’s class to the variable’s overall type — we have introduced a shift in emphasis. What happens if we go all the way? What happens if we construct a completely separate dataflow analysis system? In the next chapter we will do just that.

3.8.2 Arrays

Collection structures are typed by using the parameter part of an object type. For example an array of integers would be typed as:

$$\textit{mk-ObjectType}(\textit{Array}, [\textit{Integer}])$$

(ignoring the problems of not using type variables) or, using Johnson and Graver’s notation:

```
Array of: Integer
```

In effect this produces an homogeneous array, with all elements expected to respond to the same set of messages. By using explicit class membership testing, heterogeneous arrays could be accommodated. For example, an element at some position, indexed by x, could be tested for class membership, and dealt with accordingly:

```
(anArray at: x) testClass ifTrue: [(anArray at: x) specialMessage]
```

Under the current typing scheme, the distinction made in typing this expression would be between the set of element types making up the parameter type of the array. As a more realistic example, consider the following expression dealing with an array of characters and integers (type Array of: <Character + Integer>):

```
(anArray at: x) isCharacter ifTrue: [(anArray at: x) characterAssassination]
```

During the typing of this expression, cases are made for the array element being a character, in which case the block should type correctly, and the element being an integer, in which case the typing of the block is irrelevant. Note that no attempt has been made to determine which element in the array is being dealt with.

Both these examples have made use of explicit class membership tests, provided by the programmer. Now consider the following expressions:

```
anArray at: 3 put: aThing
(anArray at: 3) doSomethingWithThing
```

If doSomethingWithThing is a message which is recognised by all possible types held in anArray then there is nothing wrong with the above expressions. However, if the type of anArray is

Array of: <Thing + Other>

where Others do not respond to the 'do something' message, then the above expressions will not be accepted. The problem is that there is no structure visible to the type system, such as the explicit class test shown above, which would guarantee that the message is only sent to an object which understood it, despite the fact that the array element in question is clearly a Thing.

This problem marks the difference between a system which handles homogeneous collections, and one which is able to cope with heterogeneous collections. A type system alone, with its abstraction from values to types, loses the necessary information to distinguish between individual elements in a collection. Any solution to this problem, once again, requires some form of flow analysis to provide the necessary positional information.

This is actually a basic problem, not only for type systems, but also for dataflow analysis, stemming as it does from trying to determine what is in the structure at any given point in time. This problem will be looked at again in §6.8, where its implications for dataflow analysis will be examined.

3.9 Control flow and blocks

Unlike the cut-down language used throughout the main sections of this chapter, Graver and Johnson's TS does handle block expressions. Determining the type-correctness of a block expression starts by attempting to reduce its contents — an expression — using the context of the block's defining method. This is the correct way to proceed for those cases where the block will be invoked. However, because blocks are only selectively invoked, it is still correct to create a block which is 'type incorrect' under some circumstances, as long as it is not executed under those circumstances. We have already seen this in action in §3.7 and §3.8.2, but let us look at a more general example:

```
anObject isCharacter ifTrue: [anObject doCharacterThing]
```

Assuming that doCharacterThing is only understood by Characters, if anObject has an associated union type, via a type variable, of which one constituent is Character, then the typing will proceed without error. The reasoning behind this lies with the way in which the ifTrue: method has been typed for both the True case and the False case. The true case requires a type-consistent, or rather, type-correct block as its argument, while the false case accepts anything on the basis that it will not be executed. The typing of the whole expression commences with the creation of cases, one of which has anObject (or rather its type-variable stand in) mapped to the type Character. For each case, except the 'character' case, the block will fail to type correctly, due to the doCharacterThing message send which is 'character' specific. However the whole expression will type correctly so long as none of these cases are passed to the 'true' implementation of ifTrue:. This depends on how the expression which makes up the receiver of ifTrue: is handled. If the typing of this receiver expression can guarantee that only the 'character' case invokes the 'true' ifTrue:, while all the others invoke the 'false' version, then there will be no problems.

For the purposes of our discussion, we have the following 'cases':

<character> isCharacter ifTrue: [<character> doCharacterThing]

and

<other> isCharacter ifTrue: [<other> doCharacterThing]

Putting the analysis of the receiver expression, anObject isCharacter, to one side for a moment, the analysis of the blocks reveals the following 'cases':

<character> isCharacter ifTrue: <a type-consistent block>

and

<other> isCharacter ifTrue: <a type-inconsistent block>

Recapping, the two implementations of ifTrue: involved in this particular situation are the True implementation, expecting a type-consistent block, and the False implementation, expecting anything and everything, including a type-inconsistent block. So, everything rests on the receiver expression. In this particular case, the test expression, which is what the receiver expression in fact is, is constructed so that only a Character returns true. This is done by having the Character implementation of isCharacter return true, and have all other implementations in other classes return false. The necessary combinations of true and false, character and non-character cases follow, and the expression can be correctly typed.

However, this example only works because the distinction required by the ifTrue: message was made at a *class* level. The cases provided initially, breaking the situation into a character/non-character scenario, are backed up by the isCharacter message, the result of which is 'hard-wired' according to the class of its receiver, and hence, 'hard-wired' into each implementation's return type.

In contrast, if the result of the test expression is based on some run-time evaluation or object level comparison, then the case analysis breaks down. To illustrate this point, consider the above example rewritten to use an explicit run-time comparison:

anObject class = Character ifTrue: [anObject doCharacterThing]

Now, the result of the test expression comes from the run-time comparison between the class of anObject and the class name Character. There is now no way of tying in the result of the test (true or false) with a general typing of the equality method = which has a return type of <True + False> (true or false) irrespective of the message's receiver. As a knock-on effect of this, all possible cases originating from anObject will result in both True and False from typing the test expression. The effect of this continues down the expression with the further result that both type-consistent (the character case) and type-inconsistent cases (all other cases) will be considered as arguments types for both the True and False implementations of ifTrue:. The ultimate effect is that for at least one case the 'knocked-on' true case from the receiver expression will be matched by a type-inconsistent block, which will then be rejected by the True implementation of ifTrue:.

Ironically, the example which Johnson has been using since his OOPSLA'86 paper [Joh86] to describe how the 'case analysis' mechanism in TS works, namely

```
aView notNil ifTrue: [aView ... ]
```

(from method controlToNextLevel in class Controller) is actually implemented in ParcPlace Smalltalk[17] as:

```
aView == nil ifFalse: [aView ... ]
```

The situation becomes even hairier when 'non-local return blocks' are considered. A non-local return block is one which, when invoked, returns control out of the method in which it was defined, after it has finished. For example, consider the following expression, defined in the body of some (unspecified) method:

```
anObject isCharacter ifTrue: [↑anObject]
```

If anObject is a character, the test expression will return true, and the block will be invoked. At this point the 'character' referenced by anObject is returned, not only from the invoked block, but also out of its enclosing method. This action is equivalent to reaching the return expression at the end of the method.

The actions described above are, in fact, deducible by the type inference system. However, now consider the following *sequence* of expressions:

```
anObject isCharacter ifTrue: [↑anObject].
anObject unCharacterLikeMessage
```

Clearly if anObject was a character the method will already have returned by the time the unCharacterLikeMessage message is sent. As a result there should be no problems if characters do not understand unCharacterLikeMessage. However, once again, because of the lack of case information propagation between expressions, this sequence of events will not be detected, and the second expression, when analysed for the anObject/Character case, will fail.

Finally, the problem which ultimately undermines the use of type inference, and, in fact, the use of a type system with blocks, is the method which only uses non-local return blocks through which to return control. For example, consider the following simple method:

[17] Johnson's example is most likely taken from the Tektronix implementation of Smalltalk, while the ParcPlace implementation was most likely chosen for efficiency reasons.

```
testMe
    "a method which returns the number 42 or the character $a, depending
    on the state of the instance variable aFlag."
    aFlag isCharacter
        ifTrue: [↑$a]
        ifFalse: [↑42]
```

Unfortunately, the type inference system will make the implicit ↑self at end of the method explicit, and will take it into account when typing the method. At best this will result in a method return type made up of Character, Integer and the defining class of the method. Of course this type is wrong, because, as simple flow analysis will determine, the final ↑self is never executed — the method only ever returns a Character or an Integer. Although in this particular case the method could be restructured to replace the non-local return blocks with local return blocks (which pose no problems to the inference system) it is possible to construct methods which only work with non-local return blocks.[18]

We will return to the basic problem of determining the control flow, and hence the dataflow through a block in Chapter 6.

3.10 Summary

This chapter started with a look at the technique of abstract interpretation, used as a general tool for code analysis. This was then followed by an informal overview of how Graver utilised this technique for his type inference system. A formal description of a minimalist version of his system followed, taking up the bulk of this chapter. Finally an analysis and critique of this 'basic' system finished off the chapter.

The minimalist language used in this chapter has enabled us to analyse the basic properties of Graver's type inference system. In §3.5 we observed the abstraction mechanism at work as the program code was transformed from a syntactic domain to a type domain. The gradual metamorphosis of this initial typing to a final typing was carried out by the combined efforts of the iteration functions, which coordinated the management of the task, and the reduction functions, which carried out the low level type inference work.

The typing of the language provides an abstraction of the program, from the raw code, with its imperative properties, to the domain of types, with their functional properties. These functional properties can be seen as the requirement that the use of a variable must be correct with respect to its type, irrespective of the location of such a use. Clearly the use of case analysis leads to areas where this condition is weakened to allow for union types. However, outside of a given case analysis, the basic premise holds that for a given expression, its type correctness can be determined independently of the state of the system on executing that expression. We can see from this that although the type inference mechanism appears to follow an 'imperative' control-flow path as it performs its abstract interpretation, it must produce a result which is consistent with the required functional abstraction of the type system. As a result, the type

[18] An example of such a method can be found on page 154.

system itself is unable to perform any imperative dataflow analysis, beyond the simple case analysis used to help with the typing of blocks.

Section 3.7 commenced the analysis and critique of both the type system (§3.8) and the type inference system (§3.7 and §3.9). In particular, §3.8 ("problems with the type system") showed an example of valid Smalltalk code which cannot be typed, and speculated about the possibility of combining a type system with dataflow analysis to handle such problems inherent in the type system. The idea of 'breaking' the functional nature of the type system by introducing dataflow analysis suggests taking the technique of case analysis to its extreme, and applying it over the entire system, propagating the case analysis information in exactly the way in which it is not propagated in the current type inference system. Finally, §3.9 looked at the handling of control flow by the type system, and showed how the 'case analysis' of the type system is of limited use.

So far we have seen the following in action: typing as functional abstraction, the inability to type some valid Smalltalk expressions, and that control-flow analysis is only really capable of handling unary 'test' messages (eg. isNil, notNil, isCharacter etc., in combination with blocks). In the following chapters we will take these ideas further by designing a dataflow analysis system which is capable of taking the place of a type system, while also avoiding some of the problems inherent in such a type system.

Chapter 4

Dataflow analysis

Inanimate objects are classified scientifically into three major categories — those that don't work, those that break down and those that get lost.

Russell Baker

When the work covered by this thesis was first formulated, it was envisaged that a type system, like the one presented in the preceding chapter, could benefit from the addition of a supportive dataflow analysis system [JG88]. As such, the dataflow analysis system was seen as an additional part of a type inference system, providing feedback to help in the latter's analysis process. However, after some initial work on the design of a dataflow analysis mechanism, it soon became clear that there was significant overlap between the activities of the two systems in their respective searches for potential 'does-not-understand' coding errors. This, and the following two chapters, present the results of the investigation into formulating such a dataflow analysis mechanism, culminating in §6.10 which asks the question: what is type checking in a dynamically bound language?

Basic dataflow analysis is essentially the analysis and annotation of code with dataflow information — that is, knowledge about what values may be currently assigned to a variable at a given point in the code. [ASU86] provides an overview of 'global' dataflow analysis as well as a description of how such analysis can be applied to a simple imperative language with conditional expressions and control loops. More detailed descriptions of program dataflow analysis can be found in [MJ81], which also describes the use of abstract, or non-standard interpretation with respect to code analysis. [AC76], published in the *Communications of the ACM*, was one of the first papers in the field to be published in the 'popular' computing press, while [Bar77] introduced interprocedural dataflow analysis and [Ryd83] looked at the problem of minimising the amount of re-analysis needed after altering a section of code.

The main problem in applying any of these ideas in an object-oriented context is that they all assume a language for which a control-flow graph can be easily constructed, annotated with dataflow information and then analysed. In Smalltalk, with all computations being performed by dynamically bound message sends, there is no easy way of constructing a control-flow graph from which useful dataflow information can be hung. It is possible to construct a control-flow graph for a single method, but the uncertainties introduced by unbound message sends means that little information can be associated with the variables in the graph.

One solution to the more basic problem of a lack of information within the code, as put forward by the proponents of type systems (e.g. [Joh86]), has been to assign type information to the variables, so providing some information which can be beneficial to the user (in providing feedback that the type information is compatible with the construction of the code) and to the compiler (to allow it to apply judicious optimisations to the code being compiled). However, as was shown in the preceding chapter (§3.8), the introduction of types into the untyped world of Smalltalk is fraught with problems. From a historical perspective this can be seen in the contrasting fortunes of the work carried out at the same time by Suzuki and Terada, and by Deutsch and Schiffman on providing improved implementations of Smalltalk-80. Suzuki's work with Terada [ST84] on an improved implementation of Smalltalk using, in part, type declarations, was immediately superseded by the work done by Deutsch and Schiffman [DS84] using only novel implementation techniques. In order to get a balanced picture of the possibilities of type systems, and type inference systems in particular, and the potential benefits of a dataflow analysis system, it was decided to design a fully autonomous dataflow analysis mechanism rather than try to integrate such a mechanism into a type inference system.

Although there is a large body of text dealing with dataflow and control-flow analysis, there has been no real work, from a theoretical perspective at least, on flow analysis in a dynamically bound imperative language. As such, the problem to be tackled over the next three chapters is not to define a system which analyses dataflow information, but instead to define a control-flow/dataflow model and analysis mechanism which can eventually be used to provide such information.

The rest of this chapter is taken up with the introduction of a core model covering the basic elements of the analysis mechanism. The following section provides an informal introduction to the system, describing the important features, as well as an overview of the analysis mechanism. Section 4.2 introduces the language to be analysed. This language is essentially the same as the language used in Chapter 3, and shares with it a basic adherence to Wolczko's semantic definition in [Wol88]. The main elements of the dataflow analysis system are introduced in §4.3, while the formal definition of the analysis procedure commences in §4.4. Section 4.5 deals with the important issue of recursion control, crucial in a system which 'executes' the code being analysed. The definition of the analysis system continues in Chapter 5, which covers the analysis of primitive methods, and is concluded in Chapter 6, which introduces the analysis of Smalltalk blocks and hence the analysis of Smalltalk control structures. Chapter 6 finishes off with a discussion of some of the issues raised by the dataflow analysis model.

4.1 An informal introduction to dataflow analysis

The dataflow analysis system described in this and the following two chapters is based upon the same basic technique as the type inference system described in the preceding chapter, namely abstract interpretation [CC77, MJ81]. This dataflow analysis system handles assignments and simulated method lookups like the type inference system, but instead of working over a domain of types it works over a domain of 'object' entities, or object abstractions. These object abstractions are designed to reflect the structure and properties of the objects which will be created at run time.

The type inference system described in Chapter 3 works by interpreting type trees, which are structurally similar to the parse trees they are derived from, over the domain of types. In contrast, the dataflow analysis system interprets parse trees directly and over the domain of object abstractions. So what is an object abstraction?

The entities which lie at the heart of the dataflow analysis system, these object abstractions, are constructed from two parts, an external *object identity* and an internal *object structure*. The object identity will be described shortly, but first, the object structure.

The object structure represents what is known about the internal structure and functionality of an object. Object structures come in three basic varieties:

class-derived object: This is the dataflow equivalent to the *object type* in TS. It represents a single object whose class, and hence structure and functionality, is known. As a result of this knowledge it is possible to keep track of any objects its instance variables may be referencing.[1]

predefined object: This is in actuality a *class-derived object* which is distinguished by its associated object identity. It is treated as a special case in order to help establish an order of object creation between the different kinds of object.

virtual object: This is the dataflow equivalent of an unbounded type variable. It represents an object about which nothing is known except that it exists in some form. As the code is analysed all virtual objects are considered to have been created before the class-derived objects which are created explicitly in the code.

The second part of the object abstraction is the *object identity*, based on the principle that each object has a unique identity. It is the object identity that distinguishes two objects, and which is used as the basis for testing object equivalence. For a virtual object, the assigned object identity is initially temporary, and is only made permanent when case analysis over the object dictates that it is unique.

As with the object structures, there are three alternative identities: the concrete identity, the predefined identity and the virtual identity.

concrete identity: This is the identity associated with objects 'created' within the code being analysed, for example through the new-object expression. As such it can be guaranteed that each concrete identity is unique because its point of creation has been observed. In addition, it is only ever associated with class-derived objects.

[1] With respect to issues relating to object encapsulation and the violation thereof, it should be pointed out that all the information about structure and instance variables is machine generated and is, strictly speaking, not meant for human consumption, nor for human manipulation.

predefined identity: This is the identity used to distinguish the predefined objects from other class-derived objects. By associating a predefined identity with a class-derived object, the object in question can be distinguished as a predefined object.

virtual identity: The virtual identity, like the virtual object, has an uncertain future. It is used exclusively for those objects which are involved in the simulated message send which commences the real dataflow analysis process with the analysis of a method. The objects involved at this stage are the initial self object, together with its instance variables, and the argument objects passed to the method. When a virtual object associated with a virtual identity is resolved into its potential cases, the identity may continue with a certain future, if that case yields a unique object from a virtual one, or it may disappear, if that case equates the object with one already resolved.

With respect to the dataflow analysis mechanism, the first objects to be considered are the predefined objects, constructed from a *predefined identity* and a *class-derived object* body. These represent objects assumed to be in existence by the analysis process before a piece of code — a method — is analysed. The virtual objects are assumed to come into existence at the point when the analysis starts. The majority of these objects, the parameters of the method and the instance variables of the self object, are made up from *virtual identities* and *virtual objects*. The self object itself is made from a *virtual identity*, to indicate its position in the order of creation, and a *class-derived object*, recording the known structure of the object. Finally, when an object is 'created' as part of the analysis process, it is represented by a *concrete identity* together with a *class-derived object*. It must be emphasised that objects are not created in the sense that objects are created at run-time, but rather that the coming into existence of an object at a given point in the code is recorded as part of abstract interpretation of the code.

So far we have seen how an object abstraction is made up of an object identity and the object itself. The object identity represents both a unique identity (or potentially unique identity) and an indication of when the object was created, while the object represents what is known about the structure of the corresponding real object. By combining the two parts into a suitable structure, and by allowing access to the object only via the object's identity, we can construct the following mapping.

$$\text{object identity} \;\rightarrow\; \text{object}$$

This constitutes our basic domain of interpretation. Next we must consider how to handle the interpretation over this domain, and for that we need an environment.

4.1.1 The Object space

For the type inference system, the environment used for the abstract interpretation kept track of the current estimated types for all the methods in the program, together with all the type constraints already derived. The dataflow analysis system uses as the basis of its environment, a structure which models the 'object space' that occurs at run time. The dataflow object space is designed to handle the currently active assignments between variables and objects, or rather between object references (variables and parameters) and object identities (that part of an object which uniquely identifies it).

The *object space* has three components, one to keep track of the object identities, one to keep track of the object references (also known as object names), and finally one to keep track of the objects themselves. The latter component represents the object abstractions described previously, linking the external object identities with their internal representation and functionality in the form of a map.

Objects	:	object identity	$\rightarrow$	object references
References	:	object reference	$\rightarrow$	object identity
Object map	:	object identity	$\rightarrow$	object

Both the *objects* and *references* fields are affected mainly by assignments and argument passing during message sends. The *object map*, on the other hand, tends to remain unchanged during these operations. Instead, it is altered during object creation and method lookups, when it may be necessary to resolve a virtual object into a set of cases, a process which will be described in due course.

The *objects* and *references* fields actually record the same information, but from different perspectives — the former records which variables and parameters are referencing a given object, while the latter links each variable and parameter with the object currently assigned to it. This redundancy has been included to help emphasise the importance attached to both sides of the relationship, especially that at any given point in the analysis process an object should know what is referencing it. One interesting property of this structure is that by recording the relationship between objects and variables from the object's point of view, incidents of aliasing can be observed directly simply by checking the object's entry in the *objects* field, rather than performing an exhaustive search through the *references* field.

The object space only forms part of the total dataflow environment, though it is clearly the most important part. The rest of the environment is made up of two components: *last*, used to record the result of the last expression processed, and the *method invocation record* (mir), used to keep track of which methods have been processed in the current context and so aid in the detection of recursive message sends.

Last	:	object identity returned by last expression
M.I.R.	:	list of message sends and states
Object Space	:	*described above*

The *mir* component, necessary for the handling of recursive message sends, works as follows. During the analysis of a method, when a message send is encountered, the states of the receiver and arguments are recorded, together with indications of which method is to be invoked and where it is being invoked from. In this way, if the same message send invokes the same method further along in the call sequence, it can be tested against the method invocation record. If the states of the receiver and arguments are comparable with those of the previous invocation, then the message send is interpreted as a recursive one from which no further information can be safely derived by continuing the analysis under the current context.

4.1.2 Code analysis

Code analysis takes the form of examining an expression and updating the dataflow environment accordingly. Unlike the type inference system, which works on a transformed type tree structure, here the analysis is done directly on the code. As a result, no alterations to the code are made. Instead all information is stored in the dataflow environment in a form which can be easily extracted, as the analysis progresses, for whatever purposes the ultimate beneficiary of the system desires.[2]

For example, the effect of analysing an assignment statement would be as follows. Consider the following assignment:

```
var := anObject
```

The effects on the dataflow environment would be:

Last	:	object identity referenced by anObject
M.I.R.	:	*unchanged*
Object Space	:	*altered according to the assignment*

The *last* field has been updated to the result of the assignment in the same way that the type of the expression was reduced to the type of the right-hand expression in the type inference system. The *method invocation record* has remained unaltered because the expression does not involve any message sends. Finally, the *object space* would be altered as follows:

Objects	:	**add** var to the entry relating to the object referenced by anObject. **remove** var from the entry relating to the object it originally referenced.
References	:	**alter** the entry for var so that it refers to the newly-assigned object.
Object map	:	*unchanged*

For the *objects* field, the variable var must be removed from its old object and added to its new object. Likewise for the *references* field, the entry for var must be updated to indicate its newly-assigned object. Throughout these alterations the object has been handled through its identity. Its underlying state has remain unchanged, and so the *object map* field remains unchanged.

This example shows two things. Firstly, it shows that the handling of the assignment by dataflow analysis is as good as the handling provided by the type system described in the preceding chapter. Secondly, and more importantly, it shows that this system does not fall into the trap caused by multiple assignments (§3.8.1). In effect, as far as assignments go, this dataflow analysis system provides the same basic service as the type inference system, but without the potential problems.

Message sends are handled by doing a simulated lookup. For example, the expression

[2]See §6.9 for a discussion of what sort of information can be extracted from the analysis process.

```
var msg: arg
```

would be handled by first reducing the receiver and the argument to single objects. This would be achieved through abstract interpretation. The simulated lookup would then progress depending on the form of the receiving object. In the case of a predefined or class-derived object the class of the receiver can be extracted directly. In the case of a virtual object, the object must be resolved into a set of possible *cases*.

A virtual object is one whose existence is known but about which nothing further can be said. It may be simply an instance of a class, unrelated to other objects in the system, and so cases must be formed for each class providing a corresponding method for the message in question. Alternatively, it may be one of the predefined objects, in which event cases must be constructed for all possible predefined objects which provide the necessary functionality. As each of these alternative cases is derived, the virtual object ceases to exist in its old form. Instead it either has its 'body' replaced by a new class-derived object, retaining its old identity for convenience, or, when it has been equated with a predefined object, all occurrences of the old virtual identity are replaced by the identity of the predefined object.

Once a suitable receiving object has been derived, the simulated lookup can continue. If the invoked method has been found, the dataflow environment is updated to include information derived from the message send. In the case of the above expression, the dataflow environment would be altered as follows:

Last	:	*of no interest — inaccessible by invoked method*
M.I.R.	:	**add** the message send and state to list
Object Space	:	*altered according to message send*

The *last* field has not been altered after analysing the receiver and argument because it will not be accessible from the invoked method. The invoked method will always start with an expression which makes no reference to the result of the last expression processed. The *method invocation record* has been updated to include the current message send. The information added to the list of message sends records not only the message selector and the class providing the invoked method, but also information about the state of the receiver and argument (or arguments).

Given the following method definition for the invoked method,

```
msg: par
    "An example method with only one parameter"
    ...
```

and providing that the message send is not a recursive one, the *object space*, for the duration of the invoked method, would be updated as follows:

Objects	:	**add** self and par to the entries relating to those objects referenced by the receiver, var, and arg respectively. **remove** self from the entry relating to the object it originally referenced.
References	:	**alter** the entry for self so that it refers to the receiving object. **add** the entry for par so that it refers to the argument object.
Object map	:	*unchanged*

The interpretation of the invoked method can then commence with the call-site specific context information, in other words, with the receiver and arguments linked to self and the method parameters. The analysis of the invoked method then follows the same pattern as the analysis of the method which invoked it. If the analysis of the method continues to the end of its body, it becomes necessary to return the result to the invoking method. To do this it is necessary to update the object space ready to return to the analysis of the invoking method, as well as to alter the other elements of the dataflow environment to reflect the result of the message send. First, the dataflow environment,

Last	:	identity of object returned as the result of the invoked method.
M.I.R.	:	**remove** the message send from the record, indicating a successfully completion.
Object Space	:	*altered according to the message return*

and then the object space:

Objects	:	**add** self to the entry relating to the original pre-message send self object. **remove** self from the entry relating to the receiving object of the message send.
References	:	**alter** the entry for self so that it refers to the original pre-message send self object.
Object map	:	*unchanged*

If, during the initial analysis of a message send, the method invocation record indicates that this send will result in a recursive invocation of a method, the processing of this particular case must eventually be terminated. This is necessary to guarantee that the analysis process as a whole will terminate. The precise criteria under which a recursive message send is singled out for termination will be discussed in due course.

As a final note before the system is defined formally, here is a quick overview of how the analysis process proceeds.

1. Pick a class

2. Pick a method from that class

3. Set up an initial dataflow environment

4. Begin analysing the method body. If an expression is a message send, it may potentially be a recursive message send. If it is, then terminate the analysis for this case, and instead follow up another case or start on another method. If it isn't a recursive send, continue the analysis through the invoked method. Once the analysis of the chosen method is complete, choose another method until all methods from that class have been analysed. Then move on to another class.

4.2 The language

The language we shall start off with is basically the same as the language used for the type inference system described in the preceding chapter. The language does not use inheritance, and so once again a 'program' consists of a simple mapping from class names (*ClassName*) to class definitions (*ClassBody*).

$$Program = ClassMap$$

$$ClassMap = ClassName \xrightarrow{m} ClassBody$$

As with the language defined in §3.3, there is an unspecified type, *Id*, upon which various other types are based. The reasoning behind using tagged versions of the underlying type *Id* is that these various types, which include class names, method selectors and variable names, share a common textual representation, but are distinguishable by the context of their uses.

$$ClassName :: Id$$

Once again, there is no 'program body', or expression whose evaluation defines the effect of the program (§3.3, page 28). Instead all methods are considered to be potential 'program bodies'. The process of analysis defined in this chapter treats each method individually, observing the effect of evaluating each method body with a minimum of *a priori* information. As a result, there is no need to restrict the analysis to some arbitrary predefined expression. However, this will inevitably result in numerous 'bogus' control paths being followed, with the relevant 'non-bogus' paths counting as a minority of the total number of paths analysed. It is the opinion of the author that such issues as directing the analysis process should ultimately be addressed with respect to a user-interface for such a system, and are therefore outside the intended scope of this thesis. However some of these issues are addressed in §7.3.

A class-definition consists of a set of instance variables, representing a template for the state of the class's instances. Unlike the language used for the type inference system of the preceding chapter, there is no need to include provisions for the typing of the code. The functionality of the class's instances is provided by the methods field — a mapping from method selectors to method definitions.

$$\begin{aligned} ClassBody :: Inst\_vars &: InstVar\text{-set} \\ Methods &: Selector \xrightarrow{m} Method \end{aligned}$$

$$Selector :: Id$$

There are two kinds of methods: the user-defined method, programmed by the user, and the primitive method, representing an interface to low level activities required by the language and its run-time system. The primitives also represent an interface between the language and the machine on which the 'program' is being executed.

$$\mathit{Method} = \mathit{UserMethod} \cup \mathit{PrimitiveMethod}$$

The user-defined method has the same basic structure as when it was defined for the type inference system (page 29), but without the provision for type information. The structure consists of a sequence of parameters, a set of temporary variables, and an expression — the method body — whose evaluation will yield the result of the method.[3] Naturally, there is a constraint that in the sequence of parameters, no single parameter may occur more than once. This is ensured by the record's invariant.

$$\begin{aligned} \mathit{UserMethod} :: \mathit{Params} &: \mathit{ParamId}^* \\ \mathit{Temps} &: \mathit{TempVar}\text{-set} \\ \mathit{Body} &: \mathit{Expression} \end{aligned}$$

where

$$\begin{aligned} &\mathit{inv\text{-}UserMethod}(\mathit{mk\text{-}UserMethod}(\mathit{params}, \mathit{temps}, \mathit{body})) \quad \triangleq \\ &\qquad \mathsf{card}\ \mathsf{elems}\, \mathit{params} = \mathsf{len}\, \mathit{params} \end{aligned}$$

The primitive method defined for this and the following two chapters differs from that defined for the preceding chapter. In this chapter we are interested in the actual action performed by a given primitive, rather than merely the type of object it returns. To this end each primitive must record the activity it is to perform. This is the purpose of the *Primitive* field, together with the *PrimitiveId* type. The form that this primitive identity takes can be seen in Chapter 5, which deals with the analysis of primitive methods.

$$\begin{aligned} \mathit{PrimitiveMethod} :: \quad \mathit{Params} &: \mathit{ParamId}^* \\ \mathit{Primitive} &: \mathit{PrimitiveId} \end{aligned}$$

$$\mathit{PrimitiveId} :: \mathit{Id}$$

The expressions defined for our initial language are identical to those defined for type inference system in the previous chapter except for the literal expression, where there is no need for an 'inferred' type field.

$$\begin{aligned} \mathit{Expression} = \mathit{ExpressionList} &\cup \mathit{Assignment} \cup \mathit{ObjectName} \\ &\cup \mathit{Message} \cup \mathit{NewObject} \cup \mathit{Literal} \end{aligned}$$

To recap, an expression list is simply a sequence of expressions:

$$\mathit{ExpressionList} = \mathit{Expression}^*$$

[3] As with the language presented in §3.3, there is no explicit return expression. Hence all implicit '↑self' expressions are assumed to have been made explicit.

An assignment consists of a left-hand variable and a right-hand expression. The effect of this expression is to assign the result of the right-hand expression to the left-hand variable.

$$\begin{array}{lrcl} \mathit{Assignment} :: & \mathit{LHS} & : & \mathit{AssVar} \\ & \mathit{RHS} & : & \mathit{Expression} \end{array}$$

The definitions and grouping of the various variables, parameters and object references (i.e. self) are the same as for the language defined in §3.3 and shown symbolically in Figure 3.1 (page 30).

An assignable variable (*AssVar*) can be either a temporary variable (*TempVar*) or an instance variable (*InstVar*):

$$\mathit{AssVar} = \mathit{TempVar} \ \cup \ \mathit{InstVar}$$

$$\mathit{TempVar} :: \mathit{Id}$$

$$\mathit{InstVar} :: \mathit{Id}$$

$$\mathit{ParamId} :: \mathit{Id}$$

VarId brings together all the variable forms (temporary and instance) and the parameters (*ParamId*):

$$\mathit{VarId} = \mathit{TempVar} \ \cup \ \mathit{InstVar} \ \cup \ \mathit{ParamId}$$

Finally *ObjectName* combines the *VarId*s with the self reference.

$$\mathit{ObjectName} = \mathit{VarId} \ \cup \ \{\mathit{self}\}$$

The message expression is made up of a receiver expression, whose evaluation will determine the recipient of the message, a message selector, which will determine the method to be invoked, and a sequence of argument expressions, which, when evaluated, will provide the arguments, if any, required by the invoked method. In addition there is a marker field, to distinguish this message send from other message sends in the same method. The marker field is used in the detection of recursive message sends (*process_message*, page 119, and §4.5), and only has to be unique within its defining method's body.

$$\begin{array}{lrcl} \mathit{Message} :: & \mathit{Rcvr} & : & \mathit{Expression} \\ & \mathit{Sel} & : & \mathit{Selector} \\ & \mathit{Args} & : & \mathit{Expression}^{*} \\ & \mathit{Marker} & : & \mathit{Id} \end{array}$$

The new-object expression consists of a class name, and will, when executed, result in a new instance of that class. The object creation expression could have been treated as a primitive, but has been singled out, in part, to aid clarity.

$$\mathit{NewObject} :: \mathit{Class} : \mathit{ClassName}$$

The final expression, the literal, differs from the version defined in §3.3 only in that there is no need for a 'type' field. Once again, with the exception of the symbols for the 'undefined' object, nil,[4] the 'true' object, true, and the 'false' object, false, the form of the literal has been defined as a 'catch-all' for all suitable literal expressions which may eventually be included. The current system defined in this chapter has no provisions for numbers (integers in particular) or collection-type objects, two forms commonly represented by literal expressions. Issues relating to the handling of these forms of object will be discussed in §6.8.

$$Literal :: Value : \{nil, true, false\} \cup \ldots$$

The all-important dataflow environment will be described in the next section. It is this environment which is used to store information derived from the analysis of code constructed from the abstract syntax defined above.

4.3 The dataflow environment

How does the dataflow analysis proceed? How is the information extracted? Where is the information stored? Although these questions will be answered during the rest of this chapter, the basis of their answers lies in the definition of the dataflow environment. The structure of the dataflow environment controls what information is to be extracted, and how it is stored. Its importance is equivalent to that of the type environment (*TypeEnv*, page 38) in the preceding chapter. The basic structure of the environment has already been introduced informally in §4.1.1, but to recap, the environment is made up from three components: 'last', 'mir' and the object space, of which the latter is the most important. The object space is the primary structure used by the dataflow analysis system to store the information extracted by the analysis mechanism. It is within this structure that the crucial abstraction lies, namely from the actual run-time object space to a static abstraction which provides just enough information to facilitate an analysis and prediction of the run-time system at compile time.

The form the abstraction takes is made up from three basic elements: the *object reference* (i.e. variables, parameters and self), the *object identity*, and the *object*. The *object reference* is any one of the components that makes up the *ObjectName* expression. The *object identity* represents an identity which distinguishes an object from all other objects, and which, with one exception, lasts for the lifetime of that object. Finally the *object* represents everything that is known about the internal structure of that particular object. *Objects* and *object identities* always occur in pairs.

As has already been indicated in §4.1.1, within this basic form of the abstraction, there are distinctions to be made between different kinds of object identities and the form of the

[4]In Smalltalk-80, nil is a pseudo variable used to access the special 'predefined' 'undefined' object (instance of class UndefinedObject), used, amongst other things, as an initial assignment to the instance variables of a newly created object. Likewise, true and false are pseudo variables used to access predefined instances of classes True and False respectively. Throughout this chapter, nil will be used as both the syntactic literal expression for the 'undefined' object, as well as its predefined object identity. Uses of nil for the 'undefined' object and its identity should be distinguishable by their context, as should uses of the VDM nil in those sections dealing with recursion control.

information recorded for a particular object. These distinctions cover notions of when an object was first known to exist, and whether anything is actually known about the internal structure of an object.

4.3.1 *DfEnv* — the dataflow environment

The definition of the dataflow environment lies at the heart of the dataflow analysis system. The three elements which go to make up the environment are the object space, σ,[5] the method invocation record, *mir*,[6] and *last*, a record of the 'result' of the last expression analysed. Throughout the rest of this chapter, dataflow environments will generally be referenced by the greek letter δ, or variations thereof (e.g. δ', δset for a set of environments, etc.).

$$
\begin{array}{lrcl}
DfEnv :: & \sigma & : & ObjectSpace \\
 & mir & : & MethodInvocationRecord \\
 & last & : & Oop
\end{array}
$$

DfSet and *DfResult* define the form of the result produced by the analysis process.

$$DfSet = DfEnv\text{-set}$$

$$DfResult = ClassName \xrightarrow{m} Selector \xrightarrow{m} DfSet$$

The mapping of *DfResult* provides a set of dataflow environments (*DfSet*) for each method defined in each class. This is similar to the method type map used by the type inference system (§3.4.3, page 37). Therefore, for a given class and a given method selector, the result of the analysis process is a set of dataflow environments, each one corresponding to a possible control-flow path through that method, and each one containing the following information:

σ, the object space: The state of the object space on leaving the method, or the state of the object space on detecting a recursive message send.

***mir*, the method invocation record:** Either a record of the original message send which initiated the analysis of the current method, if the analysis proceeded to the end of the method, *or* a sequence of message sends leading up to a recursive message send, if the analysis was terminated early due to the detection of recursion.

***last*:** The result of the last expression analysed. In the case where the analysis continues to the end of the method, this represents the result of that method.

The method invocation record (*mir*) is made up of a sequence of *Recurs*, each *Recurs* being a packet of information recording the state of a particular message send. Its primary use is in

[5]The use of σ to indicate the object space follows the notation used by Wolczko [Wol88], who uses σ to indicate the object memory when defining the semantics of a language very close to the one being used here. Likewise *Oop* for object pointer or object identity, and *Object* for the objects themselves have also been borrowed from Wolczko's notation.

[6]Coincidentally, mir is also Russian for peace, as well as being the name of a (former Soviet) space station.

detecting recursive, and even mutually recursive, message sends. This is necessary to prevent repeated, and unending, analysis of the same method code. An initial definition of *Recurs* can be found in §4.5, while in §6.7 the definition is extended to handle blocks.

$$MethodInvocationRecord = Recurs^*$$

As has already been mentioned, the object space, and the components recorded in it, are the key elements in the dataflow analysis system, defining, as they do, the abstract domain over which the analysis takes place. In an attempt to model, at an abstract level, the workings of the run-time system, the object space consists of the following three mappings:

$$\begin{array}{rcl} \text{Object identity} & \rightarrow & \text{Object references} \\ \text{Object reference} & \rightarrow & \text{Object identity} \\ \text{Object identity} & \rightarrow & \text{Object details} \end{array}$$

'Object references' are any syntactic ('variable') references to an object. In other words, instance and temporary variables, parameters and self, which the abstract syntax combines under the single type *ObjectName*. The uses of the singular and plural in the above list can be explained as follows: a single object may be referred to by several 'variables' (references) at the same time, while a given 'variable' can only refer to one object at a time. Object identities are handled by *Oop*, with *Object* corresponding to an object's 'details'. This leads to the following definition of the object space:

$$\begin{array}{rrcl} ObjectSpace :: & Objects & : & Oop \xrightarrow{m} ObjectName\text{-set} \\ & Refs & : & ObjectName \xrightarrow{m} Oop \\ & OopMap & : & Oop \xrightarrow{m} Object \end{array}$$

where

$$\begin{array}{l} inv\text{-}ObjectSpace(mk\text{-}ObjectSpace(objects, refs, oop\_map)) \quad \triangleq \\ \quad \{nil, true, false\} \subseteq PredefinedOop \\ \quad \wedge \{nil \mapsto mk\text{-}ClassDerivedObject(UndefinedObject, \{\ \})\} \in oop\_map \\ \quad \wedge \{true \mapsto mk\text{-}ClassDerivedObject(True, \{\ \})\} \in oop\_map \\ \quad \wedge \{false \mapsto mk\text{-}ClassDerivedObject(False, \{\ \})\} \in oop\_map \\ \quad \wedge \bigcup \mathrm{rng}\, objects = \mathrm{dom}\, refs \\ \quad \wedge \mathrm{rng}\, refs = \mathrm{dom}\, objects \\ \quad \wedge \forall oop \in \mathrm{dom}\, objects \cdot \forall ref \in objects(oop) \cdot refs(ref) = oop \\ \quad \wedge \forall ref \in \mathrm{dom}\, refs \cdot ref \in objects(refs(ref)) \end{array}$$

In §4.1.1, with the original introduction of the object space, it was mentioned that the *Objects* map and the *Refs* map both record the same information. This property can be seen in the invariant defined above for the object space, where the set of variables and parameters recorded for all the objects in the *Objects* field should be exactly the same as the set of object references covered by the *Refs* field. Likewise, the set of objects referenced by variables and parameters, and recorded in the *Refs* field should be the same as the set of objects covered by the *Objects* field.

The inclusion of this redundancy has been made specifically for one reason, namely to emphasise the importance of recording information relating to specific objects. In fact the *Objects* field is not used directly at all throughout the analysis process, but the information it records is important in the detection of aliasing between an object and several variables. Although this information could be derived from the *Refs* field, its presence in the *Objects* field signifies the importance of the object and its identity, and the knowledge that can be associated with it during the analysis process.

The invariant on the object space also represents the minimum amount of information assumed prior to the analysis of a method. The invariant defined above ensures that there is an 'undefined' object (instance of class UndefinedObject), with an identity indicated by nil. This object is required by the semantics of the language, where any uninitialised variables are initialised to nil when they first come into scope. This covers, for example, the instance variables of a newly-created object before anything has been assigned to them, or the temporary variables of an invoked method at the moment of invocation. In addition, true (instance of class True) and false (instance of class False) are both provided as predefined objects. They are included, both as a convenience, and, ultimately, as a necessity due to their eventual use with the test for object equivalence in §5.3.4.

As for other potential predefined objects, although classes are not treated as objects in this language, as they are in Smalltalk-80, there is nothing, in principle, to stop them being handled as predefined objects in the object space. However for this to work, the classes must be treated as static objects, subject to interrogative messages only, and have their own 'meta' class to define their functionality. It may also be necessary to define a new form of object (in the same way that a new kind of object is required for blocks (Chapter 6)) to record the special details possessed by a class.

The object identity comes in three variations, each representing an initial property of the object in question. The three variations are the predefined object identity, the concrete identity and the virtual object identity. Respectively these cover objects assumed to exist before the commencement of code analysis, objects whose creation occurs during the code analysis, and finally those objects whose existence is determined to begin with the start of the code analysis, but whose identity is not fully resolved until later.

$$Oop = PredefinedOop \cup ConcreteOop \cup VirtualOop$$

A predefined object is an object whose existence is assumed prior to the start of the analysis of a given method. In the current system, the only predefined objects are the 'undefined' object, the 'true' object and the 'false' object, and hence the only predefined object identities are nil, true and false.

$$PredefinedOop :: Id$$

A concrete object is one whose creation occurs during the analysis of a method. Because its point of origin is known, a concrete object identity can be made unique with respect to other objects created before and after it.

ConcreteOop :: *Id*

A virtual object is one whose existence is known, but whose uniqueness of identity has yet to be determined. Virtual objects only occur at the start of the code analysis for a particular method. They are used for the arguments of the method being analysed, as well as for the instance variables of the receiving object. The main uncertainty of a virtual object, in general, is its class, and this will be discussed when the virtual object itself is defined. As far as the virtual object identity is concerned, the problem of uniqueness only becomes an issue when testing for equivalence between objects. The problem lies in allowing for cases where an unresolved virtual object is assumed to be one of the predefined objects, or is assumed to be one of the other virtual objects whose identity has already been resolved.

VirtualOop :: *Id*

The other half of the identity/object pair, *Object*, which contains the information known about a particular object at a given time, comes in only two varieties. The way in which this analysis system has been constructed, there are only two 'states of knowledge' that an object can be in. An object can either be virtual (*VirtualObject*), when nothing is known about it, or alternatively its class, and hence its structure can be known, in which case it is described as being 'class-derived' (*ClassDerivedObject*). The certainty of this distinction lies with the way in which virtual objects are resolved into class-derived objects through the analysis process. This occurs during a message send to a virtual object, when the form of the receiving object must be fixed for the analysis to progress. The form is fixed for a virtual object by generating and then following up different cases, each of which has the virtual object replaced by a suitable class-derived object.

$$Object = ClassDerivedObject \ \cup \ VirtualObject$$

A class-derived object carries around with it not only its class, but also the instance variables' structure derived from that class. By applying a suitable tagging strategy to the instance variables it should possible to distinguish between variables with the same original textual name in the *objects* and *references* mappings of the object space.[7] As such, it is necessary to re-tag the instance variables each time a new class-derived object is created. An example of this tagging can be seen in the definition of *prepare_self* (page 103).

$$\begin{aligned} ClassDerivedObject \ :: \quad Class &: ClassName \\ Inst\_vars &: InstVar\text{-set} \end{aligned}$$

A virtual object carries around a set of class-derived objects representing the possible class-derived objects into which it can be resolved.

VirtualObject :: *ClassDerivedObject*-set

To recap, we have a dataflow environment made up from the centrally important object space, together with a record of the result of the last expression and a record of past message sends to be used in the detection of recursion. The object space has two maps to link object references

[7]Such a tagging strategy might involve the addition of enough 'primes' ($'$) to the end of the variable name to make it unique within the object space.

with object identities in both directions. A third map links object identities with object 'details'. Object identities come in three varieties: predefined, virtual and concrete. Object details are either virtual (unknown) or class-derived (known).

The intention is that predefined objects, those objects assumed to exist already, will be distinguished by their 'predefined' identity. Naturally their details will come under the class-derived category. The receiver, its instance variables and the arguments of the initial method invocation will be virtual by identity and virtual by detail, with the exception being the receiver itself. The receiver will be virtual by identity, in keeping with its common creation point with the other virtual objects, but its details will be of the class-derived variety. All other objects explicitly created by the code under analysis will be concrete and class-derived.

The following section defines and describes a set of functions used for the creation and general upkeep of the object space. They range from the simple assignment of an object to a variable or object reference, through to the creation and addition of a virtual object.

4.3.2 Object Space functions

The following functions provide a convenient interface for updating the various fields of the object space, and so ensure that the space retains its integrity. This is particularly important for the *Objects* and *Refs* fields, which effectively duplicate the same information.

The object space can be thought of as consisting of two distinct parts, an external part relating to the visible parts of the space — the object identities and the object references (*Objects* and *Refs*) — and an internal part relating to the internal details of the objects within the space (*OopMap*). The object references and the object details reside wholly in their respective parts, while the object identities provide a connection between the two.

$$ref \mapsto oop \mapsto object$$

Reflecting this distinction, the object space functions provide a separation of functionality between updating the external part — mappings between object references and object identities — and altering the internal part — the mapping between object identities and object details.

The first two functions, which form a pair, provide the necessary functionality for dealing with the external part of the object space. The *assign* function updates the object space to take account of an assignment of an object identity to a variable or parameter (i.e. to an object reference). This involves updating the *Object* and *Refs* fields.

The function receives, as its parameters, an object reference (*ref* — the variable or parameter being assigned to), an object identity (*oop* the object being assigned), and an object space, σ, to be updated according to the assignment. The function deals with each field of the object space in turn, combining the results to return an updated object space. Updating the *Objects* map with the mapping

$$oop \mapsto ref$$

is dealt with by a separate function, the second function of the pair, *update_Objects_map*. The operation has been separated into two functions because of the complexity introduced when working with both new and old objects in the *Objects* field, while maintaining a consistent set of maps. For the *Refs* field all that is required is for the new mapping

$$ref \mapsto oop$$

to be added to the map, overwriting any old entry for *ref* in the case of a reassignment. The final result is created by combining the two updated fields with an unaltered object map (*oop_map*) to form a recombined and updated object space.

$$
\begin{aligned}
&assign : ObjectName \times Oop \times ObjectSpace \to ObjectSpace\\
&assign(ref, oop, \sigma) \quad \triangleq\\
&\qquad \mathsf{let}\ objects = update\_Objects\_map(ref, oop, Objects(\sigma))\\
&\qquad\quad \mathsf{in}\\
&\qquad \mathsf{let}\ refs = Refs(\sigma) \dagger \{ref \mapsto oop\}\\
&\qquad\quad \mathsf{in}\\
&\qquad \mathsf{let}\ oop\_map = OopMap(\sigma)\\
&\qquad\quad \mathsf{in}\\
&\qquad mk\text{-}ObjectSpace(objects, refs, oop\_map)
\end{aligned}
$$

Update_Objects_map takes the *ref*/*oop* pair from the *assign* function, together with *objects*, the *Objects* field from the object space, and returns *objects* updated with the new assignment. This task can be broken down into two parts. Firstly adding the mapping from the object identity to the object reference, and secondly, removing the object reference from any existing mapping left over from a previous assignment.

In adding the new mapping, two cases immediately arise. Either this is the first assignment involving this object, or this object has been assigned before. In the former case, a new mapping is required, while in the latter, the entry for the object must be altered to include the new object reference. This problem is sorted out by testing if the object identity is already in the domain of the *Objects*' mapping, and creating a suitable maplet accordingly (new maplet for a new object, updated maplet for an old one). The resulting maplet is named *new_object*.

The object reference, *ref*, may have already been assigned to before, in which case it will have to be removed from the mapping relating to such an assignment. This is handled by looking for the existence of *ref* in the range of any of the old mappings. If there is such a mapping, the relevant maplet is extracted and the object reference is removed from its range. The resulting 'reduced' maplet is named *old_object*. If the object reference is new to the object space an empty maplet is used instead.

Finally the results from the two tasks are combined with *objects*, ready to be returned to the *assign* function. By overwriting *objects* with the two updated maplets, the original pre-assignment state has been removed and replaced with a state consistent with the new assignment.

$$update\_Objects\_map : ObjectName \times Oop \times Oop \xrightarrow{m} ObjectName\text{-set} \\ \rightarrow Oop \xrightarrow{m} ObjectName\text{-set}$$

$$\begin{array}{l} update\_Objects\_map(ref, oop, objects) \quad \triangleq \\ \quad \text{let } new\_object = \text{if } oop \in \text{dom}\, objects \\ \qquad\qquad\qquad \text{then } \{oop \mapsto objects(oop) \cup \{ref\}\} \\ \qquad\qquad\qquad \text{else } \{oop \mapsto \{ref\}\} \\ \quad\; \text{in} \\ \quad \text{let } old\_object = \text{if } \exists\, old\_oop \in Oop \cdot ref \in objects(old\_oop) \\ \qquad\qquad\qquad \text{then let } old\_oop \in Oop \cdot ref \in objects(old\_oop) \\ \qquad\qquad\qquad\qquad \text{in} \\ \qquad\qquad\qquad\quad \{old\_oop \mapsto objects(old\_oop) - \{ref\}\} \\ \qquad\qquad\qquad \text{else } \{\,\} \\ \quad\; \text{in} \\ \quad objects \dagger old\_object \dagger new\_object \end{array}$$

The following function, *add_virtual_objects*, is the first to deal with the internal part of the object space. It takes a set of variables or parameters and 'initialises' them by assigning a new virtual object to each. It takes as its parameters a set of object references, *var_id_set* and an object space, σ, and returns an updated object space including the new assignment for each *var_id* in *var_id_set*:

$$\begin{array}{cc} var\_id \mapsto virtual\_oop & virtual\_oop \mapsto virtual\_object \\ (external) & (internal) \end{array}$$

The function is defined recursively, taking a single *var_id* at a time until the terminating condition, an empty *var_id_set*, has been satisfied. For a given *var_id*, first the new virtual object must be prepared. This task is performed by *prepare_virtual_object*, which takes a copy of the object space and returns a new virtual object identity together with an updated object space with the virtual identity/virtual object maplet already in place. With the object details in place in the *OopMap* of the object space, the assignment between the object reference and the object identity can be added using the *assign* function. Finally the current *var_id* must be removed from the set to be processed, ready for the function to call itself recursively with the reduced set.

$add\_virtual\_objects : VarId\text{-set} \times ObjectSpace \rightarrow ObjectSpace$

$add\_virtual\_objects(var\_id\_set, \sigma) \quad \triangleq$

if $var\_id\_set = \{\,\}$

then σ

else let $var\_id \in var\_id\_set$

in

let $(voop, \sigma') = prepare\_virtual\_object(\sigma)$

in

let $\sigma'' = assign(var\_id, voop, \sigma')$

in

let $var\_id\_set' = var\_id\_set - \{var\_id\}$

in

$add\_virtual\_objects(var\_id\_set', \sigma'')$

Prepare_virtual_object simply takes a copy of the current object space and returns a fresh virtual object identity, together with an updated object space containing an entry for the new identity and new virtual object in the *OopMap* field. The new virtual object identity is provided by *new_virtual_object_id*, which is guaranteed to supply a new unique virtual identity, based on the object identities currently being used in the object space. Next a new virtual object is created in two stages. The body of the virtual object is actually a set of class-derived objects, representing the possible classes of which the virtual object may be an instance. This set is made up from all the classes defined for the program under analysis, which, for the purposes of the analysis system, is assumed to be globally visible as *program*. The set is so constructed that case analysis may be performed by separating out the individual class-derived objects, and constructing ‘cases’ out of the resulting objects. The second step simply tags the resulting set to make it a virtual object. Finally the required maplet from the new identity to the new object is constructed, ready to be added to the *OopMap* field of the object space, which is returned together with the new object identity.

$prepare\_virtual\_object : ObjectSpace \rightarrow Oop \times ObjectSpace$

$prepare\_virtual\_object(\sigma) \quad \triangleq$

$\quad \mathsf{let}\ new\_voop = new\_virtual\_object\_id(\sigma)$

$\quad\quad \mathsf{in}$

$\quad \mathsf{let}\ v\_cdos = \{mk\text{-}ClassDerivedObject(class, Inst\_vars(program(class))) \mid class \in \mathsf{dom}\, program\}$

$\quad\quad \mathsf{in}$

$\quad \mathsf{let}\ v\_obj = mk\text{-}VirtualObject(v\_cdos)$

$\quad\quad \mathsf{in}$

$\quad \mathsf{let}\ oop\_maplet = \{new\_voop \mapsto v\_obj\}$

$\quad\quad \mathsf{in}$

$\quad (new\_voop, \mu(\sigma, OopMap \mapsto OopMap(\sigma) \cup oop\_maplet))$

The following two specifications represent two functions which are assumed to provide an unending supply of unique object identities, virtual object identities in the case of *new_virtual_object_id*, and concrete object identities in the case of *new_concrete_object_id*. Both these functions are assumed to be deterministic.

$new\_virtual\_object\_id : ObjectSpace \rightarrow VirtualOop$

$new\_virtual\_object\_id\ (\sigma : ObjectSpace)\ new\_voop : VirtualOop$

$\mathsf{post}\ new\_voop \notin \mathsf{dom}\, OopMap(\sigma)$

$new\_concrete\_object\_id : ObjectSpace \rightarrow ConcreteOop$

$new\_concrete\_object\_id\ (\sigma : ObjectSpace)\ new\_oop : ConcreteOop$

$\mathsf{post}\ new\_oop \notin \mathsf{dom}\, OopMap(\sigma)$

The final function in this section, *add_nil_objects*, performs a similar duty to *add_virtual_objects*. It takes a set of variables or parameters, *var_id_set*, together with an object space, σ, and assigns nil, the 'undefined' object, to each one. It is defined recursively in the same manner as *add_virtual_objects*. A single *var_id* is taken from the *var_id_set* and assigned the predefined identity nil (see *ObjectSpace* invariant, page 92), using the *assign* function.

$$
\begin{array}{l}
\textit{add\_nil\_objects} : \textit{VarId}\text{-set} \times \textit{ObjectSpace} \rightarrow \textit{ObjectSpace} \\
\textit{add\_nil\_objects}(\textit{var\_id\_set}, \sigma) \quad \triangleq \\
\quad \text{if } \textit{var\_id\_set} = \{\,\} \\
\quad \text{then } \sigma \\
\quad \text{else let } \textit{var\_id} \in \textit{var\_id\_set} \\
\qquad\quad \text{in} \\
\qquad \text{let } \sigma' = \textit{assign}(\textit{var\_id}, \textit{nil}, \sigma) \\
\qquad\quad \text{in} \\
\qquad \text{let } \textit{var\_id\_set}' = \textit{var\_id\_set} - \{\textit{var\_id}\} \\
\qquad\quad \text{in} \\
\qquad \textit{add\_nil\_objects}(\textit{var\_id\_set}', \sigma')
\end{array}
$$

As a summary, here is a list of the key functions introduced in this section:

$\textit{assign} : \textit{ObjectName} \times \textit{Oop} \times \textit{ObjectSpace} \rightarrow \textit{ObjectSpace}$

This assigns an object identity to an object reference, updating the *Objects* and *Refs* maps consistently, and so ensuring the integrity of the object space.

$\textit{add\_virtual\_objects} : \textit{VarId}\text{-}\mathbf{set} \times \textit{ObjectSpace} \rightarrow \textit{ObjectSpace}$

This function takes a set of variables and creates and assigns a virtual identity/object to each. It delegates responsibility for the creation of each virtual identity/virtual object pair to *prepare_virtual_object*, while leaving the linking of the variables and new object identities to the *assign* function.

$\textit{add\_nil\_objects} : \textit{VarId}\text{-}\mathbf{set} \times \textit{ObjectSpace} \rightarrow \textit{ObjectSpace}$

This function takes a set of variables and uses the *assign* function to link them to the predefined nil object.

$\textit{new\_virtual\_object\_id} : \textit{ObjectSpace} \rightarrow \textit{VirtualOop}$

This (deterministic) function provides a source of unique virtual object identities.

$\textit{new\_concrete\_object\_id} : \textit{ObjectSpace} \rightarrow \textit{ConcreteOop}$

This (also deterministic) function provides a source of unique concrete object identities.

4.4 Analysing the dataflow

The analysis process handles each class in turn, and each (user-defined) method in each class. Each treatment is done in isolation of any previous analysis, and so the order of the analysis is not significant. As has already been indicated, the result of the analysis is tied to each individual method, the result being a set of dataflow environments, one for each potential control path through that method. The overall outcome, therefore is of the form

$$\textit{ClassName} \rightarrow \textit{Selector} \rightarrow \textit{DfEnv}\text{-set}$$

which is represented by the types *DfResult* and *DfSet* (page 91).

4.4.1 Analysing a program

The entry point into the analysis process is the function *DfProgram*, which selects each class in turn and hands control over to *DfClass*. Because the state of the program under analysis will remain unchanged by the process, as an aid to clarity the program will be assumed to be a global constant, and hence universally visible, for the remainder of system.

$$
\begin{array}{l}
\mathit{DfProgram} : \mathit{Program} \rightarrow \mathit{DfResult} \\
\mathit{DfProgram}\,[\![\mathit{program}]\!] \quad \triangleq \\
\qquad \{\mathit{class} \mapsto \mathit{DfClass}\,[\![\mathit{class}]\!] \mid \mathit{class} \in \mathsf{dom}\,\mathit{program}\}
\end{array}
$$

DfClass acts as an intermediary function, invoking *DfMethod* which represents the real start of the analysis process. *DfClass* takes the class passed by *DfProgram*, and prepares an initial set of dataflow environments. This initial set, provided by *prepare_initial_env*, represents information which is common to, and needed by, the methods to be analysed for one particular class.

With the initial dataflow environments constructed, it is then possible to start analysing the methods defined for the current class. Each user-defined method is analysed, within the context of each of the dataflow environments constructed. Each call of *DfMethod* will return a set of dataflow environments, and so it is necessary to take the distributed union over each set produced by each initial dataflow environment.

$$
\begin{array}{l}
\mathit{DfClass} : \mathit{ClassName} \rightarrow \mathit{Selector} \xrightarrow{m} \mathit{DfSet} \\
\mathit{DfClass}\,[\![\mathit{class}]\!] \quad \triangleq \\
\qquad \mathsf{let}\ \mathit{class\_body} = \mathit{program}(\mathit{class}) \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{let}\ \delta\mathit{set} = \mathit{prepare\_initial\_env}(\mathit{class}) \\
\qquad\quad \mathsf{in} \\
\qquad \{\mathit{sel} \mapsto \bigcup\,\{\mathit{DfMethod}\,[\![\mathit{Methods}(\mathit{class\_body})(\mathit{sel})]\!]\,(\delta, \mathit{sel}) \mid \delta \in \delta\mathit{set}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{sel} \in \mathsf{dom}\,\mathit{Methods}(\mathit{class\_body}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \wedge \mathit{Methods}(\mathit{class\_body})(\mathit{sel}) \in \mathit{UserMethod}\}
\end{array}
$$

For obvious reasons, the major part of the construction of an initial dataflow environment, lies in dealing with the object space. The whole process of creating a set of initial environments is controlled by *prepare_initial_env*, which is described next. The need for a set of environments rather than a single one comes from the need to handle special cases arising from any predefined objects.

The initial form of the object space must contain all the predefined objects, nil, true and false, as well as the self object which will act as the receiver for the methods to be analysed. Inserting the predefined objects yields the initial space, *basic*σ. Adding the receiver, or self object yields the set of object spaces indicated above, needed to handle those cases where the current class coincides with the class of one of the predefined objects. As a result, with the present assumptions, this set will contain a single element for all cases except the UndefinedObject, True and False cases, when there will be two elements. This set of object spaces is constructed by the *prepare_self* function, which will be defined shortly.

The set of object spaces constructed so far must now be turned into a set of dataflow environments, one for each object space. A single environment is made up of an object space taken from the set already constructed, an empty sequence of method invocation records (empty because no methods have been invoked yet), and for the *last* field, nil, the 'undefined' object. The state of the *last* field is of no great importance because it is has no meaning before any code has been processed.

$$
\begin{array}{l}
\mathit{prepare\_initial\_env} : \mathit{ClassName} \longrightarrow \mathit{DfSet} \\
\mathit{prepare\_initial\_env}(\mathit{class}) \quad \triangleq \\
\qquad \mathsf{let}\ \mathit{basic}\sigma = \mathit{mk\text{-}ObjectSpace}(\{\,\}, \{\,\}, \\
\qquad\qquad\qquad\qquad \{\mathit{nil} \mapsto \mathit{mk\text{-}ClassDerivedObject}(\mathit{UndefinedObject}, \{\,\}), \\
\qquad\qquad\qquad\qquad\ \mathit{true} \mapsto \mathit{mk\text{-}ClassDerivedObject}(\mathit{True}, \{\,\}), \\
\qquad\qquad\qquad\qquad\ \mathit{false} \mapsto \mathit{mk\text{-}ClassDerivedObject}(\mathit{False}, \{\,\})\} \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{let}\ \sigma\mathit{set} = \mathit{prepare\_self}(\mathit{class}, \mathit{basic}\sigma) \\
\qquad\quad \mathsf{in} \\
\qquad \{\mathit{mk\text{-}DfEnv}(\sigma, [\,], \mathit{nil}) \mid \sigma \in \sigma\mathit{set}\}
\end{array}
$$

Preparing the receiver, or self object, is the task of *prepare_self*. Put simply, it must create a new class-derived object, a new object identity, assign this identity to the pseudo-variable self, and return an updated object space with all this information in place. It must also handle special cases arising from predefined objects.

Prepare_self takes a class name and an object space, and in return it provides a set of updated object spaces. In the simplest, most general case it returns a singleton set of object spaces. Special cases arise when the class being analysed coincides with the class of one of the predefined objects. The extra information known about a predefined object can then be utilised in its own special case, providing more specific information about its potential use.

As with the custom creation of a virtual object for a variable in the function *add_virtual_objects*, the process of creating the self object can be broken down into the creation of the object identity and object body, and the assignment of the object identity to the pseudo-variable self. The creation of the object and its identity is done first, with the identity being provided by the function *new_virtual_object_id*. The identity is 'virtual' only to place it in the order of object creation between the predefined objects and the later concrete objects. Apart from the self object which has a 'virtual' identity and a 'concrete' class-derived object structure, all the other virtual object identities will initially have virtual object structures. The first example of these latter virtual objects are the instance variables of the self object.

After the creation of the object identity, *self_oop*, and object structure, *self_object*, which is a class-derived object with its instance variables suitably tagged by *tag_inst_vars* from the class definition, the object map, *OopMap*, can be updated to include the pair as a new maplet. The *assign* function is then used to link the new object identity to the object referent, self, concluding the activities directly involved in initialising and assigning the new self object. However, one more stage in the initialisation process must be done, namely the assignment of virtual objects to the self object's instance variables. This is achieved by using the *add_virtual_objects* function (page 98). The instance variables are initialised to virtual objects to take account of the full range of objects which they may reference (in contrast to newly-created objects whose instance variables are initialised to nil).

The process up to the initialisation of the instance variables has yielded a single updated object space, σ'''. If the class of the self object coincides with any of the predefined objects, special-case object spaces must be set up to make use of the additional information. These 'predefined' object spaces (*predefined*σ*s*) are constructed by using the *assign* function to link self to any objects of the same class present in the original object space, σ.[8] Finally, the resulting object spaces are returned as a set, as required by *prepare_initial_env*.

$$prepare\_self : ClassName \times ObjectSpace \rightarrow ObjectSpace\text{-set}$$

$$
\begin{array}{l}
prepare\_self(self\_class, \sigma) \quad \triangleq \\
\quad \mathsf{let}\ self\_oop = new\_virtual\_object\_id(\sigma) \\
\quad\quad \mathsf{in} \\
\quad \mathsf{let}\ inst\_vars = tag\_inst\_vars(self\_class, \sigma) \\
\quad\quad \mathsf{in} \\
\quad \mathsf{let}\ self\_object = mk\text{-}ClassDerivedObject(self\_class, inst\_vars) \\
\quad\quad \mathsf{in} \\
\quad \mathsf{let}\ \sigma' = \mu(\sigma, OopMap \mapsto OopMap(\sigma) \cup \{self\_oop \mapsto self\_object\}) \\
\quad\quad \mathsf{in} \\
\quad \mathsf{let}\ \sigma'' = add\_virtual\_objects(inst\_vars, \sigma') \\
\quad\quad \mathsf{in} \\
\quad \mathsf{let}\ \sigma''' = assign(self, self\_oop, \sigma'') \\
\quad\quad \mathsf{in} \\
\quad \mathsf{let}\ predefined\sigma s = \{assign(self, Oop, \sigma) \\
\qquad\qquad\qquad\qquad \mid Oop \in \mathsf{dom}\, OopMap(\sigma) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ Class(OopMap(\sigma)(Oop)) = self\_class\} \\
\quad\quad \mathsf{in} \\
\quad \{\sigma'''\} \cup predefined\sigma s
\end{array}
$$

[8] By the initial construction of the object space, σ in this case, the only objects in the object space at this point will be predefined objects.

The tagging of the instance variables is done by *tag_inst_vars*. Because the tagging is done at a syntactic level, for example by appending a suitable number of 'primes' ($'$) to the end of the variable's identifier, the definition of this function has been omitted.

$tag\_inst\_vars : ClassName \times ObjectSpace \rightarrow InstVar\text{-}\mathsf{set}$

For the general case, an initial object space will contain the following items:

Objects	:	$self\_oop \mapsto \mathsf{self}$
		$virtual\_oops \rightarrow$ instance variables
References	:	$\mathsf{self} \mapsto self\_oop$
		instance variables $\rightarrow virtual\_oops$
Object map	:	$self\_oop$ (virtual) $\mapsto self\_object$ (class-derived)
		$virtual\_oops \rightarrow virtual\_objects$
		$predefined\_oops \rightarrow predefined\_objects$ (class-derived)

For a special case involving one of the predefined objects, the object space will have the following form:

Objects	:	$predefined\_oop \mapsto \mathsf{self}$
		$virtual\_oops \rightarrow$ instance variables
References	:	$\mathsf{self} \mapsto predefined\_oop$
		instance variables $\rightarrow virtual\_oops$
Object map	:	$virtual\_oops \rightarrow virtual\_objects$
		$predefined\_oops \rightarrow predefined\_objects$ (class-derived)

4.4.2 Analysing a method

The function *DfMethod* concludes the process of initialising the dataflow environment, and the object space in particular, by setting up the environment for a particular method, ready for the analysis of the method body by *DfExpression*. It takes as its primary argument a user-defined method, consisting of a sequence of formal parameters, a set of temporary variables and a method body. As secondary arguments, it takes a dataflow environment and the selector of the method being analysed. In return it provides a set of dataflow environments, the results from the analysis of each potential control path followed through by *DfExpression*.

On entry to a new method, the parameters and temporaries used in that method are suitably tagged to distinguish them, within the object space, from their use with future invocations of that method. A similar form of tagging is performed for the instance variables of a new object, and here the task is carried out by *tag_method_variables*, which deals with the parameters and temporary variables of a method, both within their respective fields of the method definition and their uses in the method body. As with *tag_inst_vars*, the definition of this function has been omitted, although its signature can be found immediately after the definition of *DfMethod*.

Like the instance variables of the self object, the parameters of the method must be initialised, and once again they are initialised to be virtual objects, using the *add_virtual_objects* function defined on page 98. In a similar fashion, the temporary variables must also be initialised, this time to the 'undefined' object, nil, using *add_nil_objects* (page 100).

A record of the current method provides the first entry for the method invocation record. The actual form of the record, together with the manner of its construction will be described in §4.5, but for now it is sufficient to say that it contains information about the classes of the receiver and arguments to the method, as well as the method selector. This information is extracted by the function *extract_recurs*, which requires the method selector, a sequence of object identities made up from the self object and argument objects, and the current dataflow environment. The leading (empty) argument will be utilised during proper message sends to help record the invoking expression. The resulting *Recurs* structure can then be inserted into the method invocation record. The last part of this function hands control over to *DfExpression* to commence the code analysis on the body of the method.

$$DfMethod : UserMethod \rightarrow (DfEnv \times Selector) \rightarrow DfSet$$

$$DfMethod\,[\![user\_method]\!]\,(\delta, sel) \quad \triangleq$$

$$\begin{array}{l}
\quad \mathsf{let}\ user_method' = tag_method_variables(user_method, \sigma(\delta)) \\
\qquad \mathsf{in} \\
\quad \mathsf{let}\ mk\text{-}UserMethod(params, temps, body) = user_method' \\
\qquad \mathsf{in} \\
\quad \mathsf{let}\ \delta' = \mu(\delta, \sigma \mapsto add_virtual_objects(\mathsf{elems}\, params, \sigma(\delta))) \\
\qquad \mathsf{in} \\
\quad \mathsf{let}\ \delta'' = \mu(\delta', \sigma \mapsto add_nil_objects(temps, \sigma(\delta'))) \\
\qquad \mathsf{in} \\
\quad \mathsf{let}\ self_oop = Refs(\sigma(\delta''))(self) \\
\qquad \mathsf{in} \\
\quad \mathsf{let}\ param_oops = get_param_oops(params, \sigma(\delta'')) \\
\qquad \mathsf{in} \\
\quad \mathsf{let}\ mir_entry = extract_recurs(_, sel, [self_oop] \frown param_oops, \delta'') \\
\qquad \mathsf{in} \\
\quad \mathsf{let}\ \delta''' = \mu(\delta, mir \mapsto [mir_entry]) \\
\qquad \mathsf{in} \\
\quad DfExpression\,[\![body]\!]\,\delta'''
\end{array}$$

$$tag\_method\_variables : UserMethod \times ObjectSpace \rightarrow UserMethod$$

The following function, *get_param_oops*, is used to extract the argument object identities from the object space, for use by *extract_recurs*. It takes a sequence of parameters and constructs a corresponding sequence of object identities by extracting them through the *Refs* map of the given object space.

$$get\_param\_oops : ParamId^* \times ObjectSpace \rightarrow Oop^*$$

$$get\_param\_oops(params, \sigma) \quad \triangleq$$
$$\quad \text{if } params = [\,]$$
$$\quad \text{then } [\,]$$
$$\quad \text{else } [Refs(\sigma)(\text{hd}\, params)] \frown get\_param\_oops(\text{tl}\, params, \sigma)$$

With the last part of the dataflow environment initialised, it is now possible for the code analysis proper to commence. For general cases, an initial object space will now contain the following items:

Objects	:	$self\_oop \mapsto$ self
		$virtual\_oops \rightarrow$ instance variables
		$virtual\_oops \rightarrow$ parameters
		nil $\rightarrow$ temporary variables
References	:	self $\mapsto self\_oop$
		instance variables $\rightarrow virtual\_oops$
		parameters $\rightarrow virtual\_oops$
		temporary variables $\rightarrow$ nil
Object map	:	$self\_oop$ (virtual) $\mapsto self\_object$ (class-derived)
		$virtual\_oops \rightarrow virtual\_objects$
		$predefined\_oops \rightarrow predefined\_objects$ (class-derived)

4.4.3 Analysing an expression

The analysis proper starts with the *DfExpression* function. This takes an expression to be analysed and a dataflow environment. In return it provides a set of dataflow environments, each one the result of following a potential control path through the expression.

$$DfExpression : Expression \rightarrow DfEnv \rightarrow DfSet$$

When applied to a list of expressions, *DfExpression* processes the list, analysing each expression in turn. *DfExpression* for this case is defined recursively, with the terminating case being an empty expression list, which returns the given dataflow environment, transformed into a singleton set to comply with the function signature. The general case initially invokes *DfExpression* on the first expression, resulting in a set of dataflow environments, and then invokes it on the rest of the expressions in the list. In order to propagate the results from the first expression, the rest of the expression sequence is analysed once for each of the 'cases' returned by that initial expression. The results from each of these subsequent cases will be a further set of dataflow environments. In other words, a single set of environments for the first expression, a set of sets for the second, a set of sets of sets for the third and so on. It is therefore necessary to take the distributed union of these sets of sets as the recursion unfolds, in order to yield a single set of dataflow environments.

$$\begin{array}{l} \mathit{DfExpression}\,[\![\mathit{expr}]\!]\,\delta \quad \triangleq \qquad\qquad\qquad\qquad \text{for } \mathit{expr} \in \mathit{ExpressionList} \\ \quad \text{if } \mathit{expr} = [\,] \\ \quad \text{then } \{\delta\} \\ \quad \text{else let } \delta\mathit{set} = \mathit{DfExpression}\,[\![\text{hd}\,\mathit{expr}]\!]\,\delta \\ \qquad\quad \text{in} \\ \qquad \bigcup\,\{\mathit{DfExpression}\,[\![\text{tl}\,\mathit{expr}]\!]\,\delta' \mid \delta' \in \delta\mathit{set}\} \end{array}$$

The analysis of an assignment expression can be broken down into three stages: the analysis of the left-hand variable, the analysis of the right-hand expression, and finally the 'assignment' of the right-hand result to the variable. Analysis of the left-hand variable is only really necessary for instance variables, when it is vital to use the correctly-tagged version, as recorded by the self object. This analysis is carried out by *extract_ass_var*, which returns the correct version of the left-hand variable. The analysis of the right-hand expression is performed by calling *DfExpression* on it, yielding a set of dataflow environments, with each environment containing a potential result in its *last* field. The assignment is performed by the *assign* function (page 96), taking the variable to be assigned to and the object identity to be assigned, located in the *last* field. By applying *assign* for each environment returned by the right-hand expression, each case returned by that expression ultimately yields an updated environment with the assignment recorded in its object space, and with the result of the whole expression left unaltered in its *last* field.

$$\begin{array}{l} \mathit{DfExpression}\,[\![\mathit{mk\text{-}Assignment}(\mathit{lhs}, \mathit{rhs})]\!]\,\delta \quad \triangleq \\ \quad \text{let } \mathit{ass\_var} = \mathit{extract\_ass\_var}(\mathit{lhs}, \sigma(\delta)) \\ \qquad \text{in} \\ \quad \text{let } \delta\mathit{set} = \mathit{DfExpression}\,[\![\mathit{rhs}]\!]\,\delta \\ \qquad \text{in} \\ \quad \{\mu(\delta', \sigma \mapsto \mathit{assign}(\mathit{lhs}, \mathit{last}(\delta'), \sigma(\delta'))) \mid \delta' \in \delta\mathit{set}\} \end{array}$$

Extract_ass_var is defined separately for the two cases which can occur. If the variable is a temporary one, the function simply returns the variable unchanged. This works because on the initial entry to a method, all occurrences of the parameters and temporaries within that method are tagged to make them unique within the object space (page 104), and so the temporary variable will already be the correctly tagged version.

$$\mathit{extract\_ass\_var} : \mathit{AssVar} \times \mathit{ObjectSpace} \rightarrow \mathit{AssVar}$$

$$\begin{array}{l} \mathit{extract\_ass\_var}(\mathit{temp\_var}, \sigma) \quad \triangleq \\ \quad \mathit{temp\_var} \end{array}$$

where

$$\mathit{temp\_var} \in \mathit{TempVar}$$

For the case when the variable is an instance variable, rather than tag its use within the method code, the correctly tagged version is extracted from the self object recorded in the object space. The function works by taking the instance variable as it occurs in the method code, *inst_var*, and finding the equivalent instance variable from the self object. The correspondence test makes use of the function *untag_inst_var*, which removes the extra tagging from the self object's instance variable, leaving it in its original form.

$$\begin{array}{l} \mathit{extract\_ass\_var}(\mathit{inst\_var}, \mathit{mk\text{-}ObjectSpace}(\mathit{objects}, \mathit{refs}, \mathit{oop\_map})) \quad \triangleq \\ \qquad \iota\ \mathit{inst\_var}' \in \mathit{Inst\_vars}(\mathit{oop\_map}(\mathit{refs}(\mathit{self}))) \cdot \\ \hfill \mathit{untag\_inst\_var}(\mathit{inst\_var}') = \mathit{inst\_var} \end{array}$$

where

$\mathit{inst\_var} \in \mathit{InstVar}$

The definition of *untag_inst_var* relies on the definition of *tag_inst_vars* in two respects. Firstly the actual implementation of the former depends on the implementation of the latter. As such, only the signature of *untag_inst_var* can be presented here:

$$\mathit{untag\_inst\_var} : \mathit{InstVar} \to \mathit{InstVar}$$

Secondly *untag_inst_var* assumes that the tagging applied by *tag_inst_vars* is reversible. For the example implementation described on page 104, this will certainly be the case: 'primes' ($'$), textually equivalent to the single quote, are not valid for use within variable or parameter identifier names, however within the confines of the analysis mechanism, their presence is quite legal. As such the presence of 'primes' at the end of a variable name can only be the consequence of tagging, and the process of their removal would therefore be straightforward.

For an object name expression (a variable, parameter or self), in general it is only necessary to follow the link from the identifier to its associated object identity through the *Refs* map of the object space of the dataflow environment. This yields an object identity which can be slotted into the *last* field of the dataflow environment to give the correct result. However for instance variables it is necessary to use the correctly tagged version, as recorded by the self object. This is done, as with the assignment expression above, by finding the instance variable of the self object, which, when stripped of its tag, corresponds to the given instance variable expression. For both cases, the resulting updated dataflow environment must be transformed into a singleton set in order to comply with the function signature.

$$\begin{array}{l} \mathit{DfExpression}[\![\mathit{expr}]\!]\,\delta \quad \triangleq \hfill \text{for } \mathit{expr} \in \mathit{ObjectName} \\ \qquad \mathsf{if}\ \mathit{expr} \in \mathit{InstVar} \\ \qquad \mathsf{then\ let}\ \mathit{inst\_var} \in \mathit{Inst\_vars}(\mathit{oop\_map}(\mathit{Refs}(\sigma(\delta))(\mathit{self}))) \cdot \\ \hfill \mathit{untag\_inst\_var}(\mathit{inst\_var}) = \mathit{expr} \\ \qquad\qquad \mathsf{in} \\ \qquad\quad \{\mu(\delta, \mathit{last} \mapsto \mathit{Refs}(\sigma(\delta))(\mathit{inst\_var}))\} \\ \qquad \mathsf{else}\ \{\mu(\delta, \mathit{last} \mapsto \mathit{Refs}(\sigma(\delta))(\mathit{expr}))\} \end{array}$$

The creation of a new object is performed by the *NewObject* expression. The necessary updates to the dataflow environment are carried out in two separate steps. A call to *prepare_new_object*, providing the class of the new object together with the current object space, returns a new object identity and an updated object space with the new object already in place. It is then only necessary to place the updated object space back into the dataflow environment, and to slot the new object identity into the *last* field. Once again, the function signature requires that the result be transformed into a singleton set.

$$
\begin{aligned}
&\mathit{DfExpression}\,[\![\mathit{mk\text{-}NewObject}(\mathit{class})]\!]\,\delta \quad \triangleq\\
&\quad \mathsf{let}\ (\mathit{new\_oop}, \sigma') = \mathit{prepare\_new\_object}(\mathit{class}, \sigma(\delta))\\
&\qquad \mathsf{in}\\
&\quad \mathsf{let}\ \delta' = \mu(\delta, \sigma \mapsto \sigma')\\
&\qquad \mathsf{in}\\
&\quad \{\mu(\delta', \mathit{last} \mapsto \mathit{new\_oop})\}
\end{aligned}
$$

The main work in creating a new object is carried out by *prepare_new_object*. This function takes the class of the new object together with an object space, and then creates and inserts the new object into that object space. It starts by creating a new object identity, using *new_concrete_object_id*, which provides a unique ‘concrete’ identity. The actual object is created by constructing a new class-derived object using the class name provided, together with a set of suitably tagged instance variables, a task performed by *tag_inst_vars* (page 104) using the indicated class. The process of updating the object space starts with the addition to the *OopMap* field of a maplet linking the new object identity to the new class-derived object. The updating is completed by initialising the instance variables to nil using the *add_nil_objects* function (page 100).

$$
\begin{aligned}
&\mathit{prepare\_new\_object} : \mathit{ClassName} \times \mathit{ObjectSpace} \to \mathit{Oop} \times \mathit{ObjectSpace}\\
&\mathit{prepare\_new\_object}(\mathit{class}, \sigma) \quad \triangleq\\
&\quad \mathsf{let}\ \mathit{new\_oop} = \mathit{new\_concrete\_object\_id}(\sigma)\\
&\qquad \mathsf{in}\\
&\quad \mathsf{let}\ \mathit{inst\_vars} = \mathit{tag\_inst\_vars}(\mathit{class}, \sigma)\\
&\qquad \mathsf{in}\\
&\quad \mathsf{let}\ \mathit{new\_object} = \mathit{mk\text{-}ClassDerivedObject}(\mathit{class}, \mathit{inst\_vars})\\
&\qquad \mathsf{in}\\
&\quad \mathsf{let}\ \sigma' = \mu(\sigma, \mathit{OopMap} \mapsto \mathit{OopMap}(\sigma) \cup \{\mathit{new\_oop} \mapsto \mathit{new\_object}\})\\
&\qquad \mathsf{in}\\
&\quad \mathsf{let}\ \sigma'' = \mathit{add\_nil\_objects}(\mathit{inst\_vars}, \sigma')\\
&\qquad \mathsf{in}\\
&\quad (\mathit{new\_oop}, \sigma'')
\end{aligned}
$$

Comparing the above function with *prepare_self* (page 103) will show how the two functions share a similar basic structure. Both functions are interested in generating new object identities and creating class-derived objects. However a closer inspection will show a fundamental difference between them. Whereas *prepare_self* requires a *virtual* object identity, and initialises its instance variables to be *virtual* objects, *prepare_new_object* deals only with concrete identities (or predefined identities in the case of the instance variables). This difference reflects the separate origins of the two objects. The self object is only an estimate, to be refined as its (potential) structure is probed through the code being analysed, when virtual objects are resolved into a sets of potential objects, each one yielding a separate case to be followed. In contrast, the structure of an object 'created' by *prepare_new_object* is known from the start of its existence.

The analysis of a literal expression follows the same pattern as for a new-object expression. It passes the main work over to *prepare_literal* and then slots the resulting object identity and object space back into the dataflow environment.

$$
\begin{array}{l}
\mathit{DfExpression}\,[\![\mathit{mk\text{-}Literal}(\mathit{value})]\!]\,\delta \quad \triangleq \\
\quad \text{let } (\mathit{new\_oop}, \sigma') = \mathit{prepare\_literal}(\mathit{value}, \sigma(\delta)) \\
\qquad \text{in} \\
\quad \text{let } \delta' = \mu(\delta, \sigma \mapsto \sigma') \\
\qquad \text{in} \\
\quad \{\mu(\delta', \mathit{last} \mapsto \mathit{new\_oop})\}
\end{array}
$$

The definition of *prepare_literal* depends on how the syntax of the language being analysed handles symbols, and this in turn dictates the kind of literal expressions that can be formed. At the moment the only literal expressions recognised by the analysis mechanism are nil, the 'undefined' object, true, the 'true' object, and false, the 'false' object. As these objects already exist in the object space, it is only necessary to return the identity of the relevant object, the predefined object identity nil, true or false, and an unchanged object space. Clearly other kinds of literal expressions would require their own specialised treatment.

$$
\begin{array}{l}
\mathit{prepare\_literal} : \ldots \times \mathit{ObjectSpace} \rightarrow \mathit{Oop} \times \mathit{ObjectSpace} \\
\mathit{prepare\_literal}(\mathit{value}, \sigma) \quad \triangleq \\
\quad \text{cases } \mathit{value} \text{ of} \\
\qquad [\![\mathit{nil}]\!] \rightarrow (\mathit{nil}, \sigma) \\
\qquad [\![\mathit{true}]\!] \rightarrow (\mathit{true}, \sigma) \\
\qquad [\![\mathit{false}]\!] \rightarrow (\mathit{false}, \sigma) \\
\quad \text{end}
\end{array}
$$

4.4.4 Analysing a message send

The analysis of the message send expression is the most complicated of all the expressions and involves several stages. The main stages, laid out in the *DfExpression* function for message sends, are: analysis of the message receiver and message arguments; analysis of the message send itself and the invoked method; and finally, making any necessary 'post-message' adjustments to the dataflow environment. These three stages can be seen in the three function calls which occur in the *DfExpression* function defined below, respectively *DfArgs*, *DfMessage*, and *merge_envs*.[9]

$$
\begin{array}{l}
\mathit{DfExpression}\,[\![\mathit{mk\text{-}Message}(\mathit{rcvr, sel, args, marker})]\!]\,\delta \quad \triangleq \\
\quad \text{let } \mathit{args\_and\_}\delta\mathit{\_set} = \mathit{DfArgs}([\mathit{rcvr}] \frown \mathit{args}, \delta) \\
\qquad \text{in} \\
\quad \{\mathit{merge\_envs}(\mathit{arg}\delta, \mathit{msg}\delta) \\
\qquad\quad \mid (\mathit{arg\_oops}, \mathit{arg}\delta) \in \mathit{args\_and\_}\delta\mathit{\_set} \\
\qquad\qquad\qquad \wedge\, \mathit{msg}\delta \in \mathit{DfMessage}(\mathsf{hd}\,\mathit{arg\_oops}, \mathit{sel}, \mathsf{tl}\,\mathit{arg\_oops}, \mathit{marker}, \mathit{arg}\delta)\}
\end{array}
$$

The first stage, that of analysing the receiver and argument expressions, is performed by calling *DfArgs*. *DfArgs* takes a sequence of expressions and a dataflow environment, and, in principle, returns a corresponding sequence of object identities (the result of analysing each expression) and an updated environment (the cumulative result of the analysis process). As the sequence of arguments is analysed using *DfExpression*, each argument yields not just a single dataflow environment, but a set of environments. The result of this is that, as each set of environments from one argument is propagated to the next argument, there is not one sequence of processed arguments, but a set of such sequences. The eventual outcome of *DfArgs* is in fact a set of 'cases', where one case is made up of a single sequence of processed arguments (now object identities) and a dataflow environment corresponding to the analysis that resulted in that sequence. The result of calling *DfArgs* — recorded as *args_and_δ_set* — is a set of argument sequence/environment cases:

$$((\text{object identities})^*, \text{dataflow environment})\text{-set}$$

The second stage, involving the message send itself, can be seen as constructing a set of cases to be dealt with by the third, post message-adjusting stage. For each case created by *DfArgs*, the sequence of argument objects, *arg_oops*, and their associated environment, *argδ*, are used as the basis of a call to *DfMessage* which handles the message send itself. *DfMessage* takes as its arguments the receiver, the message arguments, the message selector, the marker attached to the message expression (see page 89) and the dataflow environment. The result of *DfMessage*, a set of dataflow environments, forms the basis of the set of cases to be dealt with by *merge_envs*. For each case (argument sequence, environment) created by *DfArg* there is now a set of cases (environments) created by *DfMessage*.

[9] Not to be confused with the *merge_envs* function defined in Chapter 3.

The final, post-message restoration stage simply sorts out the dataflow environments resulting from the message send into final post-message dataflow environments. One by one, each pre-message environment, $arg\delta$, is paired with one of its corresponding post-message environments, $msg\delta$, and processed by *merge_envs*, which returns a single 'merged' dataflow environment. *Merge_envs* takes the post-message environment and replaces those elements which link it to the now-returned method with details re-linking it to the now-current method. In particular self must be reset to the correct object. These important details are recorded in the pre-message environment, $arg\delta$.

Analysing the arguments

DfArgs is defined next. It takes the sequence made up of the receiving expression and argument expressions of the message send, together with the dataflow environment, and analyses each expression in order, propagating the effect on the environment from expression to expression. It is defined recursively, taking the empty sequence of expressions as its terminating case, when it returns a singleton set made up of an empty sequence (of object identities) and the dataflow environment.

The general case is slightly more complicated. First the head of the argument expression sequence is analysed by *DfExpression*, resulting in a set of dataflow environments. This set is stored as $hd\delta set$, ready to be used for the analysis of the rest of the argument expressions. The recursive call of *DfArgs* can be found in the set constructor defining the result of the function. For each 'head' case ($hd\delta$) created by the analysis of the first argument, *DfArgs* is invoked on the remaining arguments. The overall result, allowing for the unwinding of the recursion on termination, is a set of cases where each case is made up of a sequence of object identities (the processed arguments) and a 'final' dataflow environment (the result of analysing the last argument expression).

Each 'head' case must now be combined with each 'tail' case derived from it. This is done by selecting a single case ($tl\_arg\_oops, tl\delta$) from the set of 'tail' cases derived from a given 'head' case ($hd\delta$), and combining the processed arguments. The 'head' argument is extracted from the *last* field of the 'head' case and concatenated with the processed 'tail' arguments, *tl_arg_oops*. Finally, the returned dataflow environment is the 'tail' environment, the environment resulting from the analysis of the last argument expression.

$$DfArgs : Expression^* \times DfEnv \to (Oop^* \times DfEnv)\text{-set}$$

$$\begin{aligned}
&DfArgs(args, \delta) \quad \triangleq \\
&\quad \text{if } args = [\,] \\
&\quad \text{then } \{([\,], \delta)\} \\
&\quad \text{else let } hd\delta set = DfExpression\, [\![\text{hd}\, args]\!]\, \delta \\
&\qquad\quad \text{in} \\
&\qquad \{([last(hd\delta)] \frown tl_arg_oops, tl\delta) \mid hd\delta \in hd\delta set \\
&\qquad\qquad\qquad \wedge (tl_arg_oops, tl\delta) \in DfArgs(\text{tl}\, args, hd\delta)\}
\end{aligned}$$

Analysing the message

DfMessage takes its main arguments in the same order as a message expression — receiver object identity, message selector, argument object identities — as well as receiving the 'positional' marker attached to the original message expression, and the necessary dataflow environment. The main work carried out by *DfMessage* is spent in deriving the class of the receiving object, before passing this information on to the function which actually processes the message send, *process_message*.

Throughout the analysis process described so far, it has been assumed that the analysis of a given expression will yield several potential results, or cases, each reflecting a different control path followed. In fact, this assumption has its basis in *DfMessage*, where the process of extracting the class of the receiver, and hence the choice of invoked method, may result in several potential classes if the receiving object is an unresolved *virtual object*. It is these individual cases that give rise to the sets of dataflow environments which occur throughout the analysis process. The resolution process, and the way in which the cases are derived will be described over the course of the following group of functions associated with *DfMessage*.

The act of extracting the receiver class is performed by invoking *find_rcvr_class*, which takes the object identity of the receiver, together with the message selector and the dataflow environment. In return it provides a set of potential classes, each paired with its own dataflow environment reflecting the decisions made in coming up with that class. This set of potential classes is stored as $class\_and\_\delta\_set$, a set of class/environment pairs. If the resulting set is not empty, then there is at least one case to follow up, otherwise there must be an incompatibility between the receiver and the message. For this latter situation, an error must be flagged through the *does_not_understand* function. This error function is assumed to be able to indicate the nature of the problem, as well as return control to some suitable point, for example an alternative case, method or class, and so allow the analysis of the rest of the system to continue. Because of the potential complexity such a function would introduce into the system being described, it has not been defined here.

In the more normal event that there are potential classes to be followed up, control is passed over to *process_message* for each of the classes selected. As each separate case will result in a set of dataflow environments, the overall result must be taken as the distributed union over the individual results.

$$DfMessage : Oop \times Selector \times Oop^{*} \times Id \times DfEnv \rightarrow DfSet$$

$$
\begin{array}{l}
DfMessage(rcvr\_oop, sel, arg\_oops, marker, \delta) \quad \triangleq \\
\quad \text{let } class\_and\_\delta\_set = find\_rcvr\_class(rcvr\_oop, sel, \delta) \\
\quad\quad \text{in} \\
\quad \text{if } class\_and\_\delta\_set \neq \{\,\} \\
\quad \text{then } \bigcup \{process\_message(rcvr\_class, rcvr\_oop, sel, arg\_oops, marker, \delta') \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid (rcvr\_class, \delta') \in class\_and\_\delta\_set\} \\
\quad \text{else } does\_not\_understand(rcvr\_oop, sel, \delta)
\end{array}
$$

Resolving the receiver

The creation of the cases described earlier starts with *find_rcvr_class*. This function acts as a filter for the real functions which do the work. It calls *get_class_and_δ_set* which returns a set of class/environment pairs. It then filters out those which do not implement the required method, and returns the resulting set to *DfMessage*.

$$\begin{array}{l}
\mathit{find_rcvr_class} : \mathit{Oop} \times \mathit{Selector} \times \mathit{DfEnv} \to (\mathit{ClassName} \times \mathit{DfEnv})\text{-set} \\
\mathit{find_rcvr_class}(\mathit{oop}, \mathit{sel}, \delta) \quad \triangleq \\
\qquad \{(\mathit{class}, \delta') \mid (\mathit{class}, \delta') \in \mathit{get_class_and_}\delta\mathit{_set}(\mathit{oop}, \delta) \\
\qquad\qquad\qquad \wedge\ \mathit{sel} \in \mathrm{dom}\, \mathit{Methods}(\mathit{program}(\mathit{class}))\}
\end{array}$$

Get_class_and_δ_set starts off the creation of cases proper by following up the receiving object's identity to the receiving object itself, through the *OopMap* of the environment's object space. At this point it is now possible to distinguish between virtual objects and class-derived objects. For the latter kind, it is only necessary to return the class of the receiver straight from the receiving object. However for the former, *virtual* kind it is necessary to separate out the potential 'real' objects that the virtual object might represent. This is a multi-stage process which starts by taking each of the class-derived objects recorded by the virtual object and creating cases for each one. This case creation is a form of resolution, and is carried out by *resolve_v_obj* for each of the potential 'real' objects.

$$\begin{array}{l}
\mathit{get_class_and_}\delta\mathit{_set} : \mathit{Oop} \times \mathit{DfEnv} \to (\mathit{ClassName} \times \mathit{DfEnv})\text{-set} \\
\mathit{get_class_and_}\delta\mathit{_set}(\mathit{oop}, \delta) \quad \triangleq \\
\qquad \mathrm{let}\ \mathit{object} = \mathit{OopMap}(\sigma(\delta))(\mathit{oop}) \\
\qquad\quad \mathrm{in} \\
\qquad \mathrm{cases}\ \mathit{object}\ \mathrm{of} \\
\qquad\qquad\quad \mathit{mk\text{-}VirtualObject}(\mathit{cdo_set}) \to \bigcup\{\mathit{resolve_v_obj}(\mathit{oop}, \mathit{cdo}, \delta) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{cdo} \in \mathit{cdo_set}\} \\
\qquad \mathit{mk\text{-}ClassDerivedObject}(\mathit{class}, \mathit{i_vs}) \to \{(\mathit{class}, \delta)\} \\
\qquad \mathrm{end}
\end{array}$$

Resolve_v_obj takes the identity of the object in question (a virtual identity) and attempts to replace its virtual body with a chosen class-derived one. In return for a virtual object identity, a suitable class-derived object (chosen for the creation of this particular 'case'), and lastly, the ubiquitous dataflow environment, it alters the environment to reflect the new 'case' or 'cases'. Because almost all the work has to be performed on the object space, the main effort of the function is delegated to *resolve_v_obj_cases*, which takes the identity, class-derived object and object space, and returns a set of updated object spaces ready to be used in the creation of the new cases.

The form of the result required by *get_class_and_δ_set* is a set of class name/environment pairs, and this form is provided simply by returning the class of the chosen 'real' object, and by updating the dataflow environment with the object space returned by *resolve_v_obj_cases*.

$$resolve\_v\_obj : Oop \times ClassDerivedObject \times DfEnv \rightarrow (ClassName \times DfEnv)\text{-set}$$

$$resolve\_v\_obj(voop, cdo, \delta) \quad \triangleq$$
$$\quad \{(Class(cdo), \mu(\delta, \sigma \mapsto \sigma')) \mid \sigma' \in resolve\_v\_obj\_cases(voop, cdo, \sigma(\delta))\}$$

Given that each class-derived object conveniently stored in the virtual object forms the basis of a particular case, it may seem strange that the whole resolving process for a single such object returns a *set* of cases. The explanation for this apparent mismatch can be found in the definition of the following function. *Resolve_v_obj_cases* takes the object identity to be resolved, the class-derived object which forms the basis of the resolution, and the object space to be updated, and returns, as indicated above, a set of object spaces. The problem is that although the primary case can be created by linking the given identity to the given class-derived object, cases may also need to be created linking the identity with class-derived objects already in the object space. Taking these two situations separately, those cases which involve the identity being linked to objects already present are grouped together as a set of object spaces recorded as *old*σ*s*, while the case based on the new object is recorded as σ''. The final result is simply the union of these cases.

$$resolve\_v\_obj\_cases : Oop \times ClassDerivedObject \times ObjectSpace \rightarrow ObjectSpace\text{-set}$$

$$\begin{array}{l}
resolve_v_obj_cases(voop, cdo, \sigma) \quad \triangleq \\
\quad \text{let } old\sigma s = \{equate_oops(oop, voop, \sigma') \\
\qquad\qquad \mid oop \in \text{dom}\, OopMap(\sigma) \\
\qquad\qquad \wedge\, oop \in PredefinedOop \cup VirtualOop \\
\qquad\qquad \wedge\, OopMap(\sigma)(oop) \in ClassDerivedObject \\
\qquad\qquad \wedge\, Class(OopMap(\sigma)(oop)) = Class(cdo)\} \\
\qquad \text{in} \\
\quad \text{let } cdo' = \mu(cdo, Inst_vars \mapsto tag_inst_vars(Inst_vars(cdo), \sigma)) \\
\qquad \text{in} \\
\quad \text{let } \sigma' = \mu(\sigma, OopMap \mapsto OopMap(\sigma) \cup \{voop \mapsto cdo'\}) \\
\qquad \text{in} \\
\quad \text{let } \sigma'' = add_virtual_objects(Inst_vars(cdo'), \sigma') \\
\qquad \text{in} \\
\quad \{\sigma''\} \cup old\sigma s
\end{array}$$

Taking the primary case first, the new class-derived object is linked to the virtual identity in two stages. After the 'new' class-derived object has had its instance variables tagged, the first, principal step involves the *OopMap* of the object space being updated with the maplet linking the identity to its newly-resolved object. The second, more subtle step is required to keep the system up to date. Now that the virtual object has been resolved, it is necessary to alter the space in such a way that it appears as if the object has always been resolved. For this it is necessary to initialise the instance variables of the 'new' object to reference virtual objects themselves.

The effect of initialising the instance variables at this point is identical in effect to initialising them at some arbitrary point in the past, and the whole process works because of the way the resolution process works in conjunction with the principle of object encapsulation. Put simply, the principle states that the only way in which an instance variable of an object can be accessed is through a message send, but it is through receiving messages that virtual objects are resolved. As a result, although the internal state of the newly-resolved object has only just been 'created', the lack of its existence up to this point will not have affected the rest of the system. However, now the creation point must be back-dated to when the original virtual object came into existence, and this is done simply by initialising the instance variables of the newly-resolved object to refer to virtual objects themselves.

The more complicated situation, namely the handling of those objects which already exist in the object space, involves the selection of such objects, and then passing them on to *equate_oops* to perform the necessary manipulation to the object space. The selection of a candidate object involves making sure it matches the following criteria:

1. The candidate object should *either* have a predefined identity, *or* a virtual identity.

2. If the candidate object has a virtual identity, it must already have been resolved into a class-derived object.

3. The class of the candidate object should be the same as the class of the object being resolved.

With a suitable object chosen, and selected by its identity, the two object identities must be passed to *equate_oops*, together with the object space, ready to be equated.

By dealing with each class separately, *resolve_v_obj* creates an exhaustive set of cases. For each class represented in the original virtual object, cases can be created where the identity is linked to a class-derived object, as well as to predefined objects and other virtual objects which have already been resolved — a form of retrospective equality carried out by *equate_oops*.

Equate_oops does more than simply equate two object identities. In fact it reflects just why virtual object identities are virtual. Because the creation of a virtual object is not observed, but merely assumed, the precise nature of the object is not known.[10] This lack of knowledge relates not only to the class of the object, which is resolved through message sends during the analysis process, but also to its relationship with other objects, in particular whether two apparently separate objects are not, in fact, the same object. Therefore the criteria listed above ensure that any possible aliasing between object references at the start of the analysis process can be handled properly.

An object only has the potential to affect other objects through message sends, by receiving a message itself and invoking a method. It follows that only resolved objects have this potential,

[10]This is in contrast to the predefined object which has an additional assumption that despite the non-observance of its creation, its precise nature *is* known.

because only fully resolved objects can invoke their methods. It also follows that when a virtual object is resolved, by receiving its first message, it is possible to consider its equality with other resolved objects without any problems of introducing inconsistencies into the object space through the prior activities of either the object being resolved or the object being equated. In creating a case by assuming the equality of the two objects and their identities it is clear that the two objects must be unified. Unifying the two object bodies is simple: remove the virtual object. As for dealing with the identities, clearly one of them must be replaced by the other, and it is this task which is performed by *equate_oops*.

Equate_oops takes the main object identity as its first argument, *main_oop*, with the virtual object identity to be overwritten as its second, *voop*. The three fields of the object space are treated separately as follows. The *Objects* field, of the form:

$$\text{object identity} \;\mapsto\; \text{(object references)-set}$$

needs to have the set of object references associated with the virtual identity added to the set associated with the main object identity. After this has been done, it is safe to remove the maplet involving the virtual identity.

The *Refs* field, of the form:

$$\text{object reference} \;\mapsto\; \text{object identity}$$

must have every object name referencing the virtual identity altered, to associate it with the main object identity.

Finally, for the *OopMap* field, it is necessary to remove the now-redundant entry involving the virtual object identity. By combining these three updated fields, the final object space has had all traces of the old virtual object identity removed, while in its place references to the main object identity have been inserted.

$$equate\_oops : Oop \times Oop \times ObjectSpace \to ObjectSpace$$

$$equate\_oops(main\_oop, voop, mk\text{-}ObjectSpace(objects, refs, oop\_map)) \;\triangleq$$
$$\quad \text{let } objects' = objects \dagger \{main\_oop \mapsto (objects(main\_oop) \cup objects(voop))\}$$
$$\quad\quad \text{in}$$
$$\quad \text{let } objects'' = \{voop\} \ntriangleleft objects$$
$$\quad\quad \text{in}$$
$$\quad \text{let } refs' = refs \;\dagger\; \{voop\_ref \mapsto main\_oop \mid \{voop\_ref \mapsto voop\} \in refs\}$$
$$\quad\quad \text{in}$$
$$\quad \text{let } oop\_map' = \{voop\} \ntriangleleft oop\_map$$
$$\quad\quad \text{in}$$
$$\quad mk\text{-}ObjectSpace(objects'', refs', oop\_map')$$

To help clarify the way in which virtual objects are resolved into different cases, here are a couple of examples showing how entries in the *OopMap* of the object space are altered. The examples assume the existence of a virtual object identity, *virtual_id* and a predefined object identity, *predefined_id*, as well as a variable (i.e. object reference), ref_1. An *OopMap* with an entry for a virtual identity/virtual object pair,

$$virtual\_id \mapsto mk\text{-}VirtualObject(\{cdo_1, cdo_2, \ldots\})$$

will, when resolved, create multiple cases, including:

$$virtual\_id \mapsto cdo_1$$

and

$$virtual\_id \mapsto cdo_2$$

By comparison, an *OopMap* with the following entries:

$$\begin{aligned} predefined\_id &\mapsto mk\text{-}ClassDerivedObject(class_1, \{\ldots\}) \\ virtual\_id &\mapsto mk\text{-}VirtualObject(\{mk\text{-}ClassDerivedObject(class_1, \{\ldots\}), \ldots\}) \end{aligned}$$

will also create multiple cases, including one case which removes the virtual object identity, leaving:

$$predefined\_id \mapsto mk\text{-}ClassDerivedObject(class_1, \{\ldots\})$$

and also changes entries in the *Refs* map of the form

$$ref_1 \mapsto virtual\_id$$

to

$$ref_1 \mapsto predifined\_id$$

In this second example, the existence of the virtual identity has been retrospectively overwritten by the predefined object.

Processing the message

With the receiver and arguments processed, and the class of the receiver determined, it is possible to proceed with the message send itself. This process is set in motion by the *process_message* function. It takes, as arguments, the receiver class, the processed receiver and argument object identities, the message selector, the 'positional' marker attached to the original message send, and the dataflow environment. In return it provides a set of possible dataflow environments representing different control paths through the invoked method.

The major part of this function is taken up in determining whether or not this is a recursive message send. This question is answered by constructing a '*recurs*' from elements of the message send, an action performed by *extract_recurs*, and comparing it with entries already recorded in the environment's method invocation record (*mir*), an action performed by *equiv_recurs*. Both these functions are defined and described in the following section, §4.5, along with the definition of the *recurs* entry itself. It is here that the message expression marker becomes significant. The marker, which has been passed along from the original message send expression, is used to restrict the detection of recursion to recursive invocations originating from the same expression.

If the test determines that this is a recursive send from which no more information may be usefully derived, then the current case being analysed must be terminated and control allowed to pass to the analysis of another potential case, or even to another method. This action is implied by the function *recursion_detected*, whose definition, like that of *does_not_understand*, has not been given.

In the case where no recursion is detected, the *recurs* extracted must be added to the *mir* kept by the dataflow environment before passing control to functions designed to handle the two kinds of methods, user-defined and primitive, *DfInvocation* and *DfPrimitive* respectively.

$$process\_message : ClassName \times Oop \times Selector \times Oop^{*} \times Id \times DfEnv \rightarrow DfSet$$

$$process\_message(rcvr\_class, rcvr\_oop, sel, arg\_oops, marker, \delta) \quad \triangleq$$
$$\quad \mathsf{let}\ current\_recurs = extract\_recurs(marker, sel, [rcvr\_oop] \frown arg\_oops, \delta)$$
$$\quad\quad \mathsf{in}$$
$$\quad \mathsf{if}\ \exists\ recurs \in \mathsf{elems}\, mir(\delta) \cdot equiv\_recurs(current\_recurs, recurs)$$
$$\quad \mathsf{then}\ recursion\_detected(current\_recurs, \delta)$$
$$\quad \mathsf{else\ let}\ \delta' = \mu(\delta, mir \mapsto mir(\delta) \frown [current\_recurs])$$
$$\quad\quad\quad \mathsf{in}$$
$$\quad\quad \mathsf{if}\ Methods(program(rcvr\_class))(sel) \in UserMethod$$
$$\quad\quad \mathsf{then}\ DfInvocation(rcvr\_class, rcvr\_oop, sel, arg\_oops, \delta')$$
$$\quad\quad \mathsf{else}\ DfPrimitive(rcvr\_class, rcvr\_oop, sel, arg\_oops, \delta')$$

DfInvocation deals with the invocation of a user-defined method. With the ultimate aim of analysing the body of the method using *DfExpression*, most of this function is spent setting up the dataflow environment to take account of the new context of the method invocation. After the correct method has been extracted from the 'global' program code, the first major task to be done, after tagging the parameters and temporaries (*tag_method_variables*), is to reset the self object to be the receiving object. This is performed by the *assign* function, and yields the first updated environment, δ'. The second update, δ'', is produced by taking the formal parameters of the invoked method and *assign*ing the message arguments to them using *pass_args*. The final update introduces the method's temporary variables, initialised to nil by *add_nil_objects*. With the dataflow environment, δ''', set up, it is then possible to analyse the invoked method body using *DfExpression*.

$$DfInvocation : ClassName \times Oop \times Selector \times Oop^* \times DfEnv \rightarrow DfSet$$

$$
\begin{array}{l}
DfInvocation(class, rcvr\_oop, sel, arg\_oops, \delta) \quad \triangleq \\
\quad \textsf{let } method = Methods(program(class))(sel) \\
\qquad \textsf{in} \\
\quad \textsf{let } method' = tag\_method\_variables(method, \sigma(\delta)) \\
\qquad \textsf{in} \\
\quad \textsf{let } \delta' = \mu(\delta, \sigma \mapsto assign(self, rcvr\_oop, \sigma(\delta))) \\
\qquad \textsf{in} \\
\quad \textsf{let } params = Params(method') \\
\qquad \textsf{in} \\
\quad \textsf{let } \delta'' = \mu(\delta', \sigma \mapsto pass\_args(params, arg\_oops, \sigma(\delta'))) \\
\qquad \textsf{in} \\
\quad \textsf{let } temps = Temps(method') \\
\qquad \textsf{in} \\
\quad \textsf{let } \delta''' = \mu(\delta'', \sigma \mapsto add\_nil\_objects(temps, \sigma(\delta''))) \\
\qquad \textsf{in} \\
\quad DfExpression\,[\![Body(method')]\!]\,\delta'''
\end{array}
$$

Pass_args takes a sequence of formal parameters and a sequence of argument object identities, and returns an updated object space, with each argument identity assigned to the corresponding formal parameter. Naturally there is a precondition that the number of arguments should equal the number of parameters to be assigned to. Strictly, this should be guaranteed by simple syntactic analysis of the code, given that, in Smalltalk-80 at least, the required number of arguments for a given method can be derived directly from the form of the method selector.

The function itself is defined recursively, with the terminating case being two empty sequences, when the object space is returned unchanged. The general case takes the head from each sequence and *assign*s the argument object identity to the formal parameter. The resulting updated object space is then used in the recursive call, applied to the tail of each sequence.

$$pass\_args : ParamId^* \times Oop^* \times ObjectSpace \rightarrow ObjectSpace$$

$$
\begin{array}{l}
pass\_args(params, arg\_oops, \sigma) \quad \triangleq \\
\quad \textsf{if } params = [\,] \\
\quad \textsf{then } \sigma \\
\quad \textsf{else let } \sigma' = assign(\textsf{hd}\, params, \textsf{hd}\, arg\_oops, \sigma) \\
\qquad\qquad \textsf{in} \\
\qquad pass\_args(\textsf{tl}\, params, \textsf{tl}\, arg\_oops, \sigma')
\end{array}
$$

$$\textsf{pre len}\, params = \textsf{len}\, arg\_oops$$

The definition of *DfPrimitive* will be delayed until the full introduction of primitive methods in Chapter 5.

$$DfPrimitive : ClassName \times Oop \times Selector \times Oop^{*} \times DfEnv \rightarrow DfSet$$

$$DfPrimitive(class, rcvr\_oop, sel, arg\_oops, \delta) \quad \triangleq$$

see Chapter 5

Post-message restoration

After a method invocation has been completed, from the analysis of the receiver and argument expressions through to the analysis of the method body, it is necessary to restore some aspects of the dataflow environment to take account of the change of context back to that of the invoking method. This is the purpose of *merge_envs*.

Merge_envs takes the dataflow environment as it was before the message send, together with the environment returned by the invoked method, and returns an environment derived mainly from the post-message environment, but with the self object and the method invocation record restored to their pre-message states.

For the duration of the message send, and in particular the invoked method, the self object is linked to the receiver object of the message send. Now that the method has returned a result, it is necessary to re-assign the original self object. In other words, to change from

$$self \mapsto rcvr\_oop$$

back to

$$self \mapsto self\_oop$$

and so re-establish the correct self object. Also, once the invoked method has returned, it is no longer a source for potential recursive sends involving itself. As a result its entry in the method invocation record must be removed. This is done by simply restoring the record to its pre-message send state, the state recorded in the pre-message environment.

The resulting dataflow environment is constructed from the updated object space, the method invocation record taken from the pre-message environment, and finally the *last* field taken from the post-message environment, which provides the result of the message send.

$$
\begin{aligned}
&\mathit{merge\_envs} : \mathit{DfEnv} \times \mathit{DfEnv} \to \mathit{DfEnv}\\
&\mathit{merge\_envs}(\mathit{pre\_msg\_}\delta, \mathit{post\_msg\_}\delta) \quad \triangleq\\
&\quad \text{let } \mathit{original\_self} = \mathit{Refs}(\sigma(\mathit{pre\_msg\_}\delta))(\mathit{self})\\
&\quad\quad \text{in}\\
&\quad \text{let } \sigma' = \mathit{assign}(\mathit{self}, \mathit{original\_self}, \sigma(\mathit{post\_msg\_}\delta))\\
&\quad\quad \text{in}\\
&\quad \text{let } \mathit{mir}' = \mathit{mir}(\mathit{pre\_msg\_}\delta)\\
&\quad\quad \text{in}\\
&\quad \text{let } \mathit{last}' = \mathit{last}(\mathit{post\_msg\_}\delta)\\
&\quad\quad \text{in}\\
&\quad \mathit{mk\text{-}DfEnv}(\sigma', \mathit{mir}', \mathit{last}')
\end{aligned}
$$

On return from a method, those temporary variables and parameters used for that method may be removed from the object space. In addition, those objects which become garbage as a result of the removal of these variables and parameters can also be removed. Because the temporary variables and parameters are re-tagged for each invocation of a method, their removal or otherwise from the object space has no effect on the rest of the analysis mechanism. Likewise, with the instance variables of an object, their personalised tagging ensures that when their object becomes garbage, they can safely be removed from the object space. What is happening, in fact, is a form of implicit garbage collecting, assumed to take place whenever an object is assigned to a variable (the previously assigned object may now be garbage), or whenever a method returns a result (the temporaries and parameters are no longer in scope).

The functions defined in this section have covered the analysis process from the level of classes through methods and down to individual expressions. There is, however, one important aspect which has not been dealt with, namely a guarantee that the analysis process will terminate. For this to be assured, it is necessary to detect, and act upon recursive message sends. The way in which this task is performed is described next.

4.5 Recursion control

Recursion control is vitally important for ensuring that the analysis process terminates, even if the code under analysis does not. Given the finite structure of the program code being analysed, only by following up recursive invocations of a given method can the analysis process run away with itself. Detecting recursive invocations is therefore necessary to prevent such a situation occurring and so guarantee the termination of the analysis process.

The implementation of recursion control is also important in determining under what conditions recursive invocations are singled out for termination, so controlling the amount and quality of information derived from such cases. There is a necessary balance to be made between catching all recursive calls and allowing some to be followed through, specifically those calls which, with different call-site information, may follow a different control path through the invoked method, and so extract new information.

```
class                Recursive
instance variables   recursionPoint

recursionPoint: aRecursive
    recursionPoint := aRecursive.
    ↑self

recur: aRecursive
    | anUninitialisedRecursive |
    anUninitialisedRecursive := self class new.
    aRecursive recursionPoint: anUninitialisedRecursive.
    self recur: anUninitialisedRecursive.
    ↑self
```

Figure 4.1: Recursion gone wild. An example of the effect of argument passing by reference on the practicalities of recursion control.

As has already been mentioned, the detection and control of recursion is handled through the method invocation record (*mir*). It is the structure of the main element of this record, the *recurs*, together with the associated comparison functions, which determines the sensitivity of the recursion detection mechanism. At its simplest, a *recurs* may be designed to contain just a record of which method is being invoked. Detecting recursion would then consist of a comparison between the method to be invoked and those methods which have already been invoked but which have yet to complete.

Unfortunately such a simplistic approach would have a serious limiting effect on the amount of code analysed, and hence the quantity of information extracted. For example, with the introduction of blocks (Chapter 6) it would have an adverse effect on the use of control-flow methods: the analysis of the following expression

```
rcvr ifTrue: [ aThing testThing ifTrue: [ aThing doSomeThing ]]
```

would terminate with the ifTrue: message send inside the block, under the simplistic assumption that it is a recursive call which will not yield any further information.

A more complex detection system would involve recording more information about the message send in the *recurs*, including details of the form and classes of the receiver and argument objects. Detecting recursion would then involve a more careful comparison of a potential method invocation with previous invocations. Taken to an extreme, such a system could attempt to record the whole of the system state which is accessible through the message send arguments. By recording the state of the receiver and argument objects, together with all the objects accessible through their instance variables and the instance variables' instance variables etc., a 'snapshot' of the relevant parts of the object space could be made.

Unfortunately recursion control based on an exhaustive 'snapshot' would not work. Such a system would depend for its effectiveness on limiting the changes to the objects accessible through the instance variables of the message expression's arguments. The accessible objects can be considered to form a sequence of trees, each rooted in one of the message expression's arguments. Each branch would represent the objects accessible through one instance variable, and each set of branches originating from one node would represent the instance variables of the object represented by that node. Each branch would eventually terminate with an object which had no instance variables, or by an instance variable which referenced an object already represented in that branch.

Recursion control based on such a structure would fail if the structure were to be susceptible to change at one of the leaves of one of the trees through a non-recursive action, and if that new leaf could be changed through a subsequent recursive invocation of the method under analysis. In other words, if, between recursive invocations, one of the trees grew through a non-recursive action, and if such an action could be repeated arbitrarily, then the recursion control would fail.

Figure 4.1 shows just such a situation. The important method in this example is recur:, which is sent to an instance of class Recursive with another Recursive as an argument:

```
|temp|
temp := Recursive new.
(Recursive new recursionPoint: temp) recur: temp
```

The key to the technique is that the main Recursive has a single instance variable which is itself a Recursive. This way, when the message is first sent, it is possible for the receiver Recursive to share its instance variable object with the argument — in fact, initially, the Recursive object referenced by the receiver's instance variable *is* the argument. However the argument Recursive is uninitialised — its instance variable is still set to reference the 'undefined' object, nil. The following diagram shows the structure of *recurs* derived from an initial recur: message send. Those elements marked by a subscript x are derived from the same object.

a Recursive	recur:	a Recursive$_x$
↓		↓
a Recursive$_x$		an Undefined Object
↓		
an Undefined Object		

When the method recur: is invoked with these arguments, a new Recursive is created. It is assigned, uninitialised, to the instance variable of the argument object. However, because the argument object is also referenced by the instance variable of the receiver this assignment will also extend the set of objects accessible through that receiver. This occurs because the argument objects are passed by reference. The subsequent recursive message send retains the original receiving object, but now has the new Recursive as the argument object. Such an invocation would seem to fit all the main criteria for recursion detection, namely that no more useful information can be extracted from this case. However, using the exhaustive 'snapshot' technique, the two invocations would be different, as a look at the 'snapshot' for the next recursive invocation shows (again, *recurs* extracted from the same object are indicated by a subscripted x):

a Recursive	recur:	a Recursive$_x$
$\downarrow$		$\downarrow$
a Recursive		an Undefined Object
$\downarrow$		
a Recursive$_x$		
$\downarrow$		
an Undefined Object		

The argument object is now referenced by the instance variable of the middle Recursive on the left hand side. Such a situation would continue indefinitely, with the recursion control allowing each subsequent invocation because of the creeping growth of one of the 'accessibility' trees.

Clearly the example above has proved that an exhaustive 'snapshot' approach of recording, at least, the class of all accessible objects cannot guarantee the termination of the analysis system. However a compromise between the two extremes discussed so far — a restricted form of the 'snapshot' method — would work. By imposing a maximum depth on the 'accessibility' trees, the number of possible states recorded by such trees would be limited, and while the amount of information extracted by the system would be greater than that possible with the initial simplistic system, the problem of creeping branches would be avoided. The limit placed on the depth of the 'accessibility' trees would naturally have to be an implementation decision.

4.5.1 The recursion environment

The basic element of the method invocation record, the *recurs*, contains both the selector of the method to be invoked and a sequence of structures, *ArgStruct*s, recording information extracted from the receiver and message arguments. In addition it also records the source of the originating message expression, so that only recursive invocations instigated from the same message send in the same method body are considered. Given that a program contains a finite amount of code, and hence a finite number of message send expressions, this will not affect the termination or otherwise of the analysis mechanism. The originating message send can be nil if this is a base case generated by *DfMethod* (page 105), but for other method invocations, the information is stored as a *MessageSource*. *Recurs* is subject to an invariant which will be defined at the end of the current section.

$$\begin{array}{rrrcl} \mathit{Recurs} & :: & \mathit{Source} & : & [\mathit{MessageSource}] \\ & & \mathit{Sel} & : & \mathit{Selector} \\ & & \mathit{Args} & : & \mathit{ArgStruct}^* \end{array}$$

With the introduction of blocks (Chapter 6), the *MessageSource* will come in two forms, with a second form to be used in connection with blocks. For this initial system, however, all message sources will be found in method bodies, and hence all *MessageSource*s will be *MethodSource*s. Each *MethodSource* must contain at least a class name and selector to determine which method contains the relevant message expression. However, in order to distinguish the correct message expression from the other message expressions within that method, the message's *Marker* field is used (page 89).

$MessageSource = MethodSource$

$$\begin{aligned} MethodSource :: \ Marker &: Id \\ Sel &: Selector \\ Class &: ClassName \end{aligned}$$

The information recorded about the receiver and argument objects, the sequence of *ArgStruct*s, is kept in the order in which it occurs in the message expression. Each *ArgStruct* records the class of its corresponding object, together with information about the state of that object's instance variables (the object's 'accessibility' tree). In this way, a single *recurs* can record information about all those objects which are potentially accessible, directly or indirectly, from both the originating message expression and the invoked method, subject to an imposed limit. By so doing, a *recurs* can record details of elements which are capable of having an effect on the way the invoked method behaves. Unfortunately such a record cannot be exhaustive. As was shown in the introduction to this section, the depth of the tree which these *ArgStruct*s form must be limited. This is represented by an invariant on the *Recurs*, which is defined after the definition of the *ArgStruct* has been completed.

Each individual *ArgStruct* can either be a *ClassStruct* or be represented by the constant VOOP. These two possibilities correspond to the two kinds of objects which can occur (*Object*, defined on page 94), namely *ClassDerivedObject* and *VirtualObject*. In this way a *recurs* can record objects whose structure has been resolved (class-derived objects) as well as those which remain 'virtual'.

$ArgStruct = ClassStruct \ \cup \ \{\text{VOOP}\}$

While a virtual object is represented by the constant, VOOP, the *ClassStruct* records both the class and instance variables of a class-derived object. Each instance variable of the object, in turn, records its own state as an *ArgStruct*, with the ultimate outcome being a tree of *ArgStruct*s. For convenience nil is used in the *Inst_vars* field to mark the lower limit of the 'accessibility' trees, imposed by the invariant function defined for the *Recurs*.

$$\begin{aligned} ClassStruct :: \quad Class &: ClassName \\ Inst\_vars &: [InstVar \xrightarrow{m} ArgStruct] \end{aligned}$$

As was mentioned above, it is necessary to limit the depth of the 'accessibility' trees represented by the *ArgStruct*s. This is represented by an invariant on the *Recurs*.

The invariant function defined for *Recurs* simply ensures that the trees of *ArgStruct*s descending from each of the *recurs*' arguments is no deeper than a predefined depth supplied by the constant DEPTH_LIMIT. The depth of each tree is calculated by the *depth* function.

$$\begin{aligned} &inv\text{-}Recurs(mk\text{-}Recurs(source, sel, args)) \quad \triangleq \\ &\qquad \forall\, arg \in \mathsf{elems}\, args \cdot depth(arg) \leq \text{DEPTH\_LIMIT} \end{aligned}$$

where

$\text{DEPTH\_LIMIT} \in \mathbb{N}$

Depth is defined on a case by case basis. The depth of a tree is calculated by incrementing a count for each recursive call of *depth*, starting from one (1) for the root. The result returned by *depth* is a natural number.

$$depth : ArgStruct \to \mathbb{N}$$

For those *ArgStruct*s which form leaves in the tree, the function returns one (1) for the level reached. This applies to both VOOPs and *ClassStruct*s with their *Inst_vars* field set to nil.

$$depth(\text{VOOP}) \quad \triangleq \quad 1$$

$$depth(mk\text{-}ClassStruct(class, \mathsf{nil})) \quad \triangleq \quad 1$$

For those *ClassStruct*s which have a non-nil *Inst_vars* field, *depth* returns one (1) if the map is empty, in which case it is a leaf anyway. If the *Inst_vars* map is non empty, *depth* is called recursively on each of the *ArgStruct*s descending from this node. The value returned by each of these calls is incremented by one to allow for the current level, and of the resulting set of depths the maximum is chosen and returned.

$$\begin{array}{l} depth(mk\text{-}ClassStruct(class, inst\_vars)) \quad \triangleq \\ \quad \mathsf{if}\ inst\_vars = \{\,\} \\ \quad \mathsf{then}\ 1 \\ \quad \mathsf{else\ let}\ tree\_depths = \{1 + depth(inst\_var) \mid inst\_var \in inst\_vars\} \\ \qquad\quad \mathsf{in} \\ \qquad \iota\ max\_depth \cdot max\_depth \in tree\_depths \\ \qquad\qquad\qquad\qquad \wedge\ \forall\ tree\_depth \in tree\_depths \cdot max\_depth \geq tree\_depth \end{array}$$

The overall effect of the *depth* function is that as the recursion unwinds after reaching the leaves of the 'accessibility' tree, the depth value is incremented until the root is reached and the overall maximum depth is determined.

In the following section the techniques of extraction and comparison of the *recurs* structures described above, will be defined.

4.5.2 Extracting, recording and testing for recursion

Before an invoked method can be analysed, a record must be made of the manner of its invocation. This is carried out by the following function, *extract_recurs*, which, given a method selector, receiver and arguments, in the form of a sequence of object identities, together with an object space, returns a *recurs*. This initial function delegates most of its task to the functions *extract_source* and *extract_arg_seq*. It then combines the resulting message source and sequence of *ArgStruct*s with the method selector to form the required *recurs*.

$$extract\_recurs : (Id \cup \{\mathsf{nil}\}) \times Selector \times Oop^* \times DfEnv \rightarrow Recurs$$

$$extract\_recurs(marker, sel, arg\_oops, \delta) \quad \triangleq$$
$\quad$ let $source = extract\_source(marker, mir(\delta))$
$\quad\quad$ in
$\quad$ let $arg\_struct\_seq = extract\_arg\_seq(arg\_oops, \sigma(\delta))$
$\quad\quad$ in
$\quad$ $mk\text{-}Recurs(source, sel, arg\_struct\_seq)$

Extract_source must first check to see if there is a source to be extracted. If this is an artificially induced method invocation, induced by *DfMethod* (page 105), then the method invocation record will be empty. For this initial case, the message source is recorded as nil. If there is a source to be extracted, all the relevant details are taken from the last *mir* entry, which will be the invocation record of the invoking method, and used to construct a *MethodSource*, currently the only form of message record.

$$extract\_source : (Id \cup \{\mathsf{nil}\}) \times MethodInvocationRecord \rightarrow (MessageSource \cup \{\mathsf{nil}\})$$

$$extract\_source(marker, mir) \quad \triangleq$$
$\quad$ if $mir = [\,]$
$\quad$ then nil
$\quad$ else let $last\_recurs = mir[\mathsf{len}\, mir]$
$\quad\quad\quad$ in
$\quad\quad$ let $last\_sel = Sel(last\_recurs)$
$\quad\quad\quad$ in
$\quad\quad$ let $last\_class = Class(\mathsf{hd}\, Args(last\_recurs))$
$\quad\quad\quad$ in
$\quad\quad$ $mk\text{-}MethodSource(marker, last\_sel, last\_class)$

The task of creating a sequence of *ArgStruct*s from a sequence of object identities is performed by *extract_arg_seq*, by recursing through the sequence provided. The base case is an empty sequence of object identities, yielding an empty sequence of *ArgStruct*s. The general case takes the head of the argument sequence, processes it with *extract_arg*, and concatenates the result with the result of processing the rest of the sequence. The call to *extract_arg* takes not only the argument to be processed and the object space, but also the depth limit imposed on the *ArgStruct* 'accessibility' tree. The value of this argument, a natural number, is provided by the constant DEPTH_LIMIT.

$$extract\_arg\_seq : Oop^* \times ObjectSpace \rightarrow ArgStruct^*$$

$$extract\_arg\_seq(arg\_oops, \sigma) \quad \triangleq$$

$$\begin{array}{l}
\quad \textsf{if } arg_oops = [\,] \\
\quad \textsf{then } [\,] \\
\quad \textsf{else let } hd_arg_oop = \textsf{hd}\, arg_oops \\
\qquad\qquad \textsf{in} \\
\qquad [extract_arg(hd_arg_oop, \sigma, \text{DEPTH_LIMIT})] \\
\qquad\qquad \frown \\
\qquad extract_arg_seq(\textsf{tl}\, arg_oops, \sigma)
\end{array}$$

Extract_arg is defined to take account of the various cases which may arise during the creation of an *ArgStruct*. It starts by extracting the object body of the object identity provided, from the object space. With the object body it is possible to distinguish between virtual and class-derived objects. If it is a virtual object then the corresponding *ArgStruct* returned is the constant VOOP.[11] The second distinction is made if the depth limit, decremented for each descending call of *extract_arg*, has reached one (1). In this latter case the *ArgStruct* is made up with nil for the instance variable map to indicate that the depth limit has been reached.

For the remaining case, namely a class-derived object which has not reached the depth limit, the depth limit is decremented for the processing of any instance variables by *extract_arg*. The resulting *ClassStruct* is constructed by taking the class of the current object together with the result of processing any instance variables.

$$extract\_arg : Oop \times ObjectSpace \times \mathbb{N} \rightarrow ArgStruct$$

$$extract\_arg(arg\_oop, \sigma, depth\_lim) \quad \triangleq$$

$$\begin{array}{l}
\quad \textsf{let } arg_object = OopMap(\sigma)(arg_oop) \\
\qquad \textsf{in} \\
\quad \textsf{if } arg_object \in VirtualObject \\
\quad \textsf{then VOOP} \\
\quad \textsf{else if } depth_lim = 1 \\
\qquad \textsf{then } mk\text{-}ClassStruct(Class(arg_object), \textsf{nil}) \\
\qquad \textsf{else let } arg_class = Class(arg_object) \\
\qquad\qquad \textsf{in} \\
\qquad\quad \textsf{let } inst_vars = \{i_v \mapsto extract_arg(Refs(\sigma)(i_v), \sigma, depth_lim - 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid i_v \in Inst_vars(arg_object)\} \\
\qquad\qquad \textsf{in} \\
\qquad\quad mk\text{-}ClassStruct(arg_class, inst_vars)
\end{array}$$

[11] Note that the receiving object will never be a virtual object. If it was virtual before the message send, then it will have been resolved during the analysis of the receiver and argument expressions of the message send.

By applying *extract_arg* to a single argument object from the message send under analysis, a tree of *ArgStruct*s can be constructed, with each element in the tree representing an object potentially accessible through that top message argument. Because of the artificially imposed limit on the depth of the tree structures, cycles of object references, caused by one object referencing another object further up the same branch, will have no effect on the termination or otherwise of the extraction mechanism.

Detecting recursion is simply a matter of comparing *recurs*. This is performed by *equiv_recurs*. The process of comparing two *recurs*, involves establishing that both *recurs* are dealing with the same message send. It then goes on to check if the receiver and arguments of the old and new message sends are of the same class. A minimalist comparison could stop there, but *equiv_recurs* takes the comparison further by testing the sources of the two message sends and comparing the information recorded by each *recurs*' argument structures.

In more detail, *equiv_recurs* takes a new *recurs*, *new_recurs*, and compares it with one taken from the method invocation record, *old_recurs*. The comparison takes the form of checking their respective method selectors, the sources of their respective message sends, and finally their respective arguments. This last task is delegated to the function *equiv_args*.

$$
\begin{aligned}
&\mathit{equiv\_recurs} : \mathit{Recurs} \times \mathit{Recurs} \rightarrow \mathbb{B} \\
&\mathit{equiv\_recurs}(\mathit{new\_recurs}, \mathit{old\_recurs}) \quad \underline{\triangle} \\
&\qquad (\mathit{Sel}(\mathit{new\_recurs}) = \mathit{Sel}(\mathit{old\_recurs})) \\
&\qquad\quad \wedge (\mathit{Source}(\mathit{new\_recurs}) = \mathit{Source}(\mathit{old\_recurs})) \\
&\qquad\qquad \wedge\, \mathit{equiv\_args}(\mathit{Args}(\mathit{new\_recurs}), \mathit{Args}(\mathit{old\_recurs}))
\end{aligned}
$$

In order to help maximise the amount of information derived through the analysis process, only recursive message sends are dealt with by the recursion detection process, rather than recursive method invocations. In this way, a method, recursively invoked through message sends in two separate sections of code will not cause the analysis process to stop. Instead, only methods invoked recursively by the same message expression in the same piece of code will be capable of causing a premature halt due to the detection of a recursive message send.

By restricting the detection of recursion to method invocations from the same expression, it becomes possible to analyse code after a self-referential message send within a given method. This can be seen in an example using blocks.[12]

[12]The equivalent example using our initial block-free language would place the 'block' code, as methods, in the boolean classes True and False, in a manner described by Wolczko, [Wol88] Chapter 5, page 108.

```
foo: bar
    "An example of a method with a recursive call. bar has an instance
    variable 'factor'."
    bar factor isNil
        ifTrue: [ ↑self ]
        ifFalse: [ self foo: bar alterFactor.
                   bar factor: 42.
                   ↑self ]
```

When the method has been invoked, it will have an entry in the method invocation record. The interesting part of the method is the self-recursive call in the 'false' half of the ifTrue:ifFalse:. Under a system which ignores the source of the message send, there is a chance that the recursive foo: message will match with the foo: entry already in the *mir*. If this is the case, a recursive message send would be signalled, and the analysis for this case would be terminated, losing any information that may have been derived from the analysis of the following expression, bar factor: 42. Instead, by recording the location of each message send, and hence the source of each method invocation, a further foo: message is allowed. In this case, two alternatives may be followed by the analysis mechanism: one which follows the control path down the 'false' side, this time ending with the signaling of a recursive send, and another in which the 'true' branch is followed, resulting in the method returning. By allowing this additional recursive message send, a case has been followed in which the recursion has terminated, effectively following the base case for the recursion, and in which the code following the recursive send can be analysed, namely bar factor: 42.

The comparison of the two sequences of *ArgStruct*s, performed by *equiv_args*, starts with a simple comparison of their respective lengths — only sequences of the same length could possibly be linked with the same method selector. The test then continues, by applying *equiv_arg_struct* to each corresponding pair of arguments, taking one argument from each sequence. In contrast to the rest of the analysis system, the comparison of two argument structures is made on the basis of the classes of the objects comprising the two arguments, rather than the objects themselves. This is a necessary compromise in order to guarantee termination. It would be trivially easy to construct a non-terminating recursive method which, with a comparison based on object identities rather than object classes, would not be detected by the recursion control mechanism. For example

```
reproduction
    "An example of a terminally recursive method."
    self class new reproduction.
    ↑self
```

would never terminate, and with a new receiving object being created for each recursive message send, the recursion would not be detected if the comparison between different invocations was made on the basis on object identities rather than object classes.

$$equiv\_args : ArgStruct^* \times ArgStruct^* \rightarrow \mathbb{B}$$

$$equiv\_args(new\_args, old\_args) \quad \triangleq$$
$$\quad \mathsf{len}\, new\_args = \mathsf{len}\, old\_args$$
$$\quad\quad \wedge \forall\ i \in \mathsf{inds}\, new\_args \cdot equiv\_arg\_struct(new\_args[i], old\_args[i])$$

The definition of *equiv_arg_struct* is made on a case-by-case basis, depending on whether the arguments are *ClassStruct*s or VOOPs.

$$equiv\_arg\_struct : ArgStruct \times ArgStruct \rightarrow \mathbb{B}$$

The first two cases deal with two *ClassStruct*s, one case dealing with non-nil instance variable maps, the other dealing with the instance variable maps both set to nil. The first of these cases, dealing with non-nil maps, has two structures which have not been terminated prematurely. In this general case, the comparison of the two structures entails a check of their respective classes, and then, if the first test succeeds, a further test of their corresponding instance variables.

In order to compare the two sets of instance variables it is necessary to pick out corresponding variables from both structures. This part of the test is only meaningful if the two structures are derived from objects of the same class (the first part of the test), when the two sets of instance variables will match. In order to select matching pairs from the two sets of variables, it is necessary to strip off the variables' tagging to reveal their original form. This is done by going through all the instance variables of the newer structure, and selecting the corresponding instance variable from the older structure, with correspondence being based on the results of applying *untag_inst_var* (page 108) to the two variables. For each pair of matching variables extracted, it is necessary to extract the *ArgStruct*s corresponding to those variables from the current structures, and process them with *equiv_arg_struct*.

$$equiv\_arg\_struct(mk\text{-}ClassStruct(new\_class, new\_ivs),$$
$$\qquad\qquad mk\text{-}ClassStruct(old\_class, old\_ivs)) \quad \triangleq$$
$$\quad new\_class = old\_class$$
$$\quad \wedge \forall\ iv_{new} \in \mathsf{dom}\, new\_ivs \cdot$$
$$\quad\quad \exists\ iv_{old} \in \mathsf{dom}\, old\_ivs \cdot$$
$$\quad\quad\quad untag\_inst\_var(iv_{new}) = untag\_inst\_var(iv_{old})$$
$$\quad\quad\quad \wedge\ equiv\_arg\_struct(new\_ivs(iv_{new}), old\_ivs(iv_{old}))$$

where

$$new\_ivs \neq \mathsf{nil} \wedge old\_ivs \neq \mathsf{nil}$$

For the case where the depth limit has been reached, and both the structures have a nil instance variable map, the comparison criteria must, by necessity, be relaxed.[13] For this case it is only necessary for the respective classes to be the same.

[13] The case when only one of the structures has a nil instance variable map will not occur, because only *ClassStruct*s at the same depth are compared, and only *ClassStruct*s at the depth limit will have a nil instance variable map.

$$
\begin{aligned}
&\mathit{equiv\_arg\_struct}(\mathit{mk\text{-}ClassStruct}(\mathit{new\_class}, \mathit{new\_ivs}),\\
&\qquad\qquad\qquad\qquad \mathit{mk\text{-}ClassStruct}(\mathit{old\_class}, \mathit{old\_ivs})) \quad \triangleq\\
&\quad \mathit{new\_class} = \mathit{old\_class}
\end{aligned}
$$

where

$\mathit{new\_ivs} = \mathsf{nil} \wedge \mathit{old\_ivs} = \mathsf{nil}$

For those cases involving a mix of *ClassStruct*s and VOOPs, the order of creation becomes important in determining the significance of the current point in the *recurs* structure. In the case when a newer *ClassStruct* is compared to an older VOOP, because the older VOOP represents an object which had not yet been fully resolved when it was recorded, it will have had the potential to be an instance of any class, including the class of the newer *ClassStruct*. As a result, any possible information to be derived from following through with the current message send on account of the *ClassStruct* in the newer *recurs* structure, would already have been gained through the older message send on account of the corresponding VOOP in the older *recurs* structure. This is because if the older element was significant with respect to the older message send, then all the information to be derived from the newer message send, through further message sends to the newer *ClassStruct*, would already have been extracted when the older VOOP was resolved while receiving the same messages. For example, consider the following mutually recursive methods:

```
bargTest: aBarg
    self specialTest
        ifTrue: [ ↑self ]
        ifFalse: [ ↑self foo: aBarg ]

foo: aBarg
    "A method which tests its argument."
    aBarg testMe
        ifTrue: [ ↑self ]
        ifFalse: [ aBarg alterMe.
                   self bargTest: aBarg.
                   ↑self insertMe: aBarg ]
```

If bargTest: is invoked with a virtual object as its argument, and the 'false' block is executed, the *recurs* recorded for the foo: message send will have a VOOP for the first argument, aBarg. Assuming that foo: results in a recursive send of bargTest:, and that the 'false' block is again executed, a *recurs* will be extracted for the foo message send for a second time. This time however, as a result of the testMe message received during the foo: method, the first argument, aBarg, will result in a *ClassStruct* rather than a VOOP. When a comparison is made between this current *recurs* and the older, still-active *recurs*, one key element of the comparison will be between the older VOOP and the newer *ClassStruct*. Because information can only be extracted through message sends, it follows that any information the newer element hopes to extract from the code will already have been extracted by the older, virtual element, the older element having received all the messages the newer element will receive.

Unfortunately this example also shows one of the limitations of the current recursion detection mechanism when dealing with mutually recursive methods. Recursion detection is limited to message sends originating from the same expression, and as such, recursion in the above example is detected for the second send of the foo: message in the body of bargTest:, which has been invoked from the body of foo:. In effect this means that the code following the bargTest: message in foo: will only be analysed if bargTest: returns via its 'true' path. This may not happen because of the assumptions on the self object which led to the invocation of foo: in the first place. Although one solution would be to follow through the recursive foo: message and hope that the test on aBarg provides a result through foo:'s 'true' path, so unraveling the recursion, this solution cannot be relied upon, given that it rests on the outcome of the test involving aBarg. In general, this is a problem relating to all mutually recursive methods, and means that with only the first recursive message send expression being detected and acted upon, some code may go unanalysed.

Returning to the definition of *equiv_arg_struct*, notwithstanding the problem with mutually recursive methods indicated above, given the potential of the older VOOP, it is clear that a newer *ClassStruct* must always match it, and the comparison must return true.

$$\mathit{equiv\_arg\_struct}(\mathit{mk\text{-}ClassStruct}(\mathit{new\_class}, \mathit{new\_ivs}), \text{VOOP}) \quad \triangleq \quad \mathsf{true}$$

In contrast with the previous case, if the newer structure is a VOOP, the reasoning works in reverse, with the newer virtual entry representing potential for information extraction beyond that which could have been realised by the older *ClassStruct*. As a result, the comparison must return false.

$$\mathit{equiv\_arg\_struct}(\text{VOOP}, \mathit{mk\text{-}ClassStruct}(\mathit{old\_class}, \mathit{old\_ivs})) \quad \triangleq \quad \mathsf{false}$$

The final case involves two VOOPs, hence two objects with equal potential for information extraction, and consequently, this case always returns true.

$$\mathit{equiv\_arg\_struct}(\text{VOOP}, \text{VOOP}) \quad \triangleq \quad \mathsf{true}$$

4.5.3 Guaranteeing termination

The termination of the analysis system is only a problem when recursive message sends are used. For non-recursive methods, the analysis of an individual method will terminate normally, and, given the finite number of methods in the program, the overall analysis process can be guaranteed to terminate.

With respect to the analysis mechanism, recursive methods, by their very nature, cannot be guaranteed to terminate, and so require some form of control in order to detect them and provide a means of prematurely terminating such cases. This is the task of the recursion control functions described in this section. It is therefore necessary to show that these recursion control functions are capable of handling recursive cases during the analysis process and so ensure that the process as a whole will complete in a finite length of time.

The task of showing the effectiveness of the recursion control process in guaranteeing termination can be broken down into two parts, firstly by showing that the recording of information for the method invocation record is itself a finite process, and secondly by showing that all recursive message sends will eventually be detected and dealt with.

Both these problems are rendered trivial by the restriction imposed in the construction of a *recurs*. The exhaustive 'snapshot' technique, described in the introduction to this section (page 123), has been shown by a counter example to be unable to guarantee termination. As a consequence, the 'accessibility' trees, which form a key element of the recursion control environment, have had a limit imposed on their maximum depth. The resulting 'stunted' trees will therefore have a known maximum number of states which the constituent nodes can assume. It follows that both the construction of the *recurs* will be finite in duration, and that the maximum number of recursive message sends before two *recurs* must, by necessity, match will also be finite. It therefore also follows that the whole dataflow analysis system must be a finite process.

4.6 Summary

In this chapter, a simple language has been introduced, and a basic dataflow analysis mechanism defined for it. In addition, a method for handling recursive message sends has been constructed. Finally, the whole analysis mechanism has been shown to be a finite process.

In the next chapter, primitive methods will be properly introduced, and the analysis mechanism will be extended to handle them.

Chapter 5

Adding primitives

The system introduced in the preceding chapter analyses a simple language, capable of the creation and manipulation of objects. The system analyses the program code by the technique of abstract interpretation — interpreting the code over the domain of abstract objects, primarily stored in an abstracted 'object space'. The system analyses each method in turn, interpreting each expression, and following through message sends to the methods which they may potentially invoke. The termination of the analysis mechanism is assured either by following through a method to the return of its result, or by the detection of a recursive message send.

However, in the simple system described so far, there are a set of fundamental operations which have not yet been dealt with, for example the ability to compare two object identities and test for their equality.[1] Such basic operations are normally provided by a set of special methods called the primitive methods (defined on page 88). These methods are implemented at a lower, machine-dependent level than the rest of the program code, and are consequently not directly accessible by the casual user. In Smalltalk-80, their presence is indicated by an identifying number, while in the system being presented here they are signified by an identifier corresponding to their method selector.

Because the primitive methods are provided as explicit fundamental functionality for the program, it is reasonable to assume that their effects may be designed into the analysis mechanism. In other words, it is reasonable to assume that primitive methods may be 'special-cased' by the dataflow analysis system. For the majority of primitives, this will mean creating a set of potential results (cases) based on the form of result specified for that primitive. However for a small number of essential primitives, the potential for exploiting their known effects can be taken further. It is the alterations to the analysis mechanism needed to allow for these key primitives that will be the focus of this chapter.

The following primitives have been chosen for special treatment, and form, in the opinion of the author, a representative sample, ranging from fundamental primitives to more general, 'useful' primitives:[2]

[1]In Smalltalk-80, the test for equality of two objects, determined by their identities, is known as object equivalence. In the rest of this chapter, and the following chapter, the terms 'object equivalence' and 'object identity equality', and their derivatives, will be used interchangeably.

[2]However they do not cover the basic functionality necessary to handle integers — which need an ordering

perform: Construct and send a message to the receiver of the perform: message, using the first argument as the message selector, with the rest of the arguments making up the arguments for the constructed message.

class Return the class of the receiving object.

become: Swap the receiver and the first argument so that all previous references to the receiver now reference the argument, and all previous references to the argument now reference the receiver.

== Test for object equivalence between the receiver and the argument. Object equivalence holds if the receiver and argument have the same object identity.

The introduction of the primitives, though catered for by the abstract syntax, does require some alterations to the object space. These alterations are, in fact, specific to the question of object equivalence, and are described and defined next.

5.1 Alterations to the Object Space

Alterations to the object space are needed to allow the introduction of a comparison primitive. The equivalence primitive takes two objects and compares their identities. As a result, the uniqueness or otherwise of an object must be carefully controlled. For predefined objects and concrete objects, their relative points of creation are known, and therefore each object's identity can be assumed unique. In contrast, the identity of a virtual object is not so certain.

In detail, a predefined object is made up of a predefined object identity together with a class-derived object body, and is explicitly 'created'[3] prior to the analysis process. As such, a predefined object has a unique identity. Likewise a concrete object, consisting of a concrete object identity and a class-derived object body, also assumes a unique identity when it is 'created' during the analysis process. The problem, therefore, lies with virtual objects, though not all virtual objects.

Virtual objects come in two varieties, unresolved virtual objects made from a virtual identity (*VirtualOop*) with a virtual object body (*VirtualObject*), and resolved virtual objects, which have retained their virtual identities but have replaced their virtual bodies with class-derived object bodies. The latter of these two forms represents an object which has at some stage been the recipient of a message, forcing it to undergo a process of resolution (*find_rcvr_class*, page 114) during which not only was its class determined (in order to determine which method to invoke), but also the uniqueness of its identity. In contrast, the identity of an unresolved virtual object remains,…well…, unresolved.

The problem for the equivalence primitive can be refined further by noting that the recipient of the equality message must be a fully-resolved object by virtue of the analysis mechanism. The problem, therefore, lies with unresolved virtual objects which are passed as the argument:

primitive — and arrays or other collection structures. The requirements for these structures will be discussed in §6.8

[3]It is worth pointing out again that 'creation' in this context is part of the abstract interpretation process.

anObject == anUnresolvedVirtualObject

Clearly when such an expression is analysed, it must yield at least two basic cases: one where the two objects are assumed to be equal, and one where they are assumed to be not equal. The problem is, how to record the latter case in order to ensure consistency with future comparisons.

Given that the problem is restricted to unresolved virtual objects, the solution must start with the definition of the virtual object body. Recalling the definition of an object body (*Object*) from §4.3 (page 94), it is either a virtual object or a class-derived object:

$$\mathit{Object} = \mathit{VirtualObject} \ \cup\ \mathit{ClassDerivedObject}$$

While the modifications necessary to handle the problem of equivalence leave the definition of the class-derived object unchanged, the definition of the virtual object must be extended.

The original definition of the virtual object (page 94) consisted solely of a tagged set of potential class-derived objects. In this modified version, the set of potential objects has been supplemented by a set of object identities. The intention is that this set of object identities will provide a record of those objects which have been compared with, and assumed *unequal to*, this unresolved virtual object.[4]

$$\begin{array}{lrcl} \mathit{VirtualObject} :: & \mathit{PossibleClass} & : & \mathit{ClassDerivedObject}\text{-set} \\ & \mathit{Neq\text{-}set} & : & \mathit{Oop}\text{-set} \end{array}$$

In order to make use of the extended definition of the virtual object, *prepare_virtual_object*, originally defined on page 99, has to be modified to initialise the set of 'non-equal' object identities to be empty. Apart from this addition, the modified definition performs the same tasks as the original version.

5.2 Alterations to the flow analysis

The alterations to the flow analysis functions defined for the basic system are mostly minor, dealing with the extra field in the virtual object definition. *Get_class_and_δ_set*, originally defined on page 114, is called when processing the receiving object during a message send to determine the recipient's class, and requires the inclusion of an extra field for the *VirtualObject* case. This is the first function to handle the modified definition during the case analysis on the structure of the the receiving object, and for the case we are interested in — that is, the case for a virtual object — the task is handed over to *resolve_v_obj* for each potential class-derived object recorded by the virtual object. At this point the set of 'non-equal' identities has not come into play. *Resolve_v_obj*, which remains unchanged from the original version (page 115), then passes on control to *resolve_v_obj_cases*, the function which deals with actually resolving the identity and structure of the virtual object.

[4]The completeness of this solution can be demonstrated by noting the following: Firstly, all object identities recorded in this set must, by virtue of the analysis mechanism, be fully-resolved identities, and secondly, in cases where the two objects are assumed to be equal, all references to the unresolved virtual object would be replaced with the 'equal' fully-resolved object. As such, the set will only contain the identities of fully resolved objects, and furthermore, the set will only exist as long as the virtual object remains unresolved.

The new definition of *resolve_v_obj_cases* has only one small modification when compared with the original version (page 115). The techniques of creating a single primary case, σ'', and a set of secondary cases, *old*σ*s* (equating the virtual object with class-derived objects already in the object space) are principally the same as used in the original version. The difference lies with the construction of the secondary cases. When considering equating the main virtual object with other objects, it is necessary, not only to ensure that such objects are either fully-resolved virtual objects or predefined objects, but also to ensure that such objects are not on the main virtual object's black list, namely the set of 'non-equal' object identities.

$$
\begin{array}{l}
\mathit{resolve\_v\_obj\_cases} : \mathit{Oop} \times \mathit{ClassDerivedObject} \times \mathit{ObjectSpace} \rightarrow \mathit{ObjectSpace}\text{-set} \\
\mathit{resolve\_v\_obj\_cases}(\mathit{voop}, \mathit{cdo}, \sigma) \quad \triangleq \\
\quad \text{let } \mathit{old}\sigma s = \{\mathit{equate\_oops}(\mathit{oop}, \mathit{voop}, \sigma') \\
\qquad\qquad\qquad\qquad \mid \mathit{oop} \in \text{dom}\,\mathit{OopMap}(\sigma) \\
\qquad\qquad\qquad\qquad\ \wedge \mathit{oop} \in \mathit{PredefinedOop} \cup \mathit{VirtualOop} \\
\qquad\qquad\qquad\qquad\ \wedge \mathit{OopMap}(\sigma)(\mathit{oop}) \in \mathit{ClassDerivedObject} \\
\qquad\qquad\qquad\qquad\ \wedge \mathit{Class}(\mathit{OopMap}(\sigma)(\mathit{oop})) = \mathit{Class}(\mathit{cdo}) \\
\qquad\qquad\qquad\qquad\ \wedge \mathit{oop} \notin \mathit{Neq\text{-}set}(\mathit{OopMap}(\sigma)(\mathit{voop}))\} \\
\qquad \text{in} \\
\quad \text{let } \mathit{cdo}' = \mu(\mathit{cdo}, \mathit{Inst\_vars} \mapsto \mathit{tag\_inst\_vars}(\mathit{Inst\_vars}(\mathit{cdo}), \sigma)) \\
\qquad \text{in} \\
\quad \text{let } \sigma' = \mu(\sigma, \mathit{OopMap} \mapsto \mathit{OopMap}(\sigma) \cup \{\mathit{voop} \mapsto \mathit{cdo}'\}) \\
\qquad \text{in} \\
\quad \text{let } \sigma'' = \mathit{add\_virtual\_objects}(\mathit{Inst\_vars}(\mathit{cdo}'), \sigma') \\
\qquad \text{in} \\
\quad \{\sigma''\} \cup \mathit{old}\sigma s
\end{array}
$$

The rest of the flow analysis functions defined in the previous chapter remain unchanged, and the basic principle behind the flow analysis process — the use of the object space and dataflow environment, together with the creation of 'cases' through the abstract interpretation of message sends — still holds good.

5.3 Analysing primitives

Now that the necessary (minor) modifications to the basic analysis system have been made, it is possible to present the definition of *DfPrimitive*, first glimpsed (briefly) on page 121. *DfPrimitive* is the primitive equivalent of *DfInvocation* (page 120), and follows a similar pattern of extracting the invoked method, initialising the self object and passing the arguments. The differences start when the action of the primitive, indicated by the *Primitive* field of the method, is extracted. Instead of invoking a method body with *DfExpression* (because there isn't one), control is instead passed to *Invoke_Primitive*, which deals with each primitive separately. The binding of the self object and the passing of parameters at this point is, at the moment, only included to maintain a level of generality. All the primitives currently dealt with by *Invoke_Primitive* handle the receiving object and arguments directly. It is assumed that future additions may deal with the objects through self and the parameter names, when the need to tag the parameter names should be considered.

$$DfPrimitive : ClassName \times Oop \times Selector \times Oop^* \times DfEnv \to DfSet$$

$DfPrimitive(class, rcvr\_oop, sel, arg\_oops, \delta) \quad \triangleq$

let $primitive\_method = Methods(program(class))(sel)$
in
let $\delta' = \mu(\delta, \sigma \mapsto assign(self, rcvr\_oop, \sigma(\delta)))$
in
let $params = Params(primitive\_method)$
in
let $\delta'' = \mu(\delta', \sigma \mapsto pass\_args(params, arg\_oops, \sigma(\delta')))$
in
let $prim\_id = Primitive(primitive\_method)$
in
$Invoke\_Primitive(prim\_id, rcvr\_oop, arg\_oops, \delta'')$

Invoke_Primitive selects the action to be performed by case analysis on the primitive identity (*primitive_id*) passed to it. Each primitive, from the set of primitives which the system can currently deal with, is processed by its own specialised function. Other primitives will eventually be dealt with on a similar individual basis. Because each primitive performs a unique task which is known before the analysis process commences, there can be no useful, practical generalised function to act as a default action for a primitive.

$$Invoke\_Primitive : PrimitiveId \times Oop \times Oop^* \times DfEnv \to DfSet$$

$Invoke\_Primitive(primitive\_id, rcvr\_oop, arg\_oops, \delta) \quad \triangleq$

cases $primitive\_id$ of
$perform \to evaluate\_message(rcvr\_oop, arg\_oops, \delta)$
$class \to find\_class(rcvr\_oop, \delta)$
$become \to become\_args(rcvr\_oop, arg\_oops, \delta)$
$equivalence \to test\_for\_equality(rcvr\_oop, arg\_oops, \delta)$
$\ldots$
end

5.3.1 Performing a message

The perform primitive is handled by *evaluate_message*. This function, which receives the identity of the receiver and argument objects together with a copy of the environment, can only proceed if the prospective selector is of a form which can be converted into a message selector. This is checked by *useful_selector*, which makes its decision based on the first argument (the selector) and the dataflow environment. If the argument is of a form which can be converted to a proper *Selector*, then this conversion is carried out by *extract_selector*. In most useful cases the prospective selector will come in the form of a literal symbol, indirectly via a variable, and hence the definition of both the testing and extraction functions will depend on the way in which such symbols are implemented. As a result, their definitions have not been given. Clearly, if the primitive is invoked with a virtual argument, there is no way the analysis can sensibly continue, and so such a case would have to be terminated.

Assuming that the prospective selector has been successfully extracted, it is then possible to begin the construction of the message send. The first step is to remove the unprocessed selector from the sequence of argument identities, after which the number of required arguments is tested against the number remaining in the sequence. This test relies on the function *number_of_params*, which assumes that the number of arguments required by a message can be deduced by the structure of the message selector, an assumption which holds for Smalltalk-80. If the numbers match, the message can be sent. If the numbers do not match, a primitive failure occurs. Primitive failures for both this test, and the test for a useful selector, are handled by *primitive_failure*. This has a similar effect to the *does_not_understand* function utilised by *DfMessage*, terminating the current case and returning control to another.

The constructed message send is eventually handled by *DfMessage*. As with all messages, the newly constructed message requires a marker to satisfy the method invocation record (*extract_recurs*, page 128). This is supplied by the marker associated with the perform primitive itself. In addition, due to the fact that message markers are only unique within a given method, it is necessary to record the source of the constructed message as the method which contained the perform: message. This task is handled by modifying the source extraction function, *extract_source*, which will be redefined shortly.

$evaluate\_message : Oop \times Oop^{*} \times DfEnv \rightarrow (\mathbb{B} \times DfSet)$

$evaluate\_message(rcvr\_oop, arg\_oops, \delta) \quad \triangleq$
    if $\neg$ $useful\_selector$(hd arg_oops, δ)
    then $primitive\_failure$('not a useful message selector',
                    $perform, rcvr\_oop, arg\_oops, \delta$)
    else let $sel = extract\_selector$(hd arg_oops, δ)
        in
      let $arg\_oops' =$ tl arg_oops
        in
      if $number\_of\_params(sel) \neq$ len tl arg_oops
      then $primitive\_failure$('wrong number of aguments',
                    $perform, rcvr\_oop, arg\_oops, \delta$)
      else let $marker = Marker(Source(mir(\delta)[$len $mir(\delta)]))$
          in
        $DfMessage(rcvr\_oop, sel, arg\_oops', marker, \delta)$

where

$arg\_oops \neq [\,]$

The signatures of the various functions used by *evaluate_message* are listed next. The definitions of both *useful_selector* and *extract_selector* have been omitted because of their dependence on how literal symbols are handled by the language implementation. *Number_of_params* is not defined because of its dependence on the convention chosen for constructing message selectors, although it assumes that the number of arguments required by a message selector can be determined from that selector.[5]

$useful\_selector : Oop \times DfEnv \rightarrow \mathbb{B}$
$extract\_selector : Oop \times DfEnv \rightarrow Selector$
$number\_of\_params : Selector \rightarrow \mathbb{N}$

As has already been mentioned, in order to ensure that the location of the message send, as recorded by the method invocation record, is specific to this message send, the relevant function, *extract_source* invoked by *extract_recurs*, has to be modified. With the original version of this function (page 128), the source (marker, source method selector, source class) of the constructed message send would be recorded as:

[5] An alternative method for determining the number of arguments would be to match the message selector with a method selector defined for the receiving object, and so deduce the required number of arguments from the method definition. However, to include the required tests to determine whether the receiver understands the message in the first place would add unnecessarily to the complexity of the *evaluate_message* function, given the reasonable assumption, derived from experience with Smalltalk-80, about the correspondence between selector structure and argument numbers.

mk-MethodSource(perform marker, perform, receiver class)

If the marker, derived from the perform: message, coincides with that of another perform: message in another method in the same class, there might be a clash if the two constructed messages use the same selector. This would occur simply because the 'source' of the constructed messages would be the perform primitive method of that class, and with the markers coinciding, the messages would be assumed to have come from the same location. The solution is to locate the constructed messages not with the perform primitive, but with the method in which the perform: message originally occurred:

mk-MethodSource(perform marker, perform's source method, perform's source class)

The function starts off in the same manner as the original, with a test on the method invocation record. The new version starts to differ at the point when the original constructs the *MethodSource*. In the new version there is a test to see if the identified source of the message send is a user-defined method. If it is, then the *MethodSource* is constructed as before. If it is not, then the source is clearly a primitive, and, at the moment, could only be the perform primitive. If the source is perform, then the *MethodSource* is constructed with method selector and class set to the source of the primitive, conveniently recorded by the primitive's *mir* entry. In fact, if this invocation of perform is itself the result of a perform, for example (taken from Smalltalk-80):

```
42 perform: #perform: with: #printString
```

then the performed printString would inherit its source from the performed perform, which would in turn inherit it from the original perform message, and so the effect is propagated as necessary. It should be noted that the introduction of a dynamic message constructor does not have an effect on the termination properties of the recursion control mechanism, because although there will be an increase in the number of message sends beyond the number initially present in the code, the number of different sources recorded will remain the same.

$$
\begin{array}{l}
\mathit{extract\_source} : (\mathit{Id} \cup \{\mathsf{nil}\}) \times \mathit{MethodInvocationRecord} \rightarrow (\mathit{MessageSource} \cup \{\mathsf{nil}\}) \\
\mathit{extract\_source}(\mathit{marker}, \mathit{mir}) \quad \triangleq \\
\quad \mathsf{if}\ \mathit{mir} = [\,] \\
\quad \mathsf{then\ nil} \\
\quad \mathsf{else\ let}\ \mathit{last\_recurs} = \mathit{mir}[\mathsf{len}\,\mathit{mir}] \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{let}\ \mathit{last\_sel} = \mathit{Sel}(\mathit{last\_recurs}) \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{let}\ \mathit{last\_class} = \mathit{Class}(\mathsf{hd}\,\mathit{Args}(\mathit{last\_recurs})) \\
\qquad\quad \mathsf{in} \\
\qquad \mathsf{if}\ \mathit{Methods}(\mathit{program}(\mathit{last\_class}))(\mathit{last\_sel}) \in \mathit{UserMethod} \\
\qquad \mathsf{then}\ \mathit{mk\text{-}MethodSource}(\mathit{marker}, \mathit{last\_sel}, \mathit{last\_class}) \\
\qquad \mathsf{else\ let}\ \mathit{last\_source} = \mathit{Source}(\mathit{last\_recurs}) \\
\qquad\qquad \mathsf{in} \\
\qquad\quad \mathit{mk\text{-}MethodSource}(\mathit{marker}, \mathit{Sel}(\mathit{last\_source}), \mathit{Class}(\mathit{last\_source}))
\end{array}
$$

5.3.2 Finding the class, and creating new objects

Class, the primitive which reports on the class of the receiver, is handled by *find_class*. The problem of determining the class of an object is eased somewhat by the fact that, through receiving this message, the object under interrogation must now be class-derived. The solution, therefore, is simply to extract the class from the object description held for the object identity provided, *oop*, by the *OopMap* of the object space. After extracting the name, however, the problem then becomes, how to convert it into an object which can be manipulated by the program. This latter task is performed by *class_as_oop*, whose definition is dependent on how classes are handled by the implementation — for example, as 'static' predefined objects (page 93), or, as class names represented by character strings. For this reason, the definition of *class_as_oop* has been omitted.

$$
\begin{array}{l}
\mathit{find\_class} : \mathit{Oop} \times \mathit{DfEnv} \rightarrow \mathit{DfSet} \\
\mathit{find\_class}(\mathit{oop}, \delta) \quad \triangleq \\
\quad \mathsf{let}\ \mathit{object} = \mathit{OopMap}(\sigma(\delta))(\mathit{oop}) \\
\qquad \mathsf{in} \\
\quad \mathsf{let}\ (\mathit{class\_oop}, \sigma') = \mathit{class\_as\_oop}(\mathit{Class}(\mathit{object}), \sigma(\delta)) \\
\qquad \mathsf{in} \\
\quad \mathsf{let}\ \delta' = \mu(\delta, \sigma \mapsto \sigma') \\
\qquad \mathsf{in} \\
\quad \{\mu(\delta', \mathit{last} \mapsto \mathit{class\_oop})\}
\end{array}
$$

When creating a new object, *class_as_oop* must alter the object space, and so it is necessary for this function to return an updated copy of the object space.

$$\mathit{class\_as\_oop} : \mathit{ClassName} \times \mathit{ObjectSpace} \rightarrow \mathit{Oop} \times \mathit{ObjectSpace}$$

Extending the idea that syntactic class names can be converted into objects to be manipulated, the object creation expression could be complemented by a primitive method which works with such class-name objects. Such a primitive is the new primitive, seen here added to the list of primitives recognised by *Invoke_Primitive*, which is handled by *create_object*.

$$\mathit{Invoke\_Primitive} : \mathit{PrimitiveId} \times \mathit{Oop} \times \mathit{Oop}^{*} \times \mathit{DfEnv} \rightarrow \mathit{DfSet}$$

$$\mathit{Invoke\_Primitive}(\mathit{primitive\_id}, \mathit{rcvr\_oop}, \mathit{arg\_oops}, \delta) \quad \triangleq$$

$$\begin{array}{l} \text{cases } \mathit{primitive\_id} \text{ of} \\ \quad \ldots \\ \mathit{new} \rightarrow \mathit{create\_object}(\mathit{rcvr\_oop}, \delta) \\ \quad \ldots \\ \text{end} \end{array}$$

Create_object performs its task by doing the inverse of *find_class* in order to extract a syntactic class name. It uses this syntactic name to construct a *NewObject* expression which can then be handled by *DfExpression*. However before the new class name can be applied to create a new object, it must be matched against one of the classes used by the program. This is done by checking that *class*, extracted by *oop_as_class*, the inverse function of *class_as_oop*, is in the domain of the program. If the proposed class cannot be found in the set defined for the program, a primitive failure occurs.

$$\mathit{create\_object} : \mathit{Oop} \times \mathit{DfEnv} \rightarrow \mathit{DfSet}$$

$$\mathit{create\_object}(\mathit{oop}, \delta) \quad \triangleq$$

$$\begin{array}{l} \text{let } \mathit{class} = \mathit{oop\_as\_class}(\mathit{oop}, \sigma(\delta)) \\ \quad \text{in} \\ \text{if } \mathit{class} \in \text{ dom } \mathit{program} \\ \text{then } \mathit{DfExpression} [\![\mathit{mk\text{-}NewObject}(\mathit{class})]\!] \, \delta \\ \text{else } \mathit{primitive\_failure}(\text{‘unknown class’}, \mathit{new}, \mathit{oop}, [\,], \delta) \end{array}$$

As with *class_as_oop*, the definition of *oop_as_class* is implementation dependent, and has therefore been omitted.

$$\mathit{oop\_as\_class} : \mathit{Oop} \times \mathit{ObjectSpace} \rightarrow \mathit{ClassName}$$

However, the inclusion of the new primitive, with its corresponding primitive handling function would cause the definition and meaning of the analysis mechanism to become unsound, simply because of the ‘dynamic’ synthesis of a new expression to be analysed [Yel87]. As such, the new expression has *not* been included in the main body of primitives handled by the analysis mechanism, despite the fact that an initial observation of the definition reveals that the inclusion of the primitive can have no overall effect on either the termination or otherwise of the analysis mechanism, or the quality of information derived by the system.

5.3.3 Become

The become primitive poses something of a problem. Apart from the fundamental semantic problems laid out in [Wol88] (§3.9.3), there is a problem in how to define the analysis function to cope with it. In principle, become swaps two objects so that whatever was referring to one object beforehand, now references the other, and vice versa. There are two possible ways in which the action of become may be expressed within the object space: one which deals with the link between references (variables) and object identities, and one which deals with the link between object identities and object descriptions. The technique chosen is the latter of the two, preserving the link between variables and object identities, while altering the link between identities and object descriptions.

The chosen technique implements the become action by swapping the object bodies of the two objects involved. This involves extracting the *OopMap* from the object space and altering the two maplets involving the receiving object's identity, *rcvr_oop*, and the argument object's identity, *arg_oop*, so that they now map to each others' original entry. The final result, the object now being referenced by what was the receiving expression, is provided by *rcvr_oop* because it is the object bodies which have changed, and not their identities.

$$\mathit{become\_args} : \mathit{Oop} \times \mathit{Oop}^* \times \mathit{DfEnv} \to \mathit{DfSet}$$

$$\mathit{become\_args}(\mathit{rcvr\_oop}, [\mathit{arg\_oop}], \delta) \quad \triangleq$$

$$\begin{array}{l}
\mathsf{let}\ \mathit{oop_map} = \mathit{OopMap}(\sigma(\delta)) \\
\quad \mathsf{in} \\
\mathsf{let}\ \mathit{rcvr_oop_maplet} = \{\mathit{rcvr_oop} \mapsto \mathit{oop_map}(\mathit{arg_oop})\} \\
\quad \mathsf{in} \\
\mathsf{let}\ \mathit{arg_oop_maplet} = \{\mathit{arg_oop} \mapsto \mathit{oop_map}(\mathit{rcvr_oop})\} \\
\quad \mathsf{in} \\
\mathsf{let}\ \mathit{oop_map}' = \mathit{oop_map} \dagger \mathit{rcvr_oop_maplet} \dagger \mathit{arg_oop_maplet} \\
\quad \mathsf{in} \\
\mathsf{let}\ \delta' = \mu(\delta, \sigma \mapsto \mu(\sigma, \mathit{OopMap} \mapsto \mathit{oop_map}')) \\
\quad \mathsf{in} \\
\{\mu(\delta, \mathit{last} \mapsto \mathit{rcvr_oop})\}
\end{array}$$

The choice of technique has been made to simplify the handling of unresolved virtual objects. If, instead, the variable to identity mapping was altered, the set of 'non-equal' identities recorded by each unresolved virtual object would have had to have been updated to keep it consistent with the assumptions made in its construction. For example consider the following mappings from variables to object identities to objects:

variable		identity		object
varVirtual	$\mapsto$	$voop$	$\mapsto$	$mkVirtualObject(cdos, \{oop_1\})$
var1	$\mapsto$	oop_1	$\mapsto$	$mkClassDerivedObject(class_1, \ldots)$
var2	$\mapsto$	oop_2	$\mapsto$	$mkClassDerivedObject(class_2, \ldots)$

In this example the objects referenced by varVirtual and var1 have been tested for equivalence (see §5.3.4, page 147) and, for this case, have been assumed unequal. Now consider the following application of the become primitive:

```
var1 become: var2
```

Under the chosen technique, as implemented by *become_args* above, the effect of this would be to swap the two object bodies:

$$\begin{array}{lllll} \text{var1} & \mapsto & oop_1 & \mapsto & mkClassDerivedObject(class_2, \ldots) \\ \text{var2} & \mapsto & oop_2 & \mapsto & mkClassDerivedObject(class_1, \ldots) \end{array}$$

Under the alternative technique of swapping the variable to identity mappings the outcome of this application would be:

$$\begin{array}{lllll} \text{var1} & \mapsto & oop_2 & \mapsto & mkClassDerivedObject(class_2, \ldots) \\ \text{var2} & \mapsto & oop_1 & \mapsto & mkClassDerivedObject(class_1, \ldots) \end{array}$$

However the information recorded for varVirtual and its unresolved virtual object would still indicate that the object's identity could not be oop_1, despite the fact that all variables referencing oop_1, the object varVirtual was not equal to, are now referencing oop_2. In this sense, the previous test for equivalence between the virtual and non-virtual objects was really a test to see if the two variables were referencing the same object. If varVirtual had been referencing the same object as these variables, it too would now be referencing oop_2. It is clear that by using the alternative technique it would be necessary to propagate the swapping of mapped object identities, in this case replacing oop_1 by oop_2, through to the set of 'non-equal' object identities recorded by any unresolved virtual objects. This would have introduced an additional complexity, avoided when using the comparatively simple technique chosen.

5.3.4 Testing for equivalence

The final primitive to be considered in this section is the primitive which handles the test for object equivalence, ==, used in the illustrative examples given above. The test for equivalence, based on a test of object identity equality, is one of the key primitives to be handled by this analysis system, and lies at the heart of the analysis mechanism. The ability to differentiate and equate objects is a necessary step towards the construction and analysis of the complex control structures which will arrive with the introduction of blocks in the next chapter.

The comparison of objects has to be done on a case by case basis, deriving each case from a different combination of object identity forms. Each main version covers a different identity form for the receiver object: *predefined*, *virtual* and *concrete*. Within each function, case analysis is applied to the single argument object. With three forms of object identity, there are a total of nine combinations to be considered.

The first main case covers receiving objects which have predefined object identities. That is, objects which are assumed to exist at the start of the analysis process. The choice of handling functions at the disposal of all three versions of the main function are: *t_f_equality* for simple comparisons; *t_f_equality_voop* for comparisons involving unresolved virtual objects; *return_false* for comparisons which are always false. Taking each kind of argument object identity in turn, predefined objects are subjected to a simple comparison based on each objects' identity. Virtual objects, which may or may not be resolved, must be subjected to a more complex test, using *t_f_equality_voop*, in order to distinguish between potentially equal *unresolved* virtuals and unequal resolved virtuals. Finally, for objects with concrete identities, which have been 'created' during the analysis process, the comparison must necessarily return false. Note that for the last case, because *return_false* returns a single dataflow environment, this result must be placed into a set in order to comply with the main function's signature.

$$\mathit{test\_for\_equality} : \mathit{Oop} \times \mathit{Oop}^* \times \mathit{DfEnv} \to \mathit{DfSet}$$

$$\mathit{test\_for\_equality}(\mathit{rcvr\_oop}, [\mathit{arg\_oop}], \delta) \quad \triangleq$$

$$\begin{array}{l} \text{cases } \mathit{arg\_oop} \text{ of} \\ \quad \mathit{mk\text{-}PredefinedOop}(\mathit{arg\_obj}) \to \mathit{t\_f\_equality}(\mathit{rcvr\_oop}, \mathit{arg\_oop}, \delta) \\ \quad \mathit{mk\text{-}VirtualOop}(\mathit{arg\_obj}) \to \mathit{t\_f\_equality\_voop}(\mathit{rcvr\_oop}, \mathit{arg\_oop}, \delta) \\ \quad \mathit{mk\text{-}ConcreteOop}(\mathit{arg\_obj}) \to \{\mathit{return\_false}(\delta)\} \\ \text{end} \end{array}$$

where

$$\mathit{rcvr\_oop} \in \mathit{PredefinedOop}$$

For an object with a virtual identity to be the receiver of a message, it must be fully resolved. That is, it must be in possession of a unique identity and a class-derived object body. If it was not fully resolved before it received the message, it will have assumed one during the analysis of the receiving object. This fact limits the possibilities with respect to equality with other object identities. Both predefined and concrete object identities are non-starters because they are of the wrong form. As for other virtual objects, resolved or otherwise, the problem is delegated to *t_f_equality_voop*.

$$\mathit{test\_for\_equality}(\mathit{rcvr\_oop}, [\mathit{arg\_oop}], \delta) \quad \triangleq$$

$$\begin{array}{l} \text{cases } \mathit{arg\_oop} \text{ of} \\ \quad \mathit{mk\text{-}PredefinedOop}(\mathit{arg\_obj}) \to \{\mathit{return\_false}(\delta)\} \\ \quad \mathit{mk\text{-}VirtualOop}(\mathit{arg\_obj}) \to \mathit{t\_f\_equality\_voop}(\mathit{rcvr\_oop}, \mathit{arg\_oop}, \delta) \\ \quad \mathit{mk\text{-}ConcreteOop}(\mathit{arg\_obj}) \to \{\mathit{return\_false}(\delta)\} \\ \text{end} \end{array}$$

where

$rcvr\_oop \in VirtualOop$

The case for a concrete, class-derived receiver object is the simplest of the three. There is no question of equality between it and any predefined or virtual object identities simply because any such objects will have been effectively created before the receiver, and so under no circumstances could the two be the same object. This leaves the case of two concrete object identities which is handled by the simple identity comparison function, *t_f_equality*.

$test\_for\_equality(rcvr\_oop, [arg\_oop], \delta) \quad \triangleq$

cases arg_oop of

$mk\text{-}PredefinedOop(arg\_obj) \rightarrow \{return\_false(\delta)\}$

$mk\text{-}VirtualOop(arg\_obj) \rightarrow \{return\_false(\delta)\}$

$mk\text{-}ConcreteOop(arg\_obj) \rightarrow t\_f\_equality(rcvr\_oop, arg\_oop, \delta)$

end

where

$rcvr\_oop \in ConcreteOop$

Before the real comparison functions are presented, here are the two functions which handle the return of the two possible outcomes of the comparison primitive, namely *true* and *false*. The functions assume the existence of the two predefined object identities, true and false, which are assumed to be the identities of instances of classes True and False respectively (page 92).

$return\_false : DfEnv \rightarrow DfEnv$

$return\_false(\delta) \quad \triangleq \quad \mu(\delta, last \mapsto false)$

Although only *return_false* has been required so far, *return_true* will be needed shortly.

$return\_true : DfEnv \rightarrow DfEnv$

$return\_true(\delta) \quad \triangleq \quad \mu(\delta, last \mapsto true)$

The simple cases mentioned above, those involving a straight comparison of object identities of the same form, are handled by *t_f_equality*. It tests to see if the identities are the same and then calls *return_true* or *return_false* depending on the outcome, so converting a logical true or false into a true or false object. As with *test_for_equality*, the results of these two functions need to be placed in a set in order to comply with the function's signature.

$$t\_f\_equality : Oop \times Oop \times DfEnv \rightarrow DfSet$$
$$t\_f\_equality(rcvr\_oop, arg\_oop, \delta) \quad \triangleq$$
$$\quad \text{if } rcvr\_oop = arg\_oop$$
$$\quad \text{then } \{return\_true(\delta)\}$$
$$\quad \text{else } \{return\_false(\delta)\}$$

T_f_equality_voop has to handle cases from two different sources. When called from the predefined receiving object case, the function needs to deal specifically with the case of an unresolved virtual object. For this situation, if the virtual object is fully resolved with a class-derived body, the test is handed over to *t_f_equality* in the firm knowledge that, because of the mismatch between object identities, the result will always be false. The interesting case, that of an unresolved virtual object, is handed over to *equal_v_obj_cases*, which is able to produce different cases based on assuming the equality or otherwise of the two object identities.

In contrast, the other source, that of two virtual objects, is interested in both cases, resolved and unresolved. If the argument object is a resolved virtual object, with a unique identity, then the test can be handed over to the simple comparison function *t_f_equality*. For the unresolved virtual object case, the problem is once again handed over to *equal_v_obj_cases* for exactly the same reasons as before, namely to produce cases which cover all possible assumptions.

$$t\_f\_equality\_voop : Oop \times VirtualOop \times DfEnv \rightarrow DfSet$$
$$t\_f\_equality\_voop(rcvr\_oop, arg\_voop, \delta) \quad \triangleq$$
$$\quad \text{let } arg\_object = OopMap(\sigma(\delta))(arg\_voop)$$
$$\quad\quad \text{in}$$
$$\quad \text{cases } arg\_object \text{ of}$$
$$\quad mk\text{-}ClassDerivedObject(c, vs) \rightarrow t\_f\_equality(rcvr\_oop, arg\_voop, \delta)$$
$$\quad mk\text{-}VirtualObject(cdos, neqs) \rightarrow equal\_v\_obj\_cases(rcvr\_oop, arg\_voop, \delta)$$
$$\quad \text{end}$$

There are two cases to be considered when comparing any object with an unresolved virtual object: equal and not equal. These two cases are handled by *equal_v_obj_true* and *equal_v_obj_false* respectively. The result of *equal_v_obj_cases* is then the union of these two cases.

$$equal\_v\_obj\_cases : Oop \times VirtualOop \times DfEnv \rightarrow DfSet$$
$$equal\_v\_obj\_cases(rcvr\_oop, arg\_voop, \delta) \quad \triangleq$$
$$\quad \text{let } true\_case = equal\_v\_obj\_true(rcvr\_oop, arg\_voop, \delta)$$
$$\quad\quad \text{in}$$
$$\quad \text{let } false\_case = equal\_v\_obj\_false(rcvr\_oop, arg\_voop, \delta)$$
$$\quad\quad \text{in}$$
$$\quad true\_case \cup false\_case$$

Before creating the 'assumed equal' case, it is necessary to make sure that these two objects have not been compared before. If they have, then for the current test to be taking place, the current control path being followed must descend from the 'assumed unequal' case of the original comparison. This is because if the 'assumed equal' path had been followed, the argument object, being an unresolved virtual object, would have been equated to and overwritten by the receiver object. For the argument object to be still in existence in its original form, this equating of objects cannot have occurred.

It is here that the extra field, *Neq-set*, introduced into the virtual object body, becomes significant. From the above reasoning, it is necessary for an unresolved virtual object to record those objects with which it could be usefully compared,[6] but to which it has been assumed unequal. Therefore every time a comparison is made between a predefined or fully-resolved virtual object and an unresolved virtual object, the latter object must make a note of the receiver's identity for each 'assumed unequal' case created. This is one of the tasks carried out by *equal_v_obj_false*. Meanwhile, for the assumed equal case the receiver object must be tested against the set of unequal object identities recorded by the argument object. If the receiver is in the set, the comparison must fail, and an empty set of cases is returned. If the comparison is a new one, then the receiver will not be in the argument's *Neq-set*, and the case construction can continue. For this, it is necessary to equate the unresolved virtual object with the receiver, removing all references to it and replacing them with references to the receiver object. There is already a function that performs this task, namely *equate_oops* (page 117), which is part of the resolution process required for unresolved virtual objects when they receive their first message. With the objects equated, all that needs to be done is to call *return_true* to round off the case construction.

$$\begin{array}{l}
\mathit{equal_v_obj_true} : \mathit{Oop} \times \mathit{VirtualOop} \times \mathit{DfEnv} \rightarrow \mathit{DfSet} \\
\mathit{equal_v_obj_true}(\mathit{rcvr_oop, arg_voop, arg_v_obj}, \delta) \quad \triangleq \\
\qquad \text{let } \mathit{arg_v_obj} = \mathit{OopMap}(\sigma(\delta))(\mathit{arg_voop}) \\
\qquad\quad \text{in} \\
\qquad \text{if } \mathit{rcvr_oop} \in \mathit{Neq\text{-}set}(\mathit{arg_v_obj}) \\
\qquad \text{then } \{\,\} \\
\qquad \text{else let } \delta' = \mu(\delta, \sigma \mapsto \mathit{equate_oops}(\mathit{rcvr_oop, arg_voop}, \sigma(\delta))) \\
\qquad\qquad\quad \text{in} \\
\qquad\qquad \{\mathit{return_true}(\delta')\}
\end{array}$$

The 'assumed unequal' case merely requires the recording of the receiving object's identity in the set of 'assumed unequal' object identities kept by the unresolved virtual object. This is done by first extracting the virtual object body from the object space, updating its *Neq-set* and then reconstructing the object space and dataflow environment. In keeping with the 'assumed equal' case, the function is finished off with a call of *return_false*.

[6]i.e. Any object except those objects whose structure would mean that any comparison would always return false, for example concrete, class-derived objects.

$$
\begin{array}{l}
\mathit{equal\_v\_obj\_false} : \mathit{Oop} \times \mathit{VirtualOop} \times \mathit{DfEnv} \rightarrow \mathit{DfSet} \\
\mathit{equal\_v\_obj\_false}(\mathit{rcvr\_oop}, \mathit{arg\_voop}, \mathit{arg\_v\_obj}, \delta) \quad \triangleq \\
\quad \text{let } \sigma' = \sigma(\delta) \\
\quad\quad \text{in} \\
\quad \text{let } \mathit{arg\_v\_obj} = \mathit{OopMap}(\sigma')(\mathit{arg\_voop}) \\
\quad\quad \text{in} \\
\quad \text{let } \mathit{arg\_v\_obj}' = \mu(\mathit{arg\_v\_obj}, \mathit{Neq\text{-}set} \mapsto \mathit{Neq\text{-}set}(\mathit{arg\_v\_obj}) \cup \{\mathit{rcvr\_oop}\}) \\
\quad\quad \text{in} \\
\quad \text{let } \sigma'' = \mu(\sigma', \mathit{OopMap} \mapsto \mathit{OopMap}(\sigma') \dagger \{\mathit{arg\_voop} \mapsto \mathit{arg\_v\_obj}'\}) \\
\quad\quad \text{in} \\
\quad \text{let } \delta' = \mu(\delta, \sigma \mapsto \sigma'') \\
\quad\quad \text{in} \\
\quad \{\mathit{return\_false}(\delta')\}
\end{array}
$$

5.4 Summary

This chapter has shown how the simple dataflow analysis system introduced in the previous chapter can be extended to handle primitive methods. By and large the intergration has been smooth, with the primitive handling functions hanging off a call to *DfPrimitive*, which provides a suitable interface between the original system and the primitive extensions. However for the equivalence (==) primitive, the integration has not been so smooth. For this primitive it has been necessary to extend the basic object model to provide proper handling of equality, specifically for unresolved virtual objects. This should be borne in mind when the issue of adding integers is discussed in §6.8.

In the next chapter, the analysis system will be extended even further to handle the analysis of blocks. The next chapter will also round off the description of the dataflow analysis system.

Chapter 6

Adding blocks

Blocks in Smalltalk-80 come in two basic varieties, local return blocks and non-local return blocks. Both kinds of blocks are essentially anonymous methods. They consist of a body, which constitutes the code to be executed, together with a list of parameters, similar to the parameters associated with a method, and a set of temporary variables. Where they differ from each other is in what happens after they have completed their execution.

A block is an object representing a piece of code. Because a block exists as an object, it is possible to both pass it around as a message argument and send it messages. In fact the execution of a block body is achieved by invoking the value primitive via a message send to that block. Invoking a block through the value message is conceptually similar to invoking a method at that point in the code, where the method invoked is in fact the block. With regard to a block's action after it has completed its execution, for a local-return block control is returned to the point where the block's invoking message was sent, in other words, to the point of the block's invocation.

Non-local return blocks are invoked in the same way as local-return blocks. Their difference lies in what happens when the block completes its execution. For both a local-return and non-local return block, the environment of its execution is fixed at its point of creation. That is, the visibility of any variables and parameters is determined by the place where the block is defined. For the non-local return block, a return path is also bound, linking the block to the point of its creation. It is this return path which determines to where the block returns control to after it has successfully completed its execution. For a non-local return block control is passed back, not to the point of invocation, not to the point of the block's creation, but back to the invocation point of the method which originally contained the block's definition.

For example, consider the following, rather contrived example. Method cee contains a message send invoking another method, dee

```
cee: aKindOfArg
    "Method, sending message dee to object referenced by aKindOfArg."
    ...
    x := aKindOfArg dee.
    ...
```

Within method dee there is a non-local return block which is sent the message eee,

```
dee
    "Method containing a block which is eventually executed by eee."
    ...
    [self miscMsg. ↑self] eee
    "This point never reached."
    ...
```

If the relevant implementation of eee invokes the non-local return block, control will return immediately to the original method cee, to the point where the message send invoking the method which defined the block, namely dee, occurred.

```
eee
    "Method invoking the receiver, a block."
    self value
```

Both kinds of blocks are primarily used for control-flow structures, for example the Smalltalk equivalent of the `if-then-else` construct, ifTrue:ifFalse:. For example, consider the following expression:

```
anObject test ifTrue: [ anObject testMessage ]
```

The implementation for ifTrue: for objects of class True is:

```
ifTrue: aTrueBlock
    "True case, so invoke block."
    ↑aTrueBlock value
```

while the implementation in class False is:

```
ifTrue: aTrueBlock
    "False case, so do nothing."
    ↑nil
```

As a result, the testMessage message is only sent if the test message returns a True object. The following example shows how a non-local return block could be used in a control structure.

```
characterOrNumber
    "Tests the state of the instance variable ownInstVar, and returns the
     number 42 if it is an integer."
    ownInstVar isInteger
                ifTrue: [ ↑42 ].
    self ditherAbout characterOrNumber
```

In this example the block represents an escape route from the recursion introduced by the last line. Using a local-return block as the argument to ifTrue: would simply return the object 42 and then carry on with the recursive message send. By using a non-local return block, after returning the object 42, control is passed out of the method, back to the place where the method was invoked.

6.1 Determining control-flow through blocks

Because blocks are essentially anonymous methods, both the local and non-local return forms could, in principle, be dealt with by the analysis mechanism, once the context of their invocation has been determined. The problem would seem to lie, therefore, not with the block itself, but in determining the context surrounding its invocation. The simple sending of a value message, leading to the value primitive, provides no obstacles. The problem instead lies with more indirect means of invoking a block — in other words control structures: messages involving blocks with some form of conditional execution.

Ultimately the question of whether a given block will be invoked is undecidable. In a 'real world' context it is complicated by several unknowns, for example user input, or the initial state of the system. However the aim of the current work on control-flow analysis is not to predict whether a given block will be invoked, but to attempt to determine the conditions under which that block will be invoked. This is a really a problem of modelling the conditions affecting the invocation of a block, using the information recorded by the object space of the dataflow analysis system. By refining the model of the object space it is hoped that useful information may be derived about the conditional invocation of blocks.

When a control structure is encountered, some knowledge of the current state can be determined from the object space. The control structure will be in the form of a message send, so the current mechanism for message passing analysis can be used. Already the system can handle discrimination at the level of classes, for example the message isNil will return true or false depending on the class of the receiver. This result can then be used as the receiver of subsequent messages, for example ifTrue:ifFalse:. The invocation of the argument blocks (one for true, one for false) is then conditional on whether the receiver, the result of the isNil message, evaluated to true (class True) or false (class False), and so which version of the ifTrue:ifFalse: method is invoked. This form of discrimination, which is a general technique not limited to true and false, can already be handled by the type inference system described in Chapter 3. Our interests lie in a much finer level of discrimination, namely discriminating between individual objects.

All control structures are built from the same basic construct, the conditional execution. Such conditional executions can be broken down into three sections: the initial test (for example the isNil message); the evaluation of the result of the test (for example sending the message ifTrue:ifFalse: to the result of the initial test); and finally, the execution or otherwise of the conditional block as a result of the evaluation (for example sending the value message to the true block in the class True implementation of ifTrue:ifFalse:).

$$\text{test} \rightarrow \text{evaluation of result} \rightarrow \text{execution of block}$$

Note that the block can only be invoked by the value message, and so ultimately, when the block is invoked, it will be invoked unconditionally (as in the implementation of ifTrue: for class True, above). In effect, therefore, the evaluation of the test result actually occurs within the act of sending the message, with the invoked method implementing the decision.

In Smalltalk, all conditionals are ultimately resolved by a method lookup, with the class of the result of the test determining whether the block is invoked. As has already been mentioned, this technique is not limited to the boolean classes True and False, but can be extended to any logic system the programmer may wish to implement.[1] In effect, this means that it is the test itself that presents a stumbling block to more refined control-flow analysis.

Further investigation into the forms that the test can take seems to reveal two basic kinds, with more complex 'test' methods ultimately being reducible to one or other of the following:

1. A simple unary message returning a value which 'plugs' straight into the result evaluation logic system. For example isNil which returns an instance of class True when sent to an UndefinedObject object, and an instance of class False when sent to any other class of object.

2. Some form of object comparison, for example testing for object equivalence (==), which must be implemented as a primitive.

With case 1 comparatively straight forward, it would seem that if we can handle case 2, and provide a suitable context for each possible result, then we will be able to analyse control-flow to a much finer level of discrimination than that made possible by a type system.

6.2 Determining dataflow through test expressions

Now that we have narrowed down the problem of determining the control-flow through a block, we must consider the new problem of determining the dataflow through a test expression. Of the two forms of test expression just described, the first can be dealt with using the basic analysis techniques introduced in §4.4. That leaves the second form, object comparison. User-defined methods do not present a problem for the analysis mechanism, and so it is not just any kind of object comparison that we are interested in, but instead fundamental comparisons which cannot be implemented other than by primitive methods. The question we must now ask is, what form can such a fundamental object comparison take?

Although a 'proper' language implementation would include 'essential' classes such as Integer which have an explicit ordering between instances, such classes have initially been omitted in order to maintain simplicity. Such classes would have added fundamental ordering comparisons, for example <, 'less than' for integers, but despite their omission here, some of the implications of adding such classes will be addressed in §6.8. However, without these classes there is only one fundamental form of comparison and that is object equivalence, or the == primitive. In effect, this means that in order to handle any 'test', all we have to do is

[1]Unfortunately, the ParcPlace Smalltalk-80 compiler always treats the message selectors ifTrue:ifFalse: and its variants as special cases, and in-lines them as traditional boolean conditional expressions. This effectively prevents the resident boolean classes from being extended to create more esoteric logic systems using the same message selectors. However, such a new logic system could be implemented using a separate set of selectors.

handle object equivalence. This is because any message send, other than a simple interrogative message, must eventually bottom-out with some primitive action (object creation, variable assignment, object comparison etc.), and the only primitive object comparison in our system is the test for object equivalence. Any other form of test expression would, by necessity, ultimately be of the first, unary message kind, and not of the second, more interesting, object comparison kind.

To recap on our object space model, we have three basic kinds of objects:

1. *predefined object* — objects of known identity, assumed to have been 'created' before the start of the code analysis.

2. *virtual object* — objects with unknown form and identity (unresolved), or with acquired form and fixed identity (resolved), assumed to have been 'created' at the start of the code analysis.

3. *class-derived object* — objects 'created' during the execution of the code, with a known, fixed identity.

As has already been covered by the == comparison primitive (page 147), in a test between two objects, the test is simple for those cases where the objects are both of kind 1 or 3. The test is even simpler for comparisons between objects of kinds 2 and 3, because the result is always false. That leaves comparisons between kinds 1 and 2, and between two objects of kind 2, for which at most two cases are created, *equal* and *not equal*. For the latter case, a note is made of the objects not equal to each other.

In this way all possible outcomes from an equivalence test can be considered, with each resulting local context being propagated to determine the overall context of the resulting control-flow. By performing this kind of dataflow on basic object comparisons, a refinement can be achieved in determining the control-flow through a control structure. Further work on refining the object space model to handle objects with an explicit ordering can extend this system to handle even more complex comparisons, in the same way that the extension to the virtual object body in §5.1 enabled the equivalence primitive to be handled properly.

6.3 Additions to the syntax

In order to keep the whole analysis mechanism simple, only local-return blocks will be dealt with. Amongst other things, non-local return blocks would necessitate the introduction of tests for each use of *DfExpression* in order to handle the potential change in control flow resulting from the execution of such blocks. Local-return blocks on the other hand can be accommodated without any changes to the basic analysis mechanism.

The only addition to the abstract syntax required for the introduction of blocks is the block literal.

$$Expression = \ldots \cup BlockLiteral$$

The block literal has the same format as a user-defined method, with a sequence of parameters, a set of temporary variables and an expression to be executed when the block is invoked. In addition, the block also has a field to record the method in which it is defined. This is similar to the 'marker' which is recorded for each message send expression, but goes one step further by assigning not only a 'marker' but also the class and method selector, in the form of a *MethodSource* (page 126), to locate the block's defining point precisely.[2] This 'block-origin' field is needed for the purposes of recursion control, both as a reference point for message expressions which occur within the block, and as a means of distinguishing between different blocks without distinguishing between blocks originating from the block literal expression.[3] Its use will be described later, in §6.7.

$$\begin{array}{lrcl} \mathit{BlockLiteral} :: & \mathit{Params} & : & \mathit{ParamId}^* \\ & \mathit{Temps} & : & \mathit{TempVar}\text{-}\mathsf{set} \\ & \mathit{Body} & : & \mathit{Expression} \\ & \mathit{Block\_origin} & : & \mathit{MethodSource} \end{array}$$

where

$$\mathit{inv\text{-}BlockLiteral}(\mathit{mk\text{-}BlockLiteral}(\mathit{params}, \mathit{temps}, \mathit{body}, \mathit{origin})) \quad \triangleq$$
$$\quad \mathsf{card\ elems}\, \mathit{params} = \mathsf{len}\, \mathit{params}$$

As with the user-defined method, there is an invariant on the block literal which ensures that there are no duplicates within the sequence of parameters.

6.4 Alterations to the Object Space

With the introduction of block literals, some changes have to be made to the object space to handle the new kind of object. First though, for the sake of clarity, *BlockOop* is defined as an alias of *ConcreteOop*, signifying that blocks are 'created' during the analysis process.

$$\mathit{BlockOop} = \mathit{ConcreteOop}$$

The block literal is catered for in the object space by providing a new object description. In addition to the original virtual and class-derived objects, there is now a block object: *BlockObject*.

$$\mathit{Object} = \mathit{VirtualObject} \cup \mathit{ClassDerivedObject} \cup \mathit{BlockObject}$$

The block object records its defining block literal directly, and in so doing records the same information as a user-defined method definition. In addition, however, it is necessary to bind the block to the right context in order to access the correct local variables and parameters

[2]The need to record the full source of a block arises because, unlike a message expression which remains within the method in which it occurs, a block, being an object, may be passed around as an argument, and hence it is necessary to provide some way of linking it with its defining method, as well as to its position within that method.

[3]This distinction is not possible through the blocks' identities, because blocks created from the same block literal expression during separate invocations of the same method will naturally have different object identities. However, as was shown on page 131, recursion control based on object identities does not work.

bound within the block's defining method. Assuming that occurrences of the parameters and variables defined for the enclosing method are tagged within the block at the same time as they are tagged for the rest of the method (page 104), the only main problem lies with the self object and its instance variables. By routing instance variable accesses through the self object (*DfExpression* for *ObjectNames*, page 108), only the self object needs to be recorded. This is handled by the *ContextSelf* field.

$$\begin{aligned} \mathit{BlockObject} :: \quad \mathit{Block} &: \mathit{BlockLiteral} \\ \mathit{ContextSelf} &: \mathit{Oop} \end{aligned}$$

6.5 Alterations to the flow analysis

The initialisation process, from the selection of a class (*DfProgram*, §4.4.1) through to the initialisation of an initial environment for that method (*DfMethod*, §4.4.2) remains unchanged. The first real change occurs for the ubiquitous expression analysis function, *DfExpression*.

$$\mathit{DfExpression} : \mathit{Expression} \rightarrow \mathit{DfEnv} \rightarrow \mathit{DfSet}$$

In fact the only alterations required for *DfExpression* are an additional constraint on the new-object expression, and the introduction of the block-literal expression. The definition of the new-object case (*NewObject*) is exactly the same as when it was first defined on page 109, with the addition of a constraint ensuring that it is never applied to class BlockClosure. This constraint guarantees that blocks are only created from block literals.

The *DfExpression* case for a block literal follows the same pattern as the new-object case. A specialist function, *prepare_new_block*, is called which takes the block-literal, creates a block object from it, inserts it into the object space, and returns the new block's object identity together with an updated object space. The object space is then inserted into the dataflow environment, which is returned once the *last* field has been set to the new object's identity.

$$\begin{aligned}
&\mathit{DfExpression}\,[\![\mathit{block_literal}]\!]\,\delta \quad \triangleq \qquad\qquad \text{for } \mathit{block_literal} \in \mathit{mk\text{-}BlockLiteral} \\
&\quad \text{let } (\mathit{block_oop}, \sigma') = \mathit{prepare_new_block}(\mathit{block_literal}, \sigma(\delta)) \\
&\qquad \text{in} \\
&\quad \text{let } \delta' = \mu(\delta, \sigma \mapsto \sigma') \\
&\qquad \text{in} \\
&\quad \{\mu(\delta', \mathit{last} \mapsto \mathit{block_oop})\}
\end{aligned}$$

The operation of *prepare_new_block* mirrors the actions of two other operations. It starts by requesting a new object identity, supplied by *new_block_object_id* in return for the object space. This action is similar to those functions which handle the creation of new objects, for example *prepare_new_object*. This initial operation is followed by a call to *tag_block_variables* which returns a copy of the block literal with its locally defined variables and parameters tagged to differentiate their use within this block from their use within other blocks derived from the same block-literal expression. This action mirrors that undertaken by *tag_method_variables* when invoking a new method. The final action necessary in the creation of a new block object is the binding of the current environment. Assuming the parameters and variables defined for both the surrounding code and the block itself are suitably tagged, the only necessary item to record is the self object. Through the self object it is possible to access not only the object itself, but also any instance variables of that object. The function is finished off by creating a new entry for the object space *OopMap*, entering it, and returning an updated object space and the new object identity.

$$\mathit{prepare\_new\_block} : \mathit{BlockLiteral} \times \mathit{ObjectSpace} \rightarrow \mathit{Oop} \times \mathit{ObjectSpace}$$

$$
\begin{array}{l}
\mathit{prepare\_new\_block}(\mathit{block\_literal}, \sigma) \quad \triangleq \\
\quad \mathsf{let}\ \mathit{block\_oop} = \mathit{new\_block\_object\_id}(\sigma) \\
\quad\quad \mathsf{in} \\
\quad \mathsf{let}\ \mathit{block\_literal}' = \mathit{tag\_block\_variables}(\mathit{block\_literal}, \sigma) \\
\quad\quad \mathsf{in} \\
\quad \mathsf{let}\ \mathit{context\_self} = \mathit{Refs}(\sigma(\delta))(\mathit{self}) \\
\quad\quad \mathsf{in} \\
\quad \mathsf{let}\ \mathit{block\_oop\_maplet} = \\
\quad\quad\quad\quad \{\mathit{block\_oop} \mapsto \textit{mk-BlockObject}(\mathit{block\_literal}', \mathit{context\_self})\} \\
\quad\quad \mathsf{in} \\
\quad \mathsf{let}\ \sigma' = \mu(\sigma, \mathit{OopMap} \mapsto \mathit{OopMap}(\sigma) \cup \mathit{block\_oop\_maplet}) \\
\quad\quad \mathsf{in} \\
\quad (\mathit{block\_oop}, \sigma')
\end{array}
$$

New_block_object_id follows the same pattern as both *new_concrete_object_id* and *new_virtual_object_id* in returning an object identity which does not already occur within the object space.

$$\mathit{new\_block\_object\_id} : \mathit{BlockLiteral} \times \mathit{ObjectSpace} \rightarrow \mathit{BlockOop}$$

$$\mathit{new\_block\_object\_id}\ (\mathit{block}\colon \mathit{BlockLiteral}, \sigma\colon \mathit{ObjectSpace})\ \mathit{b\_oop}\colon \mathit{BlockOop}$$

$$\mathsf{post}\ \mathit{b\_oop} \notin \mathsf{dom}\, \mathit{OopMap}(\sigma)$$

As with *tag_method_variables*, the definition of *tag_block_variables* has been omitted, but its signature follows.

$$\mathit{tag\_block\_variables} : \mathit{BlockLiteral} \times \mathit{ObjectSpace} \rightarrow \mathit{BlockLiteral}$$

In order to accommodate the new block object, the resolution process, performed during a message send to determine the receiver object, has to be altered. Specifically, the activity requiring the alteration is *get_class_and_*δ*_set*, which is called from *find_rcvr_class* to determine the class of the receiver object. In addition to the virtual and class-derived object cases, an extra case needs to be added to allow for block objects. Now, if the receiving object is a block object (*BlockObject*), the function returns the class name BlockClosure.

$get\_class\_and\_\delta\_set : Oop \times DfEnv \rightarrow (ClassName \times DfEnv)\text{-set}$

$get\_class\_and\_\delta\_set(oop, \delta) \quad \triangleq$

let $object = OopMap(\sigma(\delta))(oop)$

in

cases $object$ of

$mk\text{-}VirtualObject(cdo\_set, neqs) \rightarrow \bigcup\{resolve\_v\_obj(oop, cdo, \delta)$

$\mid cdo \in cdo\_set\}$

$mk\text{-}ClassDerivedObject(class, i\_vs) \rightarrow \{(class, \delta)\}$

$mk\text{-}BlockObject(block, cself) \rightarrow \{(BlockClosure, \delta)\}$

end

One final item to note is that the analysis system has so far made no special allowances for *virtual* objects being resolved into instances of class BlockClosure. In fact, this is quite allowable by the system. The only point when the lack of 'structure' (i.e. code) of a resolved block becomes important is when it is invoked by the value primitive. As for the resolution function *resolve_v_obj_cases*, the function which checks to see if a virtual object can be equated with any predefined objects or fully resolved virtual objects, the definition on page 139 still holds good simply because there are no predefined block objects, and hence no special cases involving *BlockObject*s to worry about. Virtual objects which have been resolved into instances of class BlockClosure are simply recorded as class-derived objects and are treated as such by the analysis mechanism.

With these simple alterations to the basic flow analysis functions in place, it is now possible to move on to the primitive handling functions to complete the required modifications for block objects.

6.6 Alterations to the primitives

The only real alteration to the *Invoke_Primitive* function is the inclusion of an entry for the value primitive, which is handled by the *evaluate_block* function. In addition, the function handling the class primitive has to be altered to accommodate the special class introduced by the block object, namely BlockClosure.

$$Invoke\_Primitive : PrimitiveId \times Oop \times Oop^{*} \times DfEnv \rightarrow DfSet$$

$Invoke\_Primitive(mk\text{-}PrimitiveId(primitive), rcvr\_oop, arg\_oops, \delta) \quad \triangleq$

cases $primitive$ of

$\ldots$

$value \rightarrow evaluate\_block(rcvr\_oop, arg\_oops, \delta)$

$class \rightarrow find\_class(rcvr\_oop, \delta)$

$\ldots$

end

Evaluate_block, the function which handles the invocation of a block, is very similar to *DfInvocation*, the function with handles the invocation of a user-defined method. The function starts by extracting the block object to be invoked. At this point it is necessary to test if the receiver object is in fact a fully formed *BlockObject* rather than a virtual object which has been resolved into an instance of class BlockClosure. If the receiving object is only a resolved virtual object, then the analysis for this case must be terminated. This occurs with a call of the *lack_of_block_code* function which is assumed to work in a similar way to both *primitive_failure* and *does_not_understand*, with control being passed to a different case. However, unlike the latter two functions, with *lack_of_block_code* there is the knowledge that the analysis could have continued had the block been fully formed.

If the object is a fully formed block, the function sets the proper context for the block's invocation by resetting the self object to the correct object for that block. Next, the parameters are passed using *pass_args* (page 120), and any temporary variables are initialised using *add_nil_objects* (page 100). Finally the body of the block is 'invoked', or rather, analysed using *DfExpression*.

$$
\begin{array}{l}
\textit{evaluate\_block} : \textit{Oop} \times \textit{Oop}^{*} \times \textit{DfEnv} \rightarrow \textit{DfSet} \\
\textit{evaluate\_block}(\textit{block\_oop}, \textit{arg\_oops}, \delta) \quad \triangleq \\
\quad\quad \textsf{let } \textit{block\_object} = \textit{OopMap}(\sigma(\delta))(\textit{block\_oop}) \\
\quad\quad\quad \textsf{in} \\
\quad\quad \textsf{if } \textit{block\_object} \notin \textit{BlockObject} \\
\quad\quad \textsf{then } \textit{lack\_of\_block\_code}(\textit{block\_oop}, \textit{arg\_oops}, \delta) \\
\quad\quad \textsf{else let } \delta' = \mu(\delta, \sigma \mapsto \textit{assign}(\textit{self}, \textit{ContextSelf}(\textit{block\_object}), \sigma(\delta))) \\
\quad\quad\quad\quad \textsf{in} \\
\quad\quad\quad \textsf{let } \textit{params} = \textit{Params}(\textit{Block}(\textit{block\_object})) \\
\quad\quad\quad\quad \textsf{in} \\
\quad\quad\quad \textsf{let } \delta'' = \mu(\delta', \sigma \mapsto \textit{pass\_args}(\textit{params}, \textit{arg\_oops}, \sigma(\delta'))) \\
\quad\quad\quad\quad \textsf{in} \\
\quad\quad\quad \textsf{let } \textit{temps} = \textit{Temps}(\textit{Block}(\textit{block\_object})) \\
\quad\quad\quad\quad \textsf{in} \\
\quad\quad\quad \textsf{let } \delta''' = \mu(\delta'', \sigma \mapsto \textit{add\_nil\_objects}(\textit{temps}, \sigma(\delta''))) \\
\quad\quad\quad\quad \textsf{in} \\
\quad\quad\quad \textit{DfExpression}\,[\![\textit{Body}(\textit{Block}(\textit{block\_object}))]\!]\,\delta'''
\end{array}
$$

The new version of *find_class* follows the same pattern as before, but before the class name is extracted from the object, a test is made to see if the object is a block object. If it is a block, then BlockClosure is substituted as the class name, ready for *class_as_oop* to translate it into 'object form'.

$$
\begin{array}{l}
\textit{find\_class} : \textit{Oop} \times \textit{DfEnv} \rightarrow \textit{DfSet} \\
\textit{find\_class}(\textit{oop}, \delta) \quad \triangleq \\
\quad\quad \textsf{let } \textit{object} = \textit{OopMap}(\sigma(\delta))(\textit{oop}) \\
\quad\quad\quad \textsf{in} \\
\quad\quad \textsf{let } (\textit{class\_oop}, \sigma') = \textsf{if } \textit{object} \in \textit{BlockObject} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textsf{then } \textit{class\_as\_oop}(\textit{BlockClosure}, \sigma(\delta)) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textsf{else } \textit{class\_as\_oop}(\textit{Class}(\textit{object}), \sigma(\delta)) \\
\quad\quad\quad \textsf{in} \\
\quad\quad \textsf{let } \delta' = \mu(\delta, \sigma \mapsto \sigma') \\
\quad\quad\quad \textsf{in} \\
\quad\quad \{\mu(\delta', \textit{last} \mapsto \textit{class\_oop})\}
\end{array}
$$

That concludes the necessary alterations to the analysis functions. With these relatively minor alterations, it is possible to analyse expressions involving blocks, including control structures. As can be seen in the function which handles the 'invoking' of a block, *evaluate_block*, the actual analysis process is basically the same as that used for the analysis of user-defined methods. This ties in with the observation in §6.1 that it is not the analysis of a block that poses a problem for the analysis mechanism, but the modeling of the context within which it may be invoked. This problem, for the simple language presented in this thesis, has already been dealt with by the handling of the equivalence (==) primitive.

In the following section, the analysis of blocks is rounded off by presenting the necessary alterations to the recursion control functions.

6.7 Alterations to recursion control

The introduction of block objects necessitates some more significant alterations to the recursion control functions than the addition of the perform primitive (§5.3.1). Although the basic element of the recursion control environment remains unchanged:

$$\begin{array}{rrcl} \mathit{Recurs} :: & \mathit{Source} & : & [\mathit{MessageSource}] \\ & \mathit{Sel} & : & \mathit{Selector} \\ & \mathit{Args} & : & \mathit{ArgStruct}^{*} \end{array}$$

some of the elements which make up the *recurs* have changed. The source, or location of a message, has been changed to reflect the fact that a message send may occur inside an invoked block. Message sources may now either be inside a method body (*MethodSource*) or inside a block body (*BlockSource*).

$$\mathit{MessageSource} = \mathit{MethodSource} \ \cup \ \mathit{BlockSource}$$

The *MethodSource* remains unchanged from when it was originally defined. It consists of a class name and method selector to determine within which method the message send occurs, and a marker to distinguish the right message send within that method.

$$\begin{array}{rrcl} \mathit{MethodSource} :: & \mathit{Marker} & : & \mathit{Id} \\ & \mathit{Sel} & : & \mathit{Selector} \\ & \mathit{Class} & : & \mathit{ClassName} \end{array}$$

The *BlockSource* is similar to the *MethodSource*, but with an extra level of indirection. The *BlockSource* first records the block within which the message send occurs. It does this using the *Block_origin* (a *MethodSource*) recorded by the block to locate its defining method and its defining point within that method. With the originating location of the block recorded, the *BlockSource* can make use of the marker associated with the message send to distinguish between different messages within the same block.

$$\begin{array}{rrcl} \mathit{BlockSource} :: & \mathit{Marker} & : & \mathit{Id} \\ & \mathit{Block\_origin} & : & \mathit{MethodSource} \end{array}$$

The *ArgStruct*, the structure which records information about the arguments involved in a message send, has also been extended to take account of the new block object. In addition to the *ClassStruct* and the VOOP, which correspond to the class-derived object and the unresolved virtual object respectively, there is now the *BlockStruct*.

ArgStruct = *ClassStruct* $\cup$ *BlockStruct* $\cup$ {VOOP}

Prior to the introduction of blocks, the control of recursive message sends was based upon the classes of the argument objects involved. For all objects other than blocks, this still stands, with the *recurs* inserted into the method invocation record recording, amongst other things, the classes of the argument objects. However, for block objects involved in message sends, the recursion control must be more refined to prevent the premature detection and termination of a recursive message send. This might occur, for example, if a block is involved in a message send which is subsequently sent with another block as its argument while the original message is still in effect:

```
firstVar aTest
    ifTrue: [ secondVar anotherTest
                ifTrue: [ outcome combine: firstVar and: secondVar ] ]
```

If the first test is true, the outer block,

```
[ secondVar anotherTest
    ifTrue: [ outcome combine: firstVar and: secondVar ] ]
```

will be invoked from within the ifTrue: method, using the value primitive. If the second test is also true, the inner block,

```
[ outcome combine: firstVar and: secondVar ]
```

will also be invoked using the value primitive as part of the same expression within the same ifTrue: method which invoked the outer block. However, if recursion control is applied using only the classes of the blocks in question, the second invocation will not be followed through because it will match the record of the first invocation class for class and source location for source location.

To remedy the situation it is only necessary to record more specific information for block objects, specifically the location of their definition. This is the purpose of the *Block_origin* recorded for each block literal (page 158), and, indirectly, for each block object (page 159).

BlockStruct :: *MethodSource*

This level of discrimination, lying somewhere between differentiating between classes, and differentiating between object identities, works with blocks because of the limited number of originating points which can occur within a program. By discriminating on the point of origin, recursive invocations of individual blocks, from the same source, can be detected without either the problems of being too strict (differentiating by class) or allowing too much freedom (differentiating by object identity). In the latter case, there would be a danger that recursive invocations of blocks originating from the same point would not be detected because each block object would be assigned a separate object identity as part of the analysis of their originating method.

There is, however, one unavoidable drawback in using this technique. In Smalltalk-80, some control structure methods involving blocks, 'wrap' their block arguments in block literals in order that the compiler can take short-cuts in the binding of subsequent message sends. For example the whileTrue method defined for blocks (class BlockClosure) wraps the receiver in a block literal to allow for the full implementation of whileTrue: (of which whileTrue is a special case) to be effectively bound statically:

```
whileTrue
    "Evaluate receiver once and then repeatedly while it still evaluates to true."
    ↑[ self value ] whileTrue: [ ]
```

Problems then start to arise when 'while' loops are embedded within one another. In the following example the outer block would interfere with the inner block through the wrapper block constructed in the whileTrue method shown above.

```
[ [ x msg ] whileTrue ] whileTrue
```

The outer block,

```
[ [ x msg ] whileTrue ]
```

would be 'wrapped' and sent the message whileTrue:

```
[ [ [ x msg ] whileTrue ] value ] whileTrue: [ ]
```

at which point it would be invoked, resulting in the sending of the whileTrue message to the inner block:

```
[ x msg ]
```

At this point the inner block would also be 'wrapped' and sent the message whileTrue:

```
[ [ x msg ] value ] whileTrue: [ ]
```

However, because both entries in the method invocation record would only record information about the wrapper block,

```
[ ... value ]
```

the *mir* entry originating from the outer block would match the entry for the inner block, resulting in a premature detection of recursion at the point where the inner block was about to be processed.

The above limitation notwithstanding, now that the revised elements of the recursion control mechanism have been introduced, the rest of the necessary alterations can be introduced.

6.7.1 Alterations to the extraction, recording and testing of recursion

When it comes to constructing a *recurs* from the state of a message send, the function which starts the process off, *extract_recurs*, remains unchanged from the version defined on page 128. However, both the function which extracts and records the location of the message send, and the function which extracts information relating to the arguments of the message send have changed from the versions defined on pages 128 and 144 (*extract_source*) and page 129 (*extract_arg_seq*).

The basic operation of the new version of *extract_source* follows the same routine of extracting the last entry in the method invocation record, if there is one, to see in which method the current message send is situated. As before, the *mir* entry is broken down to extract both the selector of the method and the *ArgStruct* recording information about the receiver. The reader may recall that complications were introduced by the perform primitive, requiring a special case to be made of message sends occurring inside primitive methods, namely the perform primitive. The introduction of block objects adds to this complexity by introducing another source for message sends *and* another primitive method within which message sends can occur.

The problem of determining the location of a message send is solved by first distinguishing between sources which involve a block as the self object (i.e. the receiver), and those which involve a normal *ClassStruct*. The former case is handled by *extract_block_source*, while the latter case is handled by *extract_method_source*.

$extract\_source : (Id \cup \{\mathsf{nil}\}) \times MethodInvocationRecord \rightarrow (MessageSource \cup \{\mathsf{nil}\})$

$extract\_source(marker, mir) \quad \triangleq$

if $mir = [\,]$

then nil

else let $last\_recurs = mir[\mathsf{len}\, mir]$

in

let $last\_sel = Sel(last\_recurs)$

in

let $last\_rcvr\_struct = \mathsf{hd}\, Args(last\_recurs)$

in

if $last\_rcvr\_struct \in ClassStruct$

then $extract\_method\_source(marker, last\_sel,$

$Class(last\_rcvr\_struct), last\_recurs)$

else $extract\_block\_source(marker, last\_sel, last\_rcvr\_struct, last\_recurs)$

Extract_method_source, the function which handles the *ClassStruct* case, reduces the problem to differentiating between a user-defined method source and a primitive source, which, in this case, can only be the perform primitive. The arguments passed are the positional marker of the message send being located, the selector and class of the method in which it occurs, and finally the *recurs* entry from which this information has been extracted. If the source method is, in fact, a user-defined method, then the source for the message send is constructed out of the message's marker and the class and method selector of the method in which it occurs.

If, on the other hand, the source is a primitive method, the message send must inherit the class and message selector from the source of the perform primitive which constructed it, in the same way in which it inherited the primitive's positional marker (*evaluate_message*, page 142). The inheritance of this source information is achieved by extracting the source field of the primitive's *mir* entry. With this information it is then possible to construct a source for the new message send based on the source of the current method: a *MethodSource* or a *BlockSource* as necessary.

$extract\_method\_source : Id \times Selector \times ClassName \times Recurs \rightarrow MessageSource$

$extract\_method\_source(marker, sel, class, recurs) \quad \triangleq$

if $Methods(program(class))(sel) \in UserMethod$

then $mk\text{-}MethodSource(marker, sel, class)$

else let $last\_source = Source(recurs)$

in

if $last\_source \in MethodSource$

then $mk\text{-}MethodSource(marker, Sel(last\_source), Class(last\_source))$

else $mk\text{-}BlockSource(marker, Block\_origin(last\_source))$

Extract_block_source, the second 'source-extraction' case and the one dealing with blocks, is only really interested in cases involving invoked blocks. That is, cases in which the message requiring a source occurs inside an invoked block. All other cases can be handled by *extract_method_source*. The function takes a similar set of arguments to *extract_method_source*, namely the message's marker, the selector of the source method in which the message apparently occurs, the *BlockStruct* representing the receiver for the source method, and finally the *mir* entry for the source method. The function first makes sure that the source method isn't a user-defined method. If it is, the task is handed over to *extract_method_source*. Next, a test is made to see if the now-confirmed primitive method's selector is the block-invoking value. If it is, then a *BlockSource* is created out of the message's marker and the block's declared origin, recorded in the *BlockStruct*. If the primitive's selector isn't value, in which case it must be perform, the problem is once again handed over to *extract_method_source*.

$extract\_block\_source : Id \times Selector \times BlockStruct \times Recurs \rightarrow MessageSource$

$extract\_block\_source(marker, sel, mk\text{-}BlockStruct(block\_origin), recurs) \quad \triangleq$

if $Methods(program(BlockClosure))(sel) \in UserMethod$

then $extract\_method\_source(marker, sel, BlockClosure, recurs)$

else if $Primitive(Methods(program(BlockClosure))(sel)) = value$

then $mk\text{-}BlockSource(marker, block\_origin)$

else $extract\_method\_source(marker, sel, BlockClosure, recurs)$

Extract_arg_seq has been altered to take account of the new *BlockStruct*. The new version follows the same recursive pattern as the original version (page 129), with the sequence of argument object identities being analysed recursively, one at a time. As each identity is processed, its associated object body is tested to see if it is a block object. If it is, then a *BlockStruct* is constructed, otherwise the problem is passed on to *extract_arg*.

$$
\begin{array}{l}
\mathit{extract\_arg\_seq} : \mathit{Oop}^* \times \mathit{ObjectSpace} \rightarrow \mathit{ArgStruct}^* \\
\mathit{extract\_arg\_seq}(\mathit{arg\_oops}, \sigma) \quad \triangleq \\
\quad \textsf{if } \mathit{arg\_oops} = [\,] \\
\quad \textsf{then } [\,] \\
\quad \textsf{else let } \mathit{hd\_arg\_oop} = \textsf{hd}\, \mathit{arg\_oops} \\
\qquad\quad \textsf{in} \\
\qquad \textsf{if } \mathit{OopMap}(\sigma)(\mathit{hd\_arg\_oop}) \in \mathit{BlockObject} \\
\qquad \textsf{then let } \mathit{block\_object} = \mathit{OopMap}(\sigma)(\mathit{hd\_arg\_oop}) \\
\qquad\qquad \textsf{in} \\
\qquad\quad [\mathit{mk\text{-}BlockStruct}(\mathit{Block\_origin}(\mathit{Block}(\mathit{block\_object})))] \\
\qquad\qquad\quad \frown \mathit{extract\_arg\_seq}(\textsf{tl}\, \mathit{arg\_oops}, \sigma) \\
\qquad \textsf{else } [\mathit{extract\_arg}(\mathit{hd\_arg\_oop}, \sigma, \textsc{Depth\_Limit})] \\
\qquad\qquad\quad \frown \mathit{extract\_arg\_seq}(\textsf{tl}\, \mathit{arg\_oops}, \sigma)
\end{array}
$$

Extract_arg, originally defined on page 129, has also needed to be modified, specifically to handle instance variables which have been assigned a block object. The new block objects are handled by inserting a test for them, and creating a *BlockStruct* as necessary.

$$
\begin{array}{l}
\mathit{extract\_arg} : \mathit{Oop} \times \mathit{ObjectSpace} \times \mathbb{N} \rightarrow \mathit{ArgStruct} \\
\mathit{extract\_arg}(\mathit{arg\_oop}, \sigma, \mathit{depth\_lim}) \quad \triangleq \\
\quad \textsf{let } \mathit{arg\_object} = \mathit{OopMap}(\sigma)(\mathit{arg\_oop}) \\
\qquad \textsf{in} \\
\quad \textsf{if } \mathit{arg\_object} \in \mathit{VirtualObject} \\
\quad \textsf{then } \textsc{voop} \\
\quad \textsf{else if } \mathit{arg\_object} \in \mathit{BlockObject} \\
\qquad \textsf{then } \mathit{mk\text{-}BlockStruct}(\mathit{Block\_origin}(\mathit{Block}(\mathit{arg\_object}))) \\
\qquad \textsf{else if } \mathit{depth\_lim} = 1 \\
\qquad\quad \textsf{then } \mathit{mk\text{-}ClassStruct}(\mathit{Class}(\mathit{arg\_object}), \textsf{nil}) \\
\qquad\quad \textsf{else let } \mathit{arg\_class} = \mathit{Class}(\mathit{arg\_object}) \\
\qquad\qquad\quad \textsf{in} \\
\qquad\qquad \textsf{let } \mathit{inst\_vars} = \{\mathit{i\_v} \mapsto \mathit{extract\_arg}(\mathit{Refs}(\sigma)(\mathit{i\_v}), \sigma, \mathit{depth\_lim} - 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathit{i\_v} \in \mathit{Inst\_vars}(\mathit{arg\_object})\} \\
\qquad\qquad\quad \textsf{in} \\
\qquad\qquad \mathit{mk\text{-}ClassStruct}(\mathit{arg\_class}, \mathit{inst\_vars})
\end{array}
$$

Testing to see if two *recurs* are equivalent remains unchanged at the top level. That is, the definition of *equiv_recurs* itself has remained unchanged:

$$
\begin{aligned}
&\mathit{equiv\_recurs} : \mathit{Recurs} \times \mathit{Recurs} \to \mathbb{B}\\
&\mathit{equiv\_recurs}(\mathit{new\_recurs}, \mathit{old\_recurs}) \quad \triangleq\\
&\qquad (\mathit{Sel}(\mathit{new\_recurs}) = \mathit{Sel}(\mathit{old\_recurs}))\\
&\qquad\quad \wedge (\mathit{Source}(\mathit{new\_recurs}) = \mathit{Source}(\mathit{old\_recurs}))\\
&\qquad\qquad \wedge\ \mathit{equiv\_args}(\mathit{Args}(\mathit{new\_recurs}), \mathit{Args}(\mathit{old\_recurs}))
\end{aligned}
$$

In fact, *equiv_args*, called from *equiv_recurs*, also remains unchanged from the version defined on page 132. However the function which performs the test proper, *equiv_arg_struct* (page 132), has been altered to take account of the new *BlockStruct*.

$$
\begin{aligned}
&\mathit{equiv\_args} : \mathit{ArgStruct}^* \times \mathit{ArgStruct}^* \to \mathbb{B}\\
&\mathit{equiv\_args}(\mathit{new\_args}, \mathit{old\_args}) \quad \triangleq\\
&\qquad \mathsf{len}\ \mathit{new\_args} = \mathsf{len}\ \mathit{old\_args}\\
&\qquad\quad \wedge\ \forall\ i \in \mathsf{inds}\ \mathit{new\_args} \cdot \mathit{equiv\_arg\_struct}(\mathit{new\_args}[i], \mathit{old\_args}[i])
\end{aligned}
$$

Those cases not dealing with *BlockStructs* are as they were originally defined. The new cases are defined as follows.

$$\mathit{equiv\_arg\_struct} : \mathit{ArgStruct} \times \mathit{ArgStruct} \to \mathbb{B}$$

Two *BlockStruct*s are equivalent if they have the same point of origin.

$$
\begin{aligned}
&\mathit{equiv\_arg\_struct}(\textit{mk-BlockStruct}(\mathit{new\_blck}), \textit{mk-BlockStruct}(\mathit{old\_blck})) \quad \triangleq\\
&\qquad \mathit{new\_blck} = \mathit{old\_blck}
\end{aligned}
$$

Finally, for those cases involving a mismatch between a *BlockStruct* and another kind of *ArgStruct*, the function returns false.

$$
\begin{aligned}
&\mathit{equiv\_arg\_struct}(\textit{mk-BlockStruct}(\mathit{block\_oop}), \mathit{old\_arg}) \quad \triangleq\\
&\qquad \mathsf{false}
\end{aligned}
$$

where

$$\mathit{old\_arg} \notin \mathit{BlockStruct}$$

$$
\begin{aligned}
&\mathit{equiv\_arg\_struct}(\mathit{new\_arg}, \textit{mk-BlockStruct}(\mathit{block\_oop})) \quad \triangleq\\
&\qquad \mathsf{false}
\end{aligned}
$$

where

$new\_arg \notin BlockStruct$

With respect to the basic properties of the amended recursion control, all that has changed is that recursion has been specialised for the finite number of blocks which can occur within a given program. Instead of distinguishing between receiver and argument objects by their classes, blocks are distinguished by their defining point within the program code. As for issues relating to the termination of the analysis process, termination can still be guaranteed, because the new block 'originating points' are finite in number. Consequently there will only be a finite number of new *recurs* combinations before a recursive send involving one is detected and acted upon to terminate that case.

6.8 Potential extensions and possible problems

In this section we will consider three potential extensions to the dataflow analysis system — 'integer' objects, collection-type objects and user input — and speculate on how they might be incorporated in the dataflow analysis system.

Adding integers would necessitate adding a new class whose instances have a special property, namely ordering. That is, the ability to take two or more objects of that class and place them in a pre-determined order. The question immediately arises as to how to handle such a property? A clue to a solution can be found in the handling of object equivalence in Chapter 5. Implementing object equivalence required that the virtual object be altered so that the result of such comparisons could be recorded as necessary. In the same way, the result of an ordering comparison test could be recorded for the new ordered objects.[4]

The new kind of object, required to handle the ordering information, would be modeled as a kind of class-derived ordered object, and would only apply to resolved virtual objects. Explicitly created integer objects would have their value, and hence ordering, fixed as such objects would be immutable. Their value could be recorded in their object 'body':

$ExplicitIntegerObject :: Value : \mathbb{Z}$

The new kind of ordered object would be created when an unresolved virtual object is finally resolved, either through receiving a message, or by being involved in an ordering comparison. The new object would require four fields in order to handle the outcomes of each possible comparison between two integers (less than, less than or equal to, etc.):

$$\begin{array}{lrl} OrderedIntegerObject :: & Less\_than & : Oop\text{-set} \\ & Less\_than\_or\_equal & : Oop\text{-set} \\ & Greater\_than & : Oop\text{-set} \\ & Greater\_than\_or\_equal & : Oop\text{-set} \end{array}$$

[4]On face value, this might appear to be a case of 'hard-wiring' a software feature. However ordering, like equivalence, is actually a basic property of the objects involved, implemented at a primitive level in the program code. Again, as with the equivalence primitive, higher-level ideas of equality or ordering must eventually be implemented at the primitive level.

Equality of two such ‘resolved’ integer objects would merely mean the replacement of one by the other and the merging of the information recorded in their respective fields. As the new object is compared to other integer objects, cases would be generated, with the assumption for each case being recorded in the relevant field. During the lifetime of the object it would build up sets of other integer object identities, providing a picture of where it stands in the order of things.

The introduction of arithmetic primitives for such integers would involve attempting to infer the ordering details for the result of the operation, which would itself be a new *OrderedInteger-Object*. The amount of information passed on from the operands to the result would depend on the ‘resolved’ integer operands having ordering information based on ‘explicit’ integer objects. For example, addition involving ‘resolved’ integer objects would depend on ordering information relative to ‘explicit’ integers, preferably zero (`0`), in order to help establish if both operands are positive or negative, and hence whether the result will be more positive or more negative than the operands. If there is no ‘base line’ against which the operands can determine their positiveness or negativeness, or if the signs of the operands are different, then the resulting integer object would start its existence with no ordering information about itself.

In a sense, the ‘resolved’ integer object remains virtual for the rest of its separate existence, either for the rest of its lifetime, or until it is tested for equivalence with an explicitly created integer. This is possible because of the immutable nature of the integer. Irrespective of the actual object identity assigned to the new ‘virtual’ integer, so long as it records its relationship with other integer objects, ‘virtual’ or ‘explicit’, the model will hold good. When the identity of the integer has to be known for sure, as in the case of a test for object identity equality, cases may be generated for assumptions of both equality and non-equality without any problems of inconsistency being introduced — the actions of the ‘virtual’ integer prior to the test will not have affected its state in any way that could create such inconsistencies between actions and assumptions.

Adding arrays, or collection-type objects, introduces further complexities. Collection-type objects are effectively collections of objects which are indexable, at least at an implementation level, by some class of ordered object, for example integers. The main problem with such collections lies in recording information, not only about which objects may be resident in the collection, but also about the respective positions of the various objects within the collection. In order to achieve these requirements, it would seem useful to utilise the ordering information of the ‘virtual’ integer objects, described above, when recording the objects assigned to, or accessed from such a collection.

Recording the structure of a collection object involves dealing with two types of objects: objects of known position, and objects known to exist within the collection but whose position is not known explicitly. In addition, the collection could be assumed to be populated by unresolved virtual objects for all slots not covered by objects of the first two kinds. However because such virtual objects would be resolved into one of the first two kinds the first time they are accessed, there is no need to record them explicitly. Of the recorded objects, the first kind, those of known position, would be made up of those objects either assigned to the collection with an explicit integer as the index, or resolved out of a virtual object when the collection was accessed with an explicit integer index. For the second kind, those known to exist in the collection but of unknown position, such objects would have been assigned, or resolved during an access, using a 'resolved' integer object as the index. The ordering information provided by the 'resolved' integer would, hopefully, provide some information to fix the object within a subrange of the overall collection structure.

Clearly any overlap between indexing by various combinations of 'resolved' integer objects (e.g. an object is known to exist somewhere between position `3` and position `10`) and explicit integers (e.g. another object is known to exist at position `6`) would require the generation of more cases to handle all eventualities.[5]

Finally, adding some form of input primitive would require even more alterations to the analysis mechanism. However there are two factors which should simplify the situation. Because of the simplistic nature of user input in Smalltalk, the question of whether a user-entered object is equivalent (==) to an object already in the system would only seem to arise when dealing with simple literal expressions, for example numbers (integers) or characters. All other, more complex objects can be considered unique, as are all objects explicitly created within the program code. For example, user input in the form of a screen coordinate, `300@424`, would imply a new instance of class `Point`, but the point will have two immutable integer components. This leads onto the second factor, namely that all simple literal expressions would create immutable objects, or would be made up from such immutable objects. Therefore the integers `300` and `424` would cause the re-use of the `300` and `424` objects already present in the system, or the creation of them if they did not already exist. Their immutability would ensure that no inconsistencies would be introduced by using them. A useful test for equality between points, therefore, would depend on testing for equivalence between the points' components, which, because of their immutability and hence re-usability, could be tested under the fundamental test for object identity equality.

Following the reasoning presented here, it would seem plausible that user input, with its combination of known class and unknown state, could be modelled by some form of special 'concrete-virtual' object, whose point of creation, and hence identity is known, but whose state is unknown. As messages are sent to the new object, its state will gradually be resolved, with its instance variables, or components in the case of collection objects being assigned either new 'concrete-virtual' objects or immutable objects which are already present in the object space.

6.9 Benefits of dataflow analysis

Traditional dataflow analysis usually begins with: "Construct a control-flow graph" (for example, [AC76]). Unfortunately this is a non-trivial task for Smalltalk-80, where the dynamic binding of message sends to methods means that control-flow graphs cannot be constructed beyond the assumed flow through a single method. For example, consider the flow through the following method:

[5] It might be that the generation of such cases would interfere assumption-wise with the resolution of 'virtual' integer objects into explicit integers, though this area remains unclear.

```
computeEffect: aFactor
    "An example of a complicated evaluation method involving a message
    send to the argument."
    self permute.
    aFactor compress: self permuteFactor.
    self appendFactor: aFactor.
    ↑self
```

The flow through such a method may be visualised as follows:

$$
\begin{array}{cccc}
\rightarrow & \text{enter method} & & \\
 & \downarrow & & \\
 & \text{message send} & \rightarrow & \textit{region} \\
 & \text{message result} & \leftarrow & \textit{unknown} \\
 & \downarrow & & \\
 & \text{message send} & \rightarrow & \textit{region} \\
 & \text{message result} & \leftarrow & \textit{unknown} \\
 & \downarrow & & \\
 & \text{message send} & \rightarrow & \textit{region} \\
 & \text{message result} & \leftarrow & \textit{unknown} \\
 & \downarrow & & \\
\leftarrow & \text{return result} & &
\end{array}
$$

The problem is that in the vast majority of cases there is no simple way of determining what will happen during each of the message sends (hence the *regions unknown*). This is especially true of the message sent to the argument, aFactor. It follows that if the state of the variables after each message send is unknown then all the traditional forms of dataflow analysis, based as they are on complete control-flow graphs, cannot be applied to Smalltalk.

It is here that the dataflow analysis mechanism, described and defined in this and the preceding two chapters, comes to the rescue. The analysis mechanism itself embodies the control-flow of the code, and so as it processes the program, class by class, method by method and expression by expression, it traces out a control-flow graph through the code.

In addition, as the process moves through the code, it continually updates its abstracted model of the object space that will potentially occur at run time. By comparing the state of this abstracted object space (σ in the dataflow environment) before and after the analysis of an expression and noting the result, an unprejudiced picture of the dataflow through the code can be built up in parallel with the control-flow graph.

In this way, some of the basic elements used in dataflow analysis can be recorded. For example the sets of reaching definitions for the variables used in a section of code, separated out into those definitions which are either *killed* or *generated* by ambiguous or unambiguous assignments, as well as those definitions which are valid at the start of a section of code (the *in* set) or at the end of that section (the *out* set) ([ASU86], §10.5).

Following on from the model used in [ASU86], where the sets together describe what may potentially be currently assigned to a given variable, an additional *altered* set could be added.

Such a set would describe those variables involved as arguments to a message send, for whom the class of object currently assigned will not have changed (because method parameters cannot be assigned to directly) but whose current state will be unknown. Such a technique would be useful in 'special-casing' the compilation of some methods, whereby in addition to a normally compiled version, a special version is compiled for use with arguments of a particular class.[6] By determining that the class of an object has not changed, even though its state may have, the continued static binding of message sends to that object, within that method, may be carried out with confidence, where previously the spectre of the dreaded become: primitive (§5.3.3) may have prevented this. Of course, the become: primitive is still present, and is still able to cause problems, but because of the exhaustive nature of the analysis mechanism in following up message sends to all potential methods, it should be possible to isolate those control paths which might invoke this primitive.

In summary, the main benefit provided by dataflow analysis lies in the improved information extracted from the code. In addition to the benefits provided by the raw information generated, by recording the association between an object name (variable, parameter etc.) and an object identity, not only can object 'aliasing' be handled naturally, but also the problems experienced by the type system with re-assignments to variables can be avoided. By performing simulated lookups, 'does-not-understand' situations can be detected and noted in the same way that the type inference system of Chapter 3 detects them. Finally, the analysis mechanism allows a simple way of providing selective analysis, whereby the resolution of a virtual object is limited to a small set of classes.

6.10 Is dataflow analysis a form of type checking?

So what is type checking? Is it just the elimination of does-not-understand situations? If so, then are we dealing with types in the sense of those dealt with by a type inference system, or are we dealing strictly with classes? If it is the latter, surely we are not 'type checking', but rather 'class checking'.

Looking at it another way, it could be argued that there are two forms of type checking. The first is type checking in the traditional sense, whereby the program code is consistent with respect to a particular typing of that code. In other words each component is assigned a type and the resulting typed code is shown to be consistent (e.g. the type of an object assigned to a variable is consistent with the type of the variable). In effect this is what Graver's type inference system is based on — Chapter 2 of his thesis [Gra89] is concerned with this very topic, and describes a set of inference rules which can be used to ensure that a given typing is consistent. The important point of this traditional view of type checking is that everything is evaluated within the domain of types.

[6] A form of special-casing, or selective analysis, can be carried out in the dataflow analysis system. By doctoring the set of potential object classes into which a virtual object can be resolved (page 94), the analysis process can be restricted to those cases only. For example, by restricting the set to Integers only, the control paths which can be followed through successfully by the analysis mechanism will be those paths which may potentially benefit from having special-case versions compiled for integers.

A second form of type checking could be based solely on the 'does-not-understand' situation and its elimination. As such, all analysis techniques are open for consideration, including dataflow analysis. In the case of dataflow analysis, by checking out all message/method combinations it should be possible to detect any potential 'does-not-understand' situations. The importance of this approach is that we are not restricted to one domain — types — and so we are able to work around many of the problems such domains might introduce, for example those associated with the assignment operation described in §3.8.1. Therefore, the question is: is this type checking? Since the 'does-not-understand' situation is the result of a mismatch between a message and a receiver — in particular the functionality-defining class of the receiver — it would seem that this is a 'class' issue, and because 'class' is as close to 'type' as you will ever get in 'pure' Smalltalk it would seem that we are talking about type checking.

The real argument comes when you consider the differences between the two forms. In the first, traditional, system there is an emphasis on associating types with variables. In the second, this has been avoided to prevent any problems with the assignment operation. In fact, in the second there is an emphasis on predicting the class of a given object and using this as the basis of determining type-correctness, effectively a form of pre-emptive dynamic type checking.

In conclusion, the traditional approach has been to determine type correctness with respect to a particular assignment of types to variables and expressions. In this case there can be no type checking without types. In the second system, that is the system based on dataflow analysis, type correctness is determined by literally trying out all possible classes for a given object. Clearly with the second system there are no types in the traditional sense: variables derive their 'class' from the object currently assigned to them, and this can change through successive assignments, with no attempt to ensure 'consistency' between such assignments. In the second system, therefore, there would appear to be type checking without types!

6.11 Summary

In this and the preceding two chapters a comprehensive dataflow analysis mechanism has been defined, starting with a basic system in Chapter 4, and then adding extensions to handle primitive methods (Chapter 5) and blocks (Chapter 6). By the formation of the mechanism, it has been shown to reflect similar properties to the type inference mechanism presented in Chapter 3, namely the ability to detect potential 'does-not-understand' situations, but without some of the drawbacks associated with the type system. In addition, the dataflow analysis system has been shown to have greater potential for the analysis of control structures than that possible with a type inference system.

Section 6.8 rounded off the description of the mechanism by presenting three potential extensions to system, as well as suggesting possible ways of implementing such extensions. More research is required to demonstrate that the suggested implementations are both consistent and practical within the overall framework of the analysis model. The last two sections in this chapter have presented some of the more general properties of the analysis mechanism, both as a system in its own right, and with respect to a type inference system.

Chapter 7

Conclusions

Ersatz hope can puff you up and blind you to reality. You then subsist on a diet of fat hopes, sparse deeds, and bitter fruit.

Brendan Francis

This thesis has addressed the problem of improving 'efficiency' within a Smalltalk-like language from two separate points of view: from the perspective of improving programmer efficiency, and from the perspective of the improving the run-time efficiency of an 'end product'. On the basis of observations derived from interpreting these two areas, two code analysis systems were examined, a type inference system and a dataflow analysis system. The ultimate aim was to determine, within a theoretical framework, what could be derived from the application of either of these two contrasting code-analysis systems.

7.1 Initial findings

Chapter 2 drew some early conclusions relating to the issues involved in devising a solution to either of the two 'efficiency' questions. In particular it observed that the incorporation of an additional mechanism to the incremental programming process, be it a type system, type inference system or dataflow analysis system, might bring disadvantages to the programmer, as well as benefits. With this in mind it was decided to evaluate the code analysis systems on the basis of the quality of information extracted from the program code, rather that the efficiency with which such information might be derived. This decision paralleled the observation in the same chapter relating to improving run-time efficiency, namely that the quality of the resulting code is more important that the efficiency with which it is produced.

7.2 Code analysis: types versus dataflow

Of the two code analysis systems examined, Chapter 3 looked at a type inference system based closely on Graver's system for Typed Smalltalk [Gra89], while Chapters 4 through 6 looked at a dataflow analysis system. Each examination took the form of a detailed formal definition of the system in question, from which the properties of that system could then be derived. In particular, the construction of the dataflow analysis system was carried out, not with the intention of deriving a direct implementation, but to establish a basic model from which an implementation could later be derived.

For the type inference system defined in Chapter 3, questions were raised about its potential impact on programming practices, especially with regard to rejecting 'good' code (i.e. code which is acceptable in an untyped implementation, but which is rejected when a type system is introduced). In particular, the problems addressed included the handling of multiple assignments to a single variable (ensuring that each assigned object is compatible with the type of the variable, §3.8.1), and the handling of the return type of a method with multiple potential return points (§3.9). It was noted that the assignment problem in particular seemed to arise as a result of a clash between the "one variable — one type" principle and the semantics of the language being typed, and was therefore a problem which would be suffered by any potential type system.

In contrast, the dataflow analysis system, developed over Chapters 4, 5 and 6, was able to avoid the problems encountered by the type inference system. In particular, the problem associated with multiple assignments was handled naturally. In addition, with a more refined level of control-flow analysis, the system was able to handle conditional executions better that the corresponding type inference system (§6.1).

Both analysis systems were able to detect potential 'does-not-understand' errors. In addition, both systems made use of a form of 'constant pushing', where information taken from a literal expression was used to derive further information about the code using that expression. This was particularly true for Graver's implementation of his type inference system, where some form of bounded type information about the message receiver was required in order for the system to progress with a simulated method lookup — in addition to programmer-supplied type declarations, literals were the only other source of such type information. The dataflow analysis system also used information derived from literal expressions, but, in contrast to the type inference system which recorded only its type, the dataflow system attempted to record information about the object's state.

In conclusion, for a Smalltalk-like language, it would seem that a dataflow analysis system, based on the one presented in this thesis, could provide the same error-detecting properties of a type inference system, but without some of the drawbacks that such a type system would introduce, while at the same time providing more sophisticated control-flow analysis.

7.3 Implementation considerations

The dataflow analysis system defined in this thesis attempts to follow all possible control-flow paths through a piece of code being processed, based on a minimum of *a priori* assumptions.

In cases where there is a complete lack of information influencing the possible actions of an expression, for example when sending a unary message to an unresolved virtual object, there is a real possibility of an exponential explosion of cases to be processed. Such a large number of cases would naturally degrade the usefulness of the information being derived by mixing spurious cases with the intended ones. Graver's type inference system suffers from a similar problem (message sends to unbounded type variables, page 27) and it would seem that the decision not to follow through such cases was based on this reasoning. It would seem prudent that for practical purposes an implementation of a dataflow analysis mechanism would be forced to adopt a similar strategy of abandoning potential control-flow paths which are unlikely to yield useful information.

The question then arises as to what sort of strategy to adopt. This can be answered in part by identifying some general forms of 'information-rich' and 'information-poor' expressions from which, respectively, much or little information might be extracted. Before considering 'information-rich' expressions, let us first look at 'information-poor' expressions.

An 'information-poor' expression can most easily be identified as an expression whose analysis can be seen to involve no prior assumptions. A classic example would be the first message send to an instance variable in a method:

```
msg: par
    instVar testMsg.
    ...
```

Given that the state of the receiver object has not been resolved, the only relevant assumption made, the class of the self object, can have no effect on the analysis of the message send. As a result, if the analysis of this method was 'cold', with no prior analysis leading up to it, then the analysis of all the possible methods the message send might invoke would likewise effectively be 'cold'.

However all is not lost. To make the most of such situations, rather than abandon them, it is necessary to distinguish between common and rare method selectors. If the above method was to be analysed, in a practical, well implemented system, a poorly chosen method selector such as "testMsg" should be quite rare, with only a handful of implementations. In contrast a method selector such as "=" has a significant number of implementations — significant enough to potentially overwhelm the analysis mechanism with 'bogus' analysis paths. With respect to these two extremes, the problem lies in the fact that the analysis of an expression of one kind may lead to an expression of the other kind. In other words, a method with a rare selector (e.g. selection: only implemented in classes MenuTracker and PopUpMenu) may quite easily contain a message send involving a common method selector (e.g. in the case of selection:, the message =). The key to the problem is knowing when you are beaten. By applying a suitable limit to the number of simultaneous analysis paths the analysis may progress along the lines laid out in this thesis, while at the same time preventing the process from running away with itself.

Alternatively, instead of analysing all expressions in the hope that an 'information-poor' expression turns out to be 'information-rich', a systematic search for more obviously 'information-rich' expressions might be carried out. As with Graver's type inference system the key to the problem is *a priori* information — the more you know about the elements in an expression, the more you can say with greater certainty about the potential effects of that expression. The most obviously rich source of such prior information lies with literal expressions.

Any message send to a literal expression has the immediate advantage that there can be no problems associated with message sends to instance variables. In the example code given above, the problem with the instance variable arises because the state of the self object is not known, at least with respect to that instance variable. However with a literal receiver, the state of the self object is known. In contrast with the example above, a classic example of an 'information-rich' expression is a unary message send to a literal expression:

```
msg: par
    #literal testMsg.
    ...
```

All expressions involving literals provide an important source of information which can be utilised by the analysis mechanism. Unfortunately despite their usefulness, expressions involving literals (other than unary message sends) cannot be guaranteed to be truly 'information-rich' expressions. Just as the optimistic analysis of a message send with an uncommon message selector may be ruined by the subsequent use of a common message selector, so too can the analysis of an expression involving a literal be ruined by a subsequent message send involving an unresolved variable. For example the code above may have been:

```
msg: par
    #literal testMsg: par.
    ...
```

The addition of the argument to the message send provides a means for a potentially 'information-rich' expression rapidly to become an 'information-poor' one. In this case the potentially invoked method could quite easily involve a message send to the argument, about which, for the sake of argument, nothing as known:

```
testMsg: anArgument
    anArgument argumentTestMsg.
    ...
```

As with the first approach, the 'information-rich'-only approach would also seem to require an imposed limit to make the analysis process tractable. However, despite such potential pitfalls, literal expressions do provide an important source of information, in the same way that they are essential for Graver's type inference system in the absence of type declarations.

In summary, the question of whether to adopt the "analyse everything" approach used by the dataflow analysis system presented in this thesis, with the addition of an imposed limit, or to utilise a strategy of searching for and analysing only 'information-rich' expressions can really only be answered by taking measurements from an implementation. The results would then depend on the distribution of selector names and the frequency with which they occur, as well as the number of identifiable 'information-rich' expressions. Just as importantly, the fine tuning of the imposed limit, based on the speed of the hardware and the patience of the user, might also have a significant effect, making the difference between allowing a cursory analysis of the whole system or necessitating the careful selection of code segments to make the most of what processor resources are available.

7.4 Future work

It is hoped that the dataflow analysis system presented in this thesis will eventually be implemented in some form. Naturally any useful implementation would most likely require some of the extensions suggested in §6.8. The work of the SELF group [CU91] in applying what they term ‘type analysis’ to a control-flow graph as part of their compiler, has provided encouragement that dataflow analysis does have a potential pay-off with regard to compiling efficient code.

One area which may prove fruitful is the introduction of programmer-provided ‘hints’. Rather than insist on a strict self-consistent typing of code, programmers may be given the opportunity to include class hints in their code, which could then be used to direct the dataflow analysis process. The information derived from the analysis system would then be used to enable the compiler to compile specialised versions of some methods, in a manner similar to the SELF compiler. If, on the other hand, the hints proved to be inconsistent, they would be ignored, with the programmer being notified of such a state. Such hints could in effect be the dataflow analysis answer to type declarations, without the imposition of strict type checking.

Appendix A

Miscellaneous Functions

The following functions were omitted from Chapter 3 for the sake of brevity.

A.1 From §3.4.5, Miscellaneous auxiliary functions

$$fst : X \times Y \rightarrow X$$
$$fst((a, b)) \quad \triangleq \quad a$$

$$snd : X \times Y \rightarrow Y$$
$$snd((a, b)) \quad \triangleq \quad b$$

$$dom\_seq : (X \times Y)^* \rightarrow X^*$$
$$dom\_seq(seq\_of\_pairs) \quad \triangleq$$
$$\quad \text{if } seq\_of\_pairs = [\,]$$
$$\quad \text{then } [\,]$$
$$\quad \text{else } [fst(\text{hd}\, seq\_of\_pairs)] \frown dom\_seq(\text{tl}\, seq\_of\_pairs)$$

$$rng\_seq : (X \times Y)^* \rightarrow Y^*$$
$$rng\_seq(seq\_of\_pairs) \quad \triangleq$$
$$\quad \text{if } seq\_of\_pairs = [\,]$$
$$\quad \text{then } [\,]$$
$$\quad \text{else } [snd(\text{hd}\, seq\_of\_pairs)] \frown rng\_seq(\text{tl}\, seq\_of\_pairs)$$

$seq\_as\_map : (X \times Y)^* \to X \xrightarrow{m} Y$

$seq\_as\_map(seq\_of\_pairs) \quad \triangleq$

$\quad \{a \mapsto b \mid (a, b) \in \text{elems}\, seq\_of\_pairs\}$

where

$\text{len}\, dom\_seq(seq\_of\_pairs) \;=\; \text{card elems}\, dom\_seq(seq\_of\_pairs)$

$set\_to\_seq : X\text{-set} \to X^*$

$set\_to\_seq(a\_set) \quad \triangleq$

$\quad$ if $a\_set = \{\,\}$

$\quad$ then $[\,]$

$\quad$ else let $elem \in a\_set$

$\qquad\quad$ in

$\qquad [elem] \frown set\_to\_seq(a\_set - \{elem\})$

A.2 From §3.5.1, Instantiation and initial typing

$instant\_self\_in\_vars : (AssVar \xrightarrow{m} Type) \times Type \to AssVar \xrightarrow{m} Type$

$instant\_self\_in\_vars(var\_map, self\_type) \quad \triangleq$

$\quad \{var \mapsto subst\_self(var\_map(var), self\_type) \mid var \in \text{dom}\, var\_map\}$

$instant\_self\_in\_methods : (Selector \xrightarrow{m} Method) \times Type \to Selector \xrightarrow{m} Method$

$instant\_self\_in\_methods(methods, self\_type) \quad \triangleq$

$\quad \{sel \mapsto instant\_self\_in\_method(methods(sel), self\_type) \mid sel \in \text{dom}\, methods\}$

$instant\_self\_in\_method : Method \times Type \to Method$

$instant\_self\_in\_method$
$\quad\quad(mk\text{-}UserMethod(params, temps, return, body, type\_vars), self\_type) \quad \triangleq$

let $params' = instant\_self\_in\_params(params, self\_type)$
in
let $temps' = instant\_self\_in\_vars(temps, self\_type)$
in
let $return' = subst\_self(return, self\_type)$
in
let $type\_vars' = instant\_self\_in\_tvs(type\_vars, self\_type)$
in
$mk\text{-}UserMethod(params', temps', return', body, type\_vars')$

$instant\_self\_in\_method(mk\text{-}PrimitiveMethod(params, return), self\_type) \quad \triangleq$

let $params' = instant\_self\_in\_params(params, self\_type)$
in
let $return' = subst\_self(return, self\_type)$
in
$mk\text{-}PrimitiveMethod(params', return')$

$instant\_self\_in\_params : (ParamId \times Type)^* \times Type \rightarrow (ParamId \times Type)^*$

$instant\_self\_in\_params(param\_seq, self\_type) \quad \triangleq$

if $param\_seq = [\,]$
then $[\,]$
else let $(param\_id, type) =$ hd $param\_seq$
in
$[(param\_id, subst\_self(type, self\_type))]$
$\frown$
$instant_self_in_params($tl $param\_seq, self\_type)$

$$instant\_self\_in\_tvs : (TypeVar \times Bound)^* \times Type \rightarrow (TypeVar \times Bound)^*$$

$$\begin{array}{l}
instant_self_in_tvs(typevar_seq, self_type) \quad \triangleq \\
\quad \textsf{if } typevar_seq = [\,] \\
\quad \textsf{then } [\,] \\
\quad \textsf{else let } (type_var, bound) = \textsf{hd}\, typevar_seq \\
\qquad\quad \textsf{in} \\
\qquad [(type_var, subst_self(bound, self_type))] \\
\qquad\quad \frown \\
\qquad instant_self_in_tvs(\textsf{tl}\, typevar_seq, self_type)
\end{array}$$

Appendix B

Summary of VDM notation

A good terminology is half the game

Anonymous, quoted in Bricks to Babel *by A. Koestler*

Here is a glossary of some of the 'old-style' VDM symbols used in this thesis.

$[type]$	$type \cup \{\mathsf{nil}\}$
x-set	(finite) set type constructor
card s	cardinality of set s
$a :: \ c_1 : t_1, \ldots$	a is a composite type, with components c_1 of type $t_1, \ldots$
$\mu(a, c \mapsto v)$	the composite value a, updated at c to v
$x \xrightarrow{m} y$	map type constructor from x to y
$\{a \mapsto 42, \ldots\}$	mapping from a to 42,…
dom m	the domain of map m
rng m	the range (codomain) of map m
$s \ntriangleleft m$	map m with set s removed from the domain
$\{\,\}$	the empty set or empty map
†	map overwrite operator
x^*	sequence type constructor
$[a, b, \ldots]$	the sequence $a, b, \ldots$
len s	the number of elements in sequence s
inds s	the indices of sequence s (= $\{1, \ldots, \mathsf{len}\, s\}$)
elems s	the set of elements in sequence s
$s_1 \frown s_2$	sequences s_1 and s_2 concatenated
hd s	the head (car) of sequence s
tl s	the tail (cdr) of sequence s
$\wedge, \vee, \Rightarrow, \neg$	logical connectives 'and', 'or', 'implies' and 'not'

$\forall, \exists$	Universal and Existential operators
$\iota\ e \in S \cdot p(e)$	return the unique element e in set S such that $p(e)$ holds
$\mathbb{N}, \mathbb{Z}$	natural numbers, integers
$F[\![A]\!]\,b$	function F applied to primary argument A and secondary argument b

Appendix C

Index to Types and Functions

C.1 Index to Types

C.2 Index to Functions

Bibliography

[AC76] F. E. Allen and J. Cocke. A program dataflow analysis procedure. *Comm. ACM*, 19(3):137–147, March 1976.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Atk86] Robert G. Atkinson. Hurricane: An optimizing compiler for Smalltalk. *ACM SIGPLAN Notices*, 21(11):151–158, November 1986. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications.

[Bar77] Jeffery M. Barth. An Interprocedural Data Flow Analysis Algorithm. In *Proceedings of the Fourth ACM Symposium on the Principles of Programming Languages*, pages 119–131, Los Angeles, California, January 1977.

[Bar89] Andrew J. Barnard. On introducing a type system to an object oriented language. Master's thesis, Department of Computer Science, University of Manchester, July 1989.

[BDMN73] G. M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Input Two-Nine, 1973.

[BI82] Alan H. Borning and Daniel H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Proceedings of the Ninth ACM Symposium on the Principles of Programming Languages*, pages 133–141, Albuquerque, New Mexico, 1982.

[Byt81] BYTE, August 1981. Special Smalltalk edition.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on the Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[Cox86] Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[CR86] S. Cook and G. Roberts. A compile-time type-checking system for Smalltalk-80. *Journées Langages Orientés Object*, 48:40–45, January 1986.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *ACM SIGPLAN Notices*, 24(7):146–160, July 1989. ACM SIGPLAN Conference on Programming Language Design and Implementation.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 1–15, Phoenix, Arizona, 1991.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices*, 24(10):49–70, October 1989. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications.

[DG87] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of the 1987 European Conference on Object-Oriented Programming*, pages 201–220, Paris, June 1987. AFCET.

[DS84] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the Eleventh ACM Symposium on the Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, January 1984.

[GJ90] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Proceedings of the Seventeenth ACM Symposium on the Principles of Programming Languages*, pages 136–150, San Francisco, California, 1990.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[Gra89] Justin O. Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1989. UIUCDCS–R–89–1539.

[Hen90] Andreas V. Hense. Polymorphic type inference for a simple object oriented programming language with state. Technical Report A 20/90, Universität des Saarlandes, November 1990.

[Hen91] Andreas V. Hense. Type inference for O'small. Technical Report A 06/91, Universität des Saarlandes, October 1991.

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

[Ing78] Daniel H. H. Ingalls. The Smalltalk-76 Programming System: Design and Implementation. In *Proceedings of the Fifth ACM Symposium on the Principles of Programming Languages*, pages 9–16, Tucson, Arizona, 1978.

[JG88] Ralph E. Johnson and Justin O. Graver. A user's guide to Typed Smalltalk. In *Typed Smalltalk Working Papers: 1988*. Department of Computer Science, University of Illinois at Urbana-Champaign, September 1988. UIUCDCS–R–88–1457.

[JGZ88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An optimizing compiler for Smalltalk. *ACM SIGPLAN Notices*, 23(11):18–26, November 1988. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications.

[Joh86] Ralph E. Johnson. Type-checking Smalltalk. *ACM SIGPLAN Notices*, 21(11):315–321, November 1986. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[Jon89] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall International, second edition, 1989.

[Kam88] Samuel Kamin. Inheritance in Smalltalk-80: a denotational definition. In *Proceedings of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 80–87, San Diego, California, January 1988.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[Kra83] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983. Reprinted with corrections, June 1984.

[LB83] J. Larus and W. Bush. Classy: A Method for Efficiently Compiling Smalltalk. In *Smalltalk on a RISC: Architectural investigations*, pages 186–202. Computer Science Division, University of California, Berkeley, April 1983. proceedings of CS292R.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall International, 1988.

[MJ81] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: theory and applications*. Prentice-Hall Software Series. Prentice-Hall, 1981.

[MSOP86] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented DBMS. *ACM SIGPLAN Notices*, 21(11):472–482, November 1986. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications.

[Nor88] N. D. North. A Framework for the Semantics of Object-oriented Languages. Unpublished manuscript, 1988.

[PS91a] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, 1991.

[PS91b] Jens Palsberg and Michael I. Schwartzbach. Static typing for object-oriented programming. Computer Science Department, Aarhus University, available as technical report DAIMI PB–355, 1991.

[Ren82] T. Rentsch. Object oriented programming. *ACM SIGPLAN Notices*, 17(9):51–57, September 1982.

[Rob86a] Graham Roberts. A report on the current state of the type system. Department of Computer Science, Queen Mary College, London, 1986.

[Rob86b] Graham Roberts. The typechecker in use: Issues raised and problems discovered. Department of Computer Science, Queen Mary College, London, 1986.

[Rob90] Graham Roberts. *Smalltype: a type declaration and type checking system for Smalltalk-80*. PhD thesis, Department of Computer Science, Queen Mary College, London, 1990.

[Ryd83] Barbara G. Ryder. Incremental Data Flow Analysis. In *Proceedings of the Tenth ACM Symposium on the Principles of Programming Languages*, pages 167–176, Austin, Texas, January 1983.

[Sho79] J. F. Shoch. An overview of the programming language Smalltalk-72. *ACM SIGPLAN Notices*, 14(9):64–73, September 1979.

[Smi92] Randall Smith. Object Oriented Programming. Sun Annual Lecture in Computer Science at the University of Manchester, September 1992.

[Sny86] Alan Snyder. CommonObjects: An overview. *ACM SIGPLAN Notices*, 21(10):19–28, October 1986.

[ST84] Norihisa Suzuki and Minoru Terada. Creating Efficient Systems for Object-Oriented Languages. In *Proceedings of the Eleventh ACM Symposium on the Principles of Programming Languages*, pages 290–296, Salt Lake City, Utah, January 1984.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[Suz81] Norihisa Suzuki. Inferring types in Smalltalk. In *Proceedings of the Eighth ACM Symposium on the Principles of Programming Languages*, pages 187–199, Williamsburg, Virginia, 1981.

[Ung87] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM Distinguished Dissertation. MIT Press, 1987.

[US87] David Ungar and Randall Smith. SELF: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–241, December 1987. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications.

[WM80] D. Weinreb and D. Moon. Flavors: Message passing in the Lisp machine. Memo. 602, MIT Lab. for Artificial Intelligence, 1980.

[Wol88] Mario I. Wolczko. Semantics of object-oriented languages. Technical Report UMCS–88–6–1, Department of Computer Science, University of Manchester, May 1988.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. *ACM SIGPLAN Notices*, 21(11):258–268, November 1986. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications.

[Yel87] Phillip M. Yelland. Denotational semantics for reflection in a procedural language. Unpublished manuscript, 1987.