THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Abstract Interpretation and Optimising Transformations for Applicative Programs

Alan Mycroft

Doctor of Philosophy

University of Edinburgh

1981

Abstract

This thesis describes methods for transforming applicative programs with the aim of improving their efficiency. The general justification for these techniques is presented via the concept of abstract interpretation. The work can be seen as providing mechanisms to optimise applicative programs for sequential von Neumann machines. The chapters address the following subjects.

Chapter 1 gives an overview and gentle introduction to the following technical chapters.

Chapter 2 gives an introduction to and motivation for the concept of abstract interpretation necessary for the detailed understanding of the rest of the work. It includes certain theoretical developments, of which I believe the most important is the incorporation of the concept of partial functions into our notion of abstract interpretation. This is done by associating non-standard denotations with functions just as denotational semantics gives the standard denotations.

Chapter 3 gives an example of the ease with which we can talk about function objects within abstract interpretive schemes. It uses this to show how a simple language using call-by-need semantics can be augmented with a system that annotates places in a program at which call-by-value can be used without violating the call-by-need semantics.

Chapter 4 extends the work of chapter 3 by showing that under some sequentiality restriction, the incorporation of call-by-value for call-by-need can be made complete in the sense that the resulting program will only possess strict functions except for the conditional.

Chapter 5 is an attempt to apply the concepts of abstract interpretation to a completely different problem, that of incorporating destructive operators into an applicative program. We do this in order to increase the efficiency of implementation without violating the applicative semantics by introducing destructive operators into our language.

Finally, chapter 6 contains a discussion of the implications of such techniques for real languages, and in particular presents arguments whereby applicative languages should be seen as whole systems and not merely the applicative subset of some larger language.

Table of Contents

Abstract 2
Acknowledgments 6
Chapter 1: Overview 8
1.1 Abstract Interpretation 8
1.2 The theory and practice of transforming call-by-need into call-by-value 10
1.3 Call-by-need = Call-by-value + Conditional 12
1.4 Introduction of destructive operators into applicative programs 14
Chapter 2: Introduction to Abstract Interpretation 17
2.1 Introduction to abstract interpretation 20
2.2 Abstract Interpretation for the Flowchart Idiom 26
2.2.1 Flowchart schema 26
2.2.2 Flowchart semantics 27
2.2.3 Abstract flowchart interpretation 29
2.2.4 Static Semantics 31
2.2.5 Val is naturally a set 32
2.3 Mathematical basis for Abstraction 32
2.3.1 Abstraction of Functions 33
2.3.2 Abstraction for Interpretations 34
2.4 Flowchart Abstract Interpretation 35
2.4.1 Example 36
2.4.2 Lattice of Interpretations 37
2.5 Abstract Interpretation for the Applicative Idiom 37
2.5.1 Recursion equation schema 38
2.5.2 Recursion equation semantics 39
2.5.3 The collecting interpretation 39
2.6 An Approximating Interpretation 42
2.6.1 Formal Definition of Abstraction 44
2.6.2 An alternative view of Conc and Abs 47
2.6.3 The Abstraction of Functions 47
2.6.4 Correctness of an Abstraction 48
2.6.5 Proof of the Correctness Theorem 49
2.7 Some Example Abstract Interpretations 51
2.7.1 Application: The Independent Attribute Formulation 51
2.7.2 Application: Abstracting Termination 52
2.7.3 Application: Power Sets are an Abstraction of Power Domains 54
2.8 The Lattice of Interpretations 55
2.9 Notes on the Abstraction Relation 58
2.10 Deducing Properties of Applicative Programs 60
Chapter 3: The Theory and Practice of Transforming Call-by-need into Call-by-value 62
3.1 Abstract 62
3.2 Motivation 63
3.3 Formalism 68
3.4 Notation and Definitions 69
3.5 Example 70
3.6 Pragmatics 77
3.7 Transforming programs to use call-by-value 79
3.8 Non-discrete domains 81
3.9 Discussion of runtime errors in applicative languages. 82
Chapter 4: Call-by-Need = Call-by-Value + Conditional 86
4.1 Introduction 86

4.2 Overview 86
4.3 Basic Definitions 87
4.3.1 Conventions 89
4.3.2 Operational Semantics 90
4.3.3 Definition of Sequentiality 90
4.4 Method 92
4.4.1 Total reduction to call-by-value 93
4.4.2 Production of the oracles F_I^* 94
4.4.3 The $F_I^\#$ exhibited 94
4.4.4 Operational extension of a_i 95
4.4.5 Denotational extension of a_i 96
4.4.6 Definition of F_I^* 98
4.4.7 Definition of the system oracles A_I^* 98
4.4.8 Proof of correctness 100
4.5 Getting rid of the $\perp_i$ 102
4.5.1 Proof of correctness of the overloading 105
4.6 Example 106
4.7 Computational Costs 107
4.8 Conclusions 109
Chapter 5: On Introducing Destructive Operators into Applicative Programs 111
5.1 Abstract 111
5.2 Developments from previous work 112
5.3 General Overview of the Development 113
5.4 Formalism and General Ideas 114
5.4.1 Does CONS have a side effect? 117
5.4.2 Destructive Operators 118
5.4.2.1 Simple Example 118
5.4.2.2 The choice of destructive operators 119
5.4.3 Order of evaluation 121
5.4.4 Occurrences and other Basic Ideas 123
5.5 The Extent of Possible Use of Arguments 126
5.5.1 Correctness of the Uses and Examines Interpretations 131
5.6 Usage counts 132
5.7 Isolation classes: abstract interpretations modelling usage counts 134
5.7.1 The Isolation Ordering 136
5.7.2 Isolation properties of functions 138
5.7.3 The problems of variables 140
5.7.4 The treatment of variables 141
5.7.5 The details of variable sharing 142
5.7.6 Irrelevant paths 147
5.8 Useful Transformations 148
5.8.1 Transformations to insert FREE 148
5.8.2 Transformations for IF 149
5.8.3 Replacing FREE with RPLACA/D 151
5.9 Correctness with respect to the semantics 151
5.9.1 Proof of correctness of 'real' FREE 153
5.10 Producing destructive versions of user functions 155
5.11 Worked example: Derivation of NCONC from APPEND 158
5.11.1 Producing more efficient versions of SUBST 161
5.12 Syntax and Semantics of LISP-D 164
5.12.1 Notation 164
5.12.1.1 Data Classes 164
5.12.1.2 Syntactic equations 165

5.12.1.3 Semantic Functions 165
5.12.2 Semantic Equations 165
5.12.2.1 Semantic modifications to add usage counts 169
5.12.3 A Store-less semantics for LISP-A 169
Chapter 6: Conclusions 171
6.1 Efficiency in Applicative Languages 171
6.2 Suggestions for Further Work 173
References 175

Acknowledgments

I would like to thank the following people and authorities, each of whom have contributed something to this work, either directly, or by their inspirational presence.

Firstly I must thank my supervisors Rod Burstall and Robin Milner for being so kind and tolerant of a somewhat independently spirited supervisee. Their influence, together with the research setting of Edinburgh University, have contributed much to the style and content of this thesis. I must thank both the Department of Artificial Intelligence, and the Department of Computer Science for providing such a congenial atmosphere for research.

Many of the members, past and present, of these departments are responsible, via their comments, for the detailed presentation of this work. In particular I would thank Martin Feather, Dave MacQueen and Chris Mellish at Artificial Intelligence, and also Luca Cardelli, Mike Gordon, Matthew Hennessy, Gordon Plotkin, David Rydeheard, Don Sannella and Chris Wadsworth at Computer Science.

Thanks must also be given to John Darlington of Imperial College, London, who provided both supervision and ideas during the time I spent in London.

I would also like to thank the Algebra Group at the Computer Laboratory, Cambridge University for their part in this work. They were always willing to discuss my work and particular thanks are due to Arthur Norman (who supervised me for my previous Diploma in Computer Science) for a suggestion which led to the work presented

in chapter 3.

I am grateful for the close relationship between Edinburgh University and INRIA (Institut National de Recherche en Informatique et en Automatique) which permitted me to talk to Gerard Huet and Jean-Jacques Levy.

Also deserving of thanks is the Datalogisk Afdeling of Aarhus University, which provided an especially valuable workshop on data-flow analysis, and the participants in that workshop especially Neil Jones (the organiser), Patrick and Radhia Cousot, and Flemming Nielson.

The SRC provided finance for this research via studentship B/78/313640.

Finally, I, and this work, owe a great deal to the personal support of my wife, Hilary, whose cheerfulness kept me going through the darker moments of research.

Chapter 1: Overview

This chapter gives a detailed, but non-technical introduction to the remainder of this thesis. The sections discuss the work presented in corresponding chapters.

The work presented in this thesis attempts to strike reasonable balance between theory and practice. This is a difficult aim, and achieved in few works. Clearly it is extremely hard to satisfy both the requirement for rigour and that for applicability. Here we attempt to do so by developing the idea of 'abstract interpretation' in both theoretical and practical directions, but inevitably the more practical aspects must lack rigour and the more theoretical ones seem remote from practice. We hope to convince the reader that this work exhibits practical uses of abstract interpretation for analysing applicative programs, which are well founded in theory. We would also hope that the theoretical developments are considered relevant.

1.1 Abstract Interpretation

This chapter performs two rôles. Firstly it is an introduction to abstract interpretation in its own right, and secondly it exhibits the changes to the theory that are necessary to enable us to discuss applicative languages, thereby giving a theoretical basis for the remaining chapters.

Essentially, we will follow the work of Cousot & Cousot [9] which itself is based on ideas as old as Sintzoff [47] and Naur [36], but which is also found in the 'rule of signs' given below.

Suppose we have a complicated system, for example the set of integers under addition (+) and multiplication (*), and we want to know certain properties about a computation in such a system, for example the sign of the result of (-345)*1067. Then we can either compute the result naively and take its sign, or employ the rule of signs (neg)*(pos)=(neg) to deduce that the result must be negative. Similarly this type of reasoning can show that a^2+b^2 is never negative, however consideration of $a^2+b^2-2*a*b$, shows us that all we can say is that the result is either pos, neg or zero, which we knew already.

Clearly the price paid for calculating in such a simple domain {pos,neg,zero} is that our answers to questions about the more complicated integer domain can never be exact, however we will choose our interpretations in such a manner that they imply results about corresponding calculations in the standard interpretation of symbols, as in the above cases.

Computationally, we are not interested in abstract interpretation as an alternative method of performing such simple calculations, but rather as a tool for showing that certain conditions will hold about a certain function or program point at execution time. For example, as in the work in the next section, it will enable us to show that a certain function cannot have a defined result unless a certain parameter is defined and hence that that parameter can be evaluated out of the standard order.

1.2 The theory and practice of transforming call-by-need into call-by-value

In this chapter, which has also been published as a paper of the same name [34], we consider the question of transforming a program written in a call-by-need language to one which uses call-by-value for as many of the parameter passing instances as is consistent with the call-by-need semantics. We provide some motivation for why this is a desirable thing to do, for example call-by-need is a more natural semantics for applicative languages, whereas call-by-value produces much more efficient code in situations where the two regimes are equivalent.

In the next two paragraphs the assumed mode of parameter passing will be call-by-need. We note that there are two situations where a parameter can be passed to a function, F, say, using call-by-value without disturbing the call-by-need semantics.

Firstly suppose a certain function, F, say, has the property that it always evaluates its k'th formal parameter (an operational view), or in comparable denotational terms

$$F(\ldots \perp \ldots) = \perp$$

(where $\perp$, the undefined value, occurs in the k'th position) regardless of the values of the remaining parameters. Then it is clear that we cannot affect the termination properties of F by evaluating its k'th actual parameter prior to any call.

Alternatively suppose we can show, for some actual parameter, e, say, that e has the property that, regardless of the environment in which it is evaluated, its evaluation always terminates (for

example e is of the form x+1 where we can show that the variable x is never $\perp$, as would be the case if it had been passed by value). Then we can safely (without disturbing the call-by-need semantics) evaluate this parameter before passing it to the called function, saving the possibly greater expense of constructing and evaluating a closure. Of course, if (in a particular program) all actual parameters corresponding to a particular formal parameter of a certain function have this form, we can change the function to expect a value parameter which will enhance the efficiency further.

These views must be tempered a little by the question of what equivalence means, as for example, if a program has two distinct possible run time errors that it may become ensnared upon, then such transformations as given in the preceding two paragraphs may cause a transformed program to give a different error message. Accordingly the problem of errors in applicative languages is discussed in some detail, but if we adopt the notion of error values rather than error jumpouts then the equivalence hinted at above holds. The statement of equivalence would be that the call-by-need program annotated with hints that certain parameters could be evaluated according to call-by-value would be strongly equivalent to its purely call-by-need ancestor.

The technique for propagating information about which parameters have certain termination properties is based on abstract interpretation [9] and two alternative evaluation functions $E^{\#}$ and $E^{\flat}$ are constructed by reference to the standard evaluation function E, and shown to have the properties that they reflect the behaviour

of E by giving safe lower and upper bounds on the definedness of expressions.

The fixpoint structure of the meaning of functions within these alternative interpretations is also discussed, and a section is devoted to the problem of transforming an applicative program to maximise the number of parameters susceptible to our methods. A test of the system on a 1100 line applicative program (written without knowledge of the system) showed a promising 'hit rate' of around 75%.

Since the work was completed, it has been extended by Jones [27] to be applicable to the whole lambda calculus, rather than the minimal system of recursion equations which were used here, however the extension necessarily requires a more complicated setup than the simple one adopted in this work, and hence should not be regarded as supplanting it.

1.3 Call-by-need = Call-by-value + Conditional

This chapter extends the work described in the previous chapter by showing that the idea of transforming the program to increase the number of places call-by-value can be used is complete, in the sense that a given applicative call-by-need program can be simply transformed (not interpreted) into an equivalent program for a language which has strict (call-by-value) semantics for all functions except the distinguished conditional function.

We need slight (sequentiality) restrictions on our system functions as compared to the work of the previous chapter, but

otherwise the two works address identical call-by-need languages. The interest of this work is both theoretical and practical, in that it provides an alternative formulation of call-by-need in terms of call-by-value and also in that it provides a practical alternative to thunks (or closures) for an implementation of a call-by-need language. Plotkin [40] uses the concept of closures to show the equivalence of call-by-name and call-by-value interpreters for the lambda-calculus by showing we can model call-by-name within a call-by-value interpreter. He does this by the use of lambda "buffers" which are forced to be evaluated when required.

Our transformation consists of four stages. Firstly we show that a given call-by-need program is strongly equivalent to one of the possible execution paths of a non-deterministic interpreter. This equivalence was also given by de Roever [43].

Secondly we show that it is possible to define oracles for this system of non-deterministic equations, which predict the path the computation will follow, and enable us to derive a result without using a parallel interpreter. However the oracles rely on extending the domain of discourse to include certain 'squib'-like elements to trace risky computations.

Thirdly we show that it is possible to map the extra elements added in the previous paragraph into our standard domain of discourse, by using a form of overloading. This is really necessary, both for practical and theoretical reasons since the run

time tests of domain membership are likely to be as expensive as testing whether a parameter is a closure or an evaluated closure, and also we would be open to the criticism that we are not really comparing like with like.

Finally it is necessary to discuss how expensive this process can be, and we will show that whilst it can increase the size of the program by an exponential factor of its complexity, the new program has a running time linearly related to the original. Figures (for the program cited in the previous section) suggest that the expense is rather less in practice than these worst case estimates, and costs less than a factor of two, both in time and space complexity. Moreover, due to the smaller cost (in both time and space) of call-by-value operations compared with equivalent call-by-need ones as implied by current machine architectures, the transformed program may actually be faster and smaller.

1.4 Introduction of destructive operators into applicative programs

Now we turn our mind to a rather different aspect of optimising applicative programs, and consider their implementation in terms of structure creation and destruction. In a purely applicative language our objects of discourse are merely values and correspondingly the concepts of location and reference do not enter the semantics. However an implementation will in general need to introduce these concepts in order to perform the management of (the finite amount of) store in a real machine. Having implemented a store allocation scheme of some sort, we will find that it is necessary to include some form of store de-allocator and also that

data objects must share their store with each other in order to fit the computation into a real machine.

Store de-allocation is often performed by some form of garbage collection (a separate process which collects all data objects which cannot influence the future computation and returns them to free store). However garbage collection can be quite expensive, and as machines become larger takes longer.

Here we seek to reduce the overhead imposed by garbage collection by determining some of the points in a program where a structure is used for the last time before losing its last reference. There has been other work done on this area but mainly for languages without procedures, a good account being given in [28]. However Schwarz [45, 46] and Pettorossi [38, 39] have considered the problem for applicative languages. Pettorossi's work addresses the situation where structures do not share store, unlike a real implementation. Schwarz's work does consider the problem of sharing, however he uses an operational model of a term re-writing system, which is not close enough to the standard semantics to enable simple proofs to be constructed, and also suffers from the deficiency that the user must declare the possibilities for destroying objects along with the program. Here we form a more denotational model which enables us to exhibit, as fixpoints, the amount of sharing present in certain structures.

The technique is to construct an alternative interpretation for the program, which we use instead of the standard semantics. The

objects in this alternative interpretation are <u>isolation</u> <u>classes</u>, which model the sharing in the original language. The terminology is borrowed from Schwarz.

For simplicity the theory is developed for a single constructor function (CONS), however this can be simply extended. On the other hand we deviate strongly from standard practice and make the important choice of only permitting a single destructive operator, which we will call FREE. The intention is that FREE will return its CONS node argument back onto the free list for re-allocation, and thus, in a LISP-like syntax we can see that RPLACA and RPLACD (and hence all destructive operators) can be written in terms of FREE. For example

```
(RPLACD X Y)  =  (DCONS X (CAR X) Y)
```

where

```
(DCONS X Y Z)  =  (PROG2 (FREE X) (CONS Y Z))
```

Unfortunately, this type of definition is not very amenable to proof, and therefore, instead of adopting a direct approach, we choose a two stage construction whereby we define FREE in the semantics to merely mark its CONS node argument, so that it produces a run-time error on further reference. This enables us to insert FREE's freely, subject to the restriction that we must be able to show that the resultant program cannot actually produce such a run time error. Then we show that the two versions of FREE produce the same results for any original program from which they are both derived.

Chapter 2: Introduction to Abstract Interpretation

The purpose of this chapter is twofold, firstly it introduces the general concept of abstract, or non-standard, interpretation. This was applied to computation by Sintzoff [47] (although Naur used a special case for type checking [36]) and greatly developed by Cousot & Cousot [8, 9, 10]. Wegbreit [52] seems to have been the first to use a lattice theoretic model for the objects in our abstract domain. Having said this, we should note that the idea of abstract interpretation as manifested in the 'rule of signs' discussed below pre-dates computation.

Equally importantly, this chapter extends and re-expresses many of these ideas in forms more suitable for applicative languages, rather that the usual flowchart idiom. In particular it forms a technical basis for the following chapters. However much of the formal development needs considerable mathematical skill not required for the applications presented in later chapters. It is recommended that this chapter is omitted from section 2.2 on first reading.

The standard work on abstract interpretation is either operationally based [8, 9, 10, 27] or denotationally based but suffering from the drawback of being unable to express the concept of recursive functions [11, 37]. In either case flow analytic methods fail to build a natural strong theory including partial functions. Here we use the concept of power domain, rather than that of power set used in the above works, to build a theory which naturally considers partial functions and termination. We indicate

how the standard power set based theory can be seen as an abstraction (in the usual sense due to Cousot) of our theory based on power domains. Donzeau-Gouge [11] and Nielson [37] both use continuation semantics [48] for their model which enables them to use similar structures to Cousot. It could be argued that for applicative languages we should merely choose that subset of continuation semantics that is required, however we will defend the opposite point of view, that an applicative style of semantics is required. The main reason for this is that we would like a direct semantics which follows the natural applicative semantics as closely as possible. This will enable simple and general proof rules to be derived for the correctness of any interpretations we may care to develop, rather than a more distant semantics which inhibits correctness proofs.

This chapter is structured in the following manner. Firstly we undertake a review of flowchart programs, and a particular non-standard interpretation called the collecting interpretation. Next we present some lattice mathematics which will be used in the following section to derive more abstract interpretations than the collecting interpretation. In passing we note that the question of domain structure does not really arise in a flowchart setting since all our basic flowchart operations are total. This corresponds to the ability to use power sets for our model of abstract interpretation.

In section 2.5 we review the idea of a recursive program scheme, Σ, together with a standard interpretation. This is followed by

showing that there is a canonical induced interpretation (the collecting interpretation) on a power domain which represents computations in (the standard interpretation of) Σ using sets of values. Sections 2.6.1 and 2.6.3 develop the idea of an abstraction of the collecting interpretation which calculates (a representation of) a superset of the values which would be computed in the collecting interpretation. As in the flowchart scheme presented above this abstracted domain will be simpler to compute in. We give several examples of the use of such interpretations, including the representation of the Cousot collecting interpretation as an abstraction of our collecting interpretation. Finally we will examine some ideas for extending the relation of abstraction to all pairs of abstract interpretations, rather than the use above which just compares abstract interpretations with the standard collecting interpretation. This will enable us to build a lattice of interpretations as developed by the Cousots for flowchart programs.

One interestingly intermediate work is that of Jones [27] in which he applies the notions of abstract interpretation to the lambda-calculus. However he does this by showing that a lambda-calculus program can be considered to possess program points by virtue of noting that the pieces of code handled by the interpreter are all either substructures of the original code for the program or substructures of the result of a base (system) function or a constant. Essentially he models the states that would be processed by the mathematical interpreter presented by Plotkin [40]. This

enables him to discuss questions of termination within a power set based model. However the disadvantages of such an approach are similar to the objections to the use of continuation semantics given above - namely that our model is removed from the natural semantics.

2.1 Introduction to abstract interpretation

A (standard) simple example is the most useful way of setting the scene. Let us suppose that we need a certain amount of information about the result of the calculation

(-357) * 1078

in order to optimise the details of performing the calculation, for example if multiplication (*) were an expensive operation. The classical 'rule of signs'

(-) * (+) = (-)

can be used to infer that the result of the above calculation is negative, without the need to perform a (possibly expensive) multiplication, but rather by performing a calculation in a simpler domain. This is not the only possible abstract calculation we can perform, for example we can deduce that the result is even by performing

ODD * EVEN = EVEN

or even show that the magnitude of the result is between 10^5 and 10^7 by using the calculation

(3 digit number) * (4 digit number) = (6 or 7 digit number).

The work of Patrick and Radhia Cousot, which will be discussed in more detail later, shows that the set of possible interpretations forms a partial order under a certain relation

called abstraction and a suitable restriction on the elements of this set gives us a lattice structure.

Readers familiar with the slide rule, and the more astute users of calculators, will recognise the above calculations as typical 'checking' calculation performed by the user in order to verify the actual calculations. (Or to choose decimal points for slide rule calculations.) In this work however, we do not consider these non-standard calculations for the purpose of checking other calculations, but rather for the selection of efficient evaluation mechanisms for our real calculation.

To formalise the above rule of signs example we will define our domains (for the purposes of arithmetic these will be sets, but for computation complete partial orders (cpo's) will be required). We will consider

$$*: \text{Int} \times \text{Int} \rightarrow \text{Int}$$

to be the standard interpretation of multiplication on integers. Now we may introduce a new set

$$\text{Sign} = \{(+), (-)\}$$

together with an operation

$$\circledast: \text{Sign} \times \text{Sign} \rightarrow \text{Sign}$$

defined by

$$a \circledast b = (+) \quad \text{if } a = b$$
$$a \circledast b = (-) \quad \text{otherwise}$$

Since no misunderstanding can arise it is common to write the symbol '*' in both domains. This can be regarded (computationally) as overloading '*' or (mathematically) as providing a new interpretation or semantics for '*'. However we will distinguish

'*' and '⊕' for the remainder of this section.

Clearly this new interpretation of symbols is of no use unless we can relate the effects of '*' and '⊕' in some manner. We will do this by defining functions

$$\text{Abs: Int}-\{0\} \rightarrow \text{Sign} \qquad \text{(Abstraction)}$$

$$\text{Conc: Sign} \rightarrow 2^{\text{Int}} \qquad \text{(Concretisation)}$$

with definitions

$$\begin{aligned} \text{Abs}(i) &= (+) \quad \text{if} \quad i > 0 \\ &= (-) \quad \text{if} \quad i < 0 \end{aligned}$$

and

$$\text{Conc}(p) = \{i \in \text{Int}: \text{Abs}(i) = p\}.$$

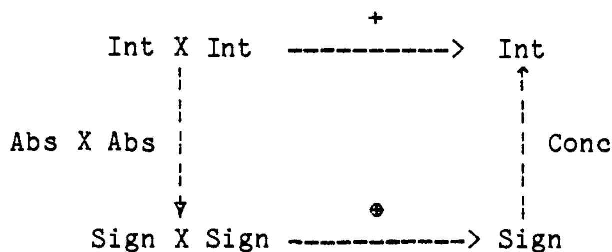
(We will omit the 0 element of Int from the discussion temporarily and discuss it later - it is associated with neither (+) nor (-) as far as the rule of signs is concerned.) This enables us to derive

$$\text{Abs}(a * b) = \text{Abs}(a) \oplus \text{Abs}(b)$$

and hence

$$a * b \in \text{Conc}(\text{Abs}(a) \oplus \text{Abs}(b))$$

This is most easily visualised as



The reader should keep this form of picture in mind as we develop the following details.

However, in general such a simple model is insufficient for the reason that begins to show itself in the "number of digits" example above in that we cannot satisfactorily take as our abstract

universe Num_of_Digits since the interpretation of '*' cannot be treated as a map

$\circledast$: Num_of_Digits X Num_of_digits $\rightarrow$ Num_of_Digits

since the example above shows that

3 $\circledast$ 4 = 6 <u>or</u> 7

and so the abstract multiplication operation cannot be <u>closed</u>. Hence, in this case it would be necessary to choose $2^{\text{Num_of_Digits}}$ or some similar set to model the abstract domain. As a similar example, if we extend our signs universe

{(+), (-)}

to cover addition we would derive

(+) + (-) = ($\pm$) = (unknown sign).

Again we see the desirability of using $2^{\{(+),(-)\}}$ in that it allows us to include such concepts. We can now re-introduce the 0 element of Int and treat Abs(0) as ($\pm$), or for more accuracy in our abstract computations at the expense of a more complicated domain we could change our Sign set to be {(+),(-),(0)} and use the absorbtive properties of 0 under * to define

Abs(0) = (0)

x $\circledast$ (0) = (0) $\circledast$ x = (0)

in our abstract domain. Computationally the inclusion of {}, the empty set, in $2^{\{(+),(-)\}}$ is often a good thing since it naturally corresponds to "No possible associated concrete values" such as would be formed after an unavoidable error or a non-referenced variable.

While we are extending our abstraction process in such a manner it is desirable to remove the annoyance of Conc(Abs(x)) not having the same 'type' as x (it is a set type) and change the definition

of Abs to account for this deriving

$$\text{Aval} = \{(+), (-), (\pm)\}$$
$$\text{Abs: } 2^{\text{Int}} \rightarrow \text{AVal}$$
$$\text{Conc: AVal} \rightarrow 2^{\text{Int}}$$

defined (for non-empty sets) by

$$\begin{aligned} \text{Abs}(S) &= (+) \text{ if } \text{Abs}'(s) = (+) \;\; \forall \text{ s in S} \\ &= (-) \text{ if } \text{Abs}'(s) = (-) \;\; \forall \text{ s in S} \\ &= (\pm) \text{ otherwise} \end{aligned}$$

and

$$\text{Conc}(A) = \bigcup \{S: \text{Abs}(S) = A\}$$

where Abs' is the old version of Abs defined previously.

This idea is reasonable when we are merely considering a single abstract interpretation, though we will later want to discuss classes of interpretations and their relationship to one another. Therefore we will adopt a slightly different strategy and define a canonical abstract interpretation, called the 'collecting' interpretation, which models the concrete interpretation with no loss of information (the two interpretations determine one another). The collecting interpretation will have abstract value domain (AVal) the power set (or power domain in our later work) of the concrete value domain (CVal) and the basic operations defined as the set extension of functions, for example for given

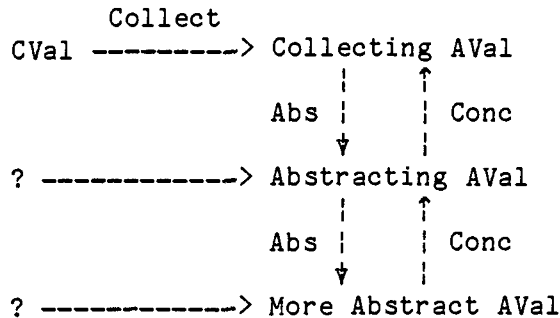
$$\text{f: CVal X CVal} \rightarrow \text{CVal}$$

we define

$$\text{f': AVal X AVal} \rightarrow \text{AVal}$$
$$\text{f'}(X,Y) = \{f(x,y): x \in X, y \in Y\}$$

The collecting interpretation will represent the top element of a lattice (ordered by a relation called abstraction) of

interpretations modelling the standard interpretation as in the above. The maps between abstract interpretations can then be simple maps not involving power sets. A suitable diagram is:



The ?'s are given in this diagram to illustrate the fact that abstract interpretations may, or may not, be collecting interpretations for some other standard interpretation (AVal's corresponding to collecting interpretations can only have certain specific numbers of elements). One other feature of the above representation is that is permits a more symmetric notation which removes the constant reference to power sets.

We choose not to force AVal to be a power set (as was used in the examples above) since this reduces the generality. As an example of why we may not desire AVal to be a full power set, let us return to the Num_of_Digits example above and observe (at least if we just use it for repeated multiplication) that any abstract value can be considered to be an interval

$$[a,b]$$

with a and b numbers representing the upper and lower bounds of number of digits in the result. This would suffice to ensure that the abstract multiplication operation is closed. It would be given by an interval arithmetic operation:

$$[a,b] \circledast [c,d] \;=\; [a+c-1,\ b+d]$$

2.2 Abstract Interpretation for the Flowchart Idiom

It is recommended that the remainder of this chapter is skipped on first reading in order to become acquainted with the applications that motivate the theory.

This section follows the development presented by the Cousots in [8, 9] in which the program under analysis is written in flowchart style. Therefore the interesting compile time problems are those based on the possible sets of values associated with particular variables (and possibly their interrelation) at a particular program point.

2.2.1 Flowchart schema

Firstly, we will define the syntax of a simple flowchart language. A flowchart program, P, is a labelled directed graph, (Node,Arc) where Arc is a subset of Node X Node giving the edges, together with a labelling of Node with statements. Arcs (in Arc) will also be called program points and referred to by ⟦label:⟧. We define functions

$$\text{Pred, Succ: Node} \rightarrow 2^{\text{Arc}}$$

by

$$\text{Pred}(n) = \{(a,n) \in \text{Arc}\}$$
$$\text{Succ}(n) = \{(n,a) \in \text{Arc}\}$$

We denote the cardinality of a set S by Card(S). We also assume the existence of syntactic categories Var of variables and Exp of expressions.

The possible statements, and the corresponding restrictions on the nodes they can label are:

An entry node: There is only one of these (called Entry). It has

Card(Pred(Entry)) = 0 and Card(Succ(Entry)) = 1. We will define the program point Start by

{Start} = Succ(Entry)

An exit node: This has Succ(n) = {Stop} = {(Exit,Exit)}. Again, without loss of generality we can assume there is just one of these, called Exit.

An assignment node:

These have Card(Succ(n)) = 1. Their label is of the form

⟦Var := Exp⟧.

A test node: These have Card(Succ(n)) = 2. They have a test part in Exp, and Succ(n) is labelled as: a true branch SuccT(n), and a false branch SuccF(n).

Because the values of variables change at nodes, it is only generally sensible to talk about the value of a variable at program points.

2.2.2 Flowchart semantics

We will assume the existence of a set Val, of values, including an uninitialised value '?'. There is no point in using the conventional lattice structure for Val, since undefined values can only occur due to the program looping, and Val is not a suitable place to put them. (See section 2.2.5.) We will also require a concept of environments:

Env = Var $\rightarrow$ Val.

Thus giving a variable a value at a program assignment node means the environments at the program points around that node differ on the assigned variable. Similarly will we assume the existence of an evaluation function for elements of Exp which occur in tests and assignments:

Eval: Exp $\rightarrow$ Env $\rightarrow$ Val.

A program state, then, is given by a pair

State = Arc X Env

describing the program point and current environment. We now define a function

NState: State → State

which describes the one step state transformation function. The definitions of the flowchart state transformations are quite simple:

For an assignment node, n, say,

⟦l: x := e; m:⟧

we define the transformer to be

```
NState(l,r)  =  (m,r')
where r'⟦x⟧  =  Eval⟦e⟧(r)
      r'⟦y⟧  =  r⟦y⟧ for all y ≠ x
```

This states that the environment r' on the arc, m, leading from n is the same as the environment r on the arc, l, leading to n except for the variable x, which takes the value of the expression e, evaluated in the environment r.

Similarly for a choice node

⟦a: <u>if</u> e <u>then</u> <u>goto</u> l <u>else</u> <u>goto</u> m⟧

we derive

```
NState(a,r) = (l,r)  if  Eval⟦e⟧(r) = true
            = (m,r)  if  Eval⟦e⟧(r) = false
```

For the Exit node, we can define

Nstate(Stop,r) = (Stop,r)

and note that the Entry node action has already been defined as we will start the program at Start where

{Start} = Succ(Entry).

Now we need to specify the initial (empty) environment present at the entry node:

$$\text{InitEnv} = \lambda\ \text{var}.\ ?$$

We choose here to avoid the convention of using $\perp$ for the value of an uninitialised variable for two reasons. Firstly the notion of error we get by referring to a variable without a value is distinct from the idea of a program looping, and secondly later formalism will require $\perp$ to be treated carefully (see section 2.2.5).

These definitions of NState give a standard operational style flowchart semantics with an initial state

$$\text{InitState} = (\text{Start},\text{InitEnv})$$

Thus, if the limit of

$$\text{NState}^n(\text{InitState}) \text{ as } n \rightarrow \infty$$

exists and is of the form

$$(\text{Stop},\text{AnswerEnv})$$

then AnswerEnv gives the final values of the variables after executing P. If the limit does not exist or has a Arc component not Stop then the program loops forever.

2.2.3 Abstract flowchart interpretation

We will now exhibit a canonical abstract interpretation (called the 'collecting' interpretation) associated with this standard interpretation.

The possible contexts in our program are given by

$$\text{Context} = 2^{\text{Env}}$$

and will model the set of environments which will occur at a

program point during execution. The <u>context vectors</u> are given by

ContextVector = Arc $\rightarrow$ 2^{Env}

in other words, given a context vector cv then, for each program point, p, cv(p) gives the set of possible environments which can exist at p.

Now it is necessary to define a method to form the context vector corresponding to the set of run time environments. We will do this by defining

InitContext = λ arc. arc=Start $\rightarrow$ {InitEnv}, {}

which gives the initial context vector associated with starting at Start, and a fixpoint iteration whose limit is the desired context. (This can also be seen as stating that the desired context is the least fixpoint of a certain equation.) The previous section gives a method (NState) by which, given an environment before the evaluation of a particular node, we are able to derive a corresponding environment which would exist after the execution of the code at that node. We must now 'lift' this idea from a map

NState: State $\rightarrow$ State

to a map

NContext: ContextVector $\rightarrow$ ContextVector

for our fixpoint formulation. NContext will correspond to our NState function which gives the next state from a given state, but instead will show how a context vector is affected by 'one step' execution. We define

NContext(cv) = λ arc. cv(arc) $\cup$
{env: (arc,env)=NState(a,e), e $\in$ cv(a), a $\in$ Arc}

Since 2^{Env} has a natural subset ordering and NContext is

continuous with respect to this ordering we can form the limit

$$\text{LimCV} = \text{NContext}^n(\text{InitContext}) \text{ as } n \rightarrow \infty$$

which exists, and gives the exact set of environments associated with each program point during the standard computation of P. This explains the name of 'collecting' interpretation. Note the close relationship between the two interpretations: the set given by

LimCV(Stop)

is empty if the standard computation is non-terminating, and otherwise is the singleton set {AnswerEnv} as defined above.

2.2.4 Static Semantics

Taking the idea of the collecting interpretation further, we can now consider running a program on a set of input values, rather than defining a single program execution as above. We will do this by considering that the program, P, can be started (at Start) with any one of a given set of initial variable binding environments. (Alternatives are providing a 'read' statement and allowing more than one Entry node.) To do this let InitEnvSet be a set of elements of Env and consider

$$\text{LimCVSet} = \{\lim \text{NContext}^n(cv) : cv \in \text{InitCVSet}\}$$

which gives the set of possible final context vectors, where

$$\text{InitCVSet} = \{(\lambda \text{ arc. arc=Start} \rightarrow \{e\},\{\}) : e \in \text{InitEnvSet}\}$$

We can now propagate the 'set-ness' of LimCVSet to Context (which is a set of environments) by defining

$$\text{LimStaticCV: ContextVector } (= \text{Arc} \rightarrow 2^{\text{Env}})$$

$$\text{LimStaticCV}(p) = \bigcup \{f(p) : f \in \text{LimCVSet}\}.$$

This interpretation is called the static semantic interpretation of P and generalises the collecting interpretation (one can

retrieve the collecting interpretation by merely restricting InitCVSet to singleton sets).

The static semantic interpretation is very useful because it has the property that all other semantic interpretations can be considered as abstractions of it. See [9] (where it is called I_{SS}) and section 2.3.2.

2.2.5 Val is naturally a set

Note that this exposition, which corresponds to the Cousots work, never uses the ordering of Val. Their work does, but as a mechanism to allow the standard semantic ideas to be simply used, rather than in an essential manner. To us Val is just a set (including an error element considered to be incomparable with other elements of Val). Note that defining a partial order on Val (and hence on dependent concepts like Env) would make difficulties in deciding what we mean by 2^{Env}. It is not obvious which ordering on subsets should be used. This problem clearly has to be tackled in a more direct manner for applicative languages where the concept of partial function is central (see section 2.5).

2.3 Mathematical basis for Abstraction

In this section we will follow the Cousots and introduce the concepts of abstraction and concretisation functions between two general lattices[1], although in general these will just be subsets of a power set. Nielson [37] examines the reasons behind the choice of, and the possibilities for weakening, the following definitions in much greater detail.

---

[1]Here the term "lattice" will mean complete lattice

Let L and M be lattices, then we will define Abs and Conc to be <u>adjoined</u> if they satisfy the conditions that

$Abs: L \rightarrow M$

$Conc: M \rightarrow L$

Abs and Conc are monotonic

$Abs(l) \sqsubseteq m \quad \Leftrightarrow \quad l \sqsubseteq Conc(m)$

These conditions ensure that L and M in some sense model one another. The notion of adjoinedness is essentially the same as that of a 'Galois connection' used in classical lattice theory (see [1]). In such circumstances Abs and Conc determine each other and thus only one need be specified. Explicitly

$$Conc(m) = \bigsqcup \{l: Abs(l) \sqsubseteq m\}$$

$$Abs(l) = \bigsqcap \{m: l \sqsubseteq Conc(m)\}$$

(see Nielson [37]).

Further we will say that Abs and Conc are <u>exactly</u> <u>adjoined</u> if the final condition for adjoined is strengthened to

$$Conc(Abs(x)) \sqsupseteq x$$

$$Abs(Conc(x)) = x$$

The purpose of this is to ensure that M does not contain redundant elements and much of the theory goes through without it. However with exactness we can view Conc o Abs as an upper closure operator on L. It also enables us to write

$$Conc(m) = \bigsqcup \{l: Abs(l) = m\}.$$

2.3.1 Abstraction of Functions

Let L_1 and L_2 be two lattices, and M_1 and M_2 be lattices abstracting L_1 and L_2 respectively via functions Abs_i and $Conc_i$ as above. Now consider a function $g: L \rightarrow L$ and a function $h: M \rightarrow M$ which we will want to consider as abstracting g. (There is an

algebra theoretic view of all this which will be given in the corresponding sections when our more general power domain based theory is developed.)

Just as we said Conc and Abs are exactly adjoined if Conc o Abs $\sqsupseteq$ id we will require that computations in the M_i have a similar property relative to computations in the L_i. We will say that h *abstracts* g (more properly (h,M_1,M_2) abstracts (g,L_1,L_2) with respect to the Abs_i and $Conc_i$) if we have

$$Conc_2(h(Abs_1(x))) \sqsupseteq g(x)$$

or

$$Conc_2 \circ h \circ Abs_1 \sqsupseteq g.$$

Given g we can always find such an h, for example, take h: $M_1 \rightarrow M_2$ defined by h(x) = T where T is the top element of M_2. However such a definition will not tell us a great deal about the computation we are modelling and as such it is worth noting that there exists a 'best' (in the sense of preserving most information) abstraction function defined by

$$h = Abs_2 \circ g \circ Conc_1$$

2.3.2 Abstraction for Interpretations

The abstraction relation given above for functions may be shown to be preserved by both composition and by taking of least fixed points. (For proof adapt the more general proof given in section 2.6.5 by taking $\sqsubseteq = \subset$ and $\sqcup = \cup$. A direct proof is fairly simple.) This gives a general basis and justification for performing computation in an abstract domain and inferring results about a real computation.

2.4 Flowchart Abstract Interpretation

We will now apply the mathematical model of adjoined functions on lattices and functions respecting them to the problem of constructing abstract interpretations for our flowchart scheme.

We have defined Context to be the set of possible environments at a given program point in our flowchart schema. For the purpose of determining an approximation to the set of states which can exist at any given program point we can now follow the Cousots' idea and use the mathematical model of adjoined functions given above.

At any given program point we have a lattice of possible environments, namely Context ordered by set inclusion. Now suppose that we have another lattice AbsCtxt which is an abstraction of Context. Then we can define AbsCtxtVector corresponding to ContextVector above by

AbsCtxtVector = Arc → AbsCtxt.

The lattice structures of Context and AbsCtxt carry across to ContextVector and AbsCtxtVector in the standard ordering of functions by their images. Now the NContext function maps ContextVector onto itself and, by the general theory above, has a corresponding AbsNCtxt which maps AbsCtxtVector onto itself and is an abstraction of NContext. Because of the properties of abstraction any computation carried out in ContextVector (such as computing the collecting interpretation result) can be modelled by a corresponding calculation in AbsCtxtVector and concretising the latter will give an element higher in the lattice (= subset)

ordering than the former. That is we have modelled in our abstract domain all the computations which can occur in the collecting interpretation (together, in all probability, with some which cannot).

2.4.1 Example

As an example, we will show the independent attribute method (IAM) given by Jones and Muchnick [29] (but also used in the Cousots' original formulation [8]) is an abstraction of the collecting interpretation (which Jones and Muchnick call the relational attribute method). In the collecting interpretation (COL), given above, we define the set of contexts to be

$$\text{Ctxt-COL} = 2^{\text{Var} \rightarrow \text{Val}}$$

thus giving the set of environments possible at a program point. In IAM we define contexts by

$$\text{Ctxt-IAM} = \text{Var} \rightarrow 2^{\text{Val}}$$

which gives the set of possible values associated with each variable. We can see intuitively that IAM is weaker in that it does not allow us to represent the fact that a variable may not have a certain value when another variable takes a specified value. We can set up the abstraction function

$$\text{Abs: Ctxt-COL} \rightarrow \text{Ctxt-IAM}$$

by

$$\text{Abs}(C) = \lambda v. \{f(v) : f \in C\}.$$

This defines

$$\text{Conc}(C') = \{f \in \text{Ctxt-COL}: f(x) \in C'(x) \ \forall x \in \text{Var}\}.$$

For more examples see the Cousots' expository paper [9], but we will press on to consider applicative languages.

2.4.2 Lattice of Interpretations

The abstraction relation on interpretations just defined is transitive and reflexive and therefore forms a quasi-partial order (for proof consider composition and the identity function). We can define an equivalence relation on interpretations by identifying interpretations which are both abstractions of each other.

Upon identifying such equivalent interpretations we derive a quotient relation on equivalence classes of interpretations of a given schema which is now a partial order on the lattice of equivalence classes.

2.5 Abstract Interpretation for the Applicative Idiom

This section develops a variant of the Cousot style of abstract interpretation presented in [9] which is more suitable for applicative languages. In applicative languages the notion of program point is not immediately available, although Jones [27] presents an interesting use of dataflow analysis for the lambda calculus in which he essentially constructs a representation of program points by modelling the states processed by the mathematical operational interpreter of Plotkin [40]. His work is of interest because it represents a half way house between the formalism of this section and that used for flowcharts, albeit in an operational formulation.

For applicative languages then, with no notion of program point, the important concepts are:

- the meaning of functions

- the possible values of their parameters and results.

We would therefore like to derive methods which will enable us to approximate the possible sets of binding environments which can exist at a certain function call, just as it is natural to consider the set of environments which could exist at a program point in the flowchart idiom. For functions we will examine

$$\hat{f}(S) = \{f(x_1 \ldots x_k) : (x_1 \ldots x_k) \in S\}$$

(or superset approximations thereto) for sets S of tuples of values. This will be used to construct (an approximation to) the set of values computed by a function when given (an approximation to) a set of possible argument tuples.

<u>2.5.1 Recursion equation schema</u>

Let $\{F_i;\ 1 \leq i \leq n\}$ be a set of uninterpreted function symbols, with arity k_i; $\{A_i\}$ be base function symbols, with arity r_i; and $\{X_i\}$ be a countable set of individual parameters.

A program schema, Σ, is a set of recursion equations

$$\{F_i(X_1 \ldots X_{k_i}) = U_i;\ \ 1 \leq i \leq n\}$$

with the U_i members of WFF(k_i) where the WFF(p) are the sets of well formed terms constructed from

$$\{A_i;F_j;X_1 \ldots X_p\}$$

These equations provide a functional environment for the evaluation of terms from WFF(0) under a given interpretation. However, for definiteness, we will assume that k_1=0 and the 'program' consists of evaluating

$$[\![F_1()]\!]$$

in this environment.

2.5.2 Recursion equation semantics

An interpretation, I, of a program schema is a pair <D,a_i> where D is a domain (here cpo), and the

a_i: $D^* \rightarrow D$

are functions interpreting the A_i and hence define constants and base functions. They must satisfy some suitability condition (here continuity). Such an interpretation naturally defines a semantic function

$Eval_I$: Expressions $\rightarrow$ Denotations

by providing meanings to atomic terms, and thence to compound expressions. This provides meanings, f_i, to the F_i by the standard least fixpoint method.

As a parenthetic remark oriented at the reader familiar with universal algebra we may view $\{A_i\}$ and $\{A_i\} \cup \{F_i\}$ as the operator parts of signatures $(V,\{A_i\})$ and $(V,\{A_i\} \cup \{F_i\})$ of algebras with a single sort, V say. By abuse of notation we will use the name A-algebra to refer to a $(V,\{A_i\})$-algebra and the name AF-algebra similarly. Thus an interpretation (D,a_i) is (a carrier and functions for) an A-algebra. A recursive program scheme Σ induces an algebra morphism (which we shall also call Σ) from A-algebras to AF-algebras by composition and taking of fixpoints. In general, except for the carriers (which will be called D, L, M), we will use capitals for sorts and operators of the signature, and lower case for elements of the carrier and functions.

2.5.3 The collecting interpretation

This section differs from the treatment given elsewhere (only

Cousot [10] and Jones [27] consider functions) in that we extend the theory to cover partial functions. Partial functions are represented in the standard mathematical manner as returning $\perp$, the bottom element of some CPO, when the conceptual partial function is undefined. This enables us to derive a strong theory, rather than in the Cousots' work above where we have to qualify results with "if the expression terminates". The latter has a correspondence with theories of partial correctness such as Floyd's flowchart proofs [14], further developed by Hoare [24]. There termination must be established independently rather than being naturally considered as part of a theory of strong correctness. The Cousots' paper above uses recursive procedures (not functions) so that it is quite suitable for applying Floyd-like rules to derive weak properties of functional behaviour.

Here we will apply such considerations to derive a 'collecting' interpretation in a different manner from the Cousots. This will require that we use a power domain rather than their use of a power set ordered by inclusion. This gives a more natural collecting interpretation, which can be justly claimed to be more suitable on the grounds that the power set interpretation is an abstraction of our power domain interpretation (see section 2.7.3).

In order to build a collecting interpretation for Σ which considers sets of values we must first define what we mean by the set of all subsets of D. Let

$$E = 2^D$$

be the power domain (see Plotkin [41]) of the CPO $(D, \sqsubseteq)$. If D is

flat, then E is just the set of non-empty subsets of D such that all infinite elements of E contain $\perp$.

E is associated with the Egli-Milner ordering $\sqsubseteq$ which is defined by

$$P \sqsubseteq Q \text{ iff } (\forall p \in P.\ \exists q \in Q.\ p \sqsubseteq q)\ \&$$
$$(\forall q \in Q.\ \exists p \in P.\ p \sqsubseteq q)$$

For a flat domain D, that is

$$p \sqsubseteq q \text{ iff } p=\perp \text{ or } p=q$$

this reduces to

$$P \sqsubseteq Q \text{ iff } P=Q \text{ or}$$
$$\perp \in P\ \&\ P-\{\perp\} \subset Q.$$

E also has an induced subset ordering, and here we will follow Hennessy and Plotkin [23] and define a nd-cpo (non-deterministic cpo) $(L, \sqsubseteq_L, \cup_L)$ to be a cpo with a continuous operation called union $\cup$: L X L $\rightarrow$ L satisfying the standard (set) axioms for union (commutativity, associativity and idempotency ($x \cup x = x$)). The union operation naturally defines a subset relation on L given by

$$l_1 \subset l_2 \text{ iff } l_1 \cup l_2 = l_2$$

which we shall assume available when required.

We will define the collecting interpretation, J, say, to be $(2^{D^*}, b_i)$ where the b_i are given by the following derivations from the definitions of the a_i:

$$b_i: 2^{D^*} \rightarrow 2^D$$

which is defined by

$$b_i(S) = \{a_i(x_1 \ldots x_{r_i}): (x_1 \ldots x_{r_i}) \in S\}.$$

The b_i are continuous under the Egli-Milner ordering.

This correspondingly induces collecting definitions g_i for the F_i given by (the least fixpoint of)

$$g_i(S) = \{Eval_J[\![U_i]\!][\{x_j\}/X_j]: (x_1 \ldots x_{k_i}) \in S\}$$
$$Eval_J[\![X_i]\!]\ r = r[\![X_i]\!]$$
$$Eval_J[\![A_i(e_1 \ldots e_{r_i})]\!]\ r = b_i[Tuple(Eval_J[\![e_1]\!]r \ldots Eval_J[\![e_{r_i}]\!]r)]$$
$$Eval_J[\![F_i(e_1 \ldots e_{k_i})]\!]\ r = g_i[Tuple(Eval_J[\![e_1]\!]r \ldots Eval_J[\![e_{k_i}]\!]r)]$$
$$Tuple(S_1 \ldots S_k) = S_1 \times \ldots \times S_k$$

Note that these equations only represent a monotonic functional under the Egli-Milner ordering. The power set (inclusion) ordering does not have this property, since we wish the first approximation to the g_i to be $\lambda S. \{\perp\}$.

As in the flowchart formulation of abstract interpretation we have that the denotation of program schema Σ under I is the value, x, say, if and only if the denotation of Σ under J is the set X = {x}. Because of this property, we will study the relations of abstract interpretations with J, rather than I.

2.6 An Approximating Interpretation

Let (L,b_i) such an interpretation (L,b_i), with $L = (L,\sqsubseteq_L,\cup_L)$, and $(M,\sqsubseteq)$ be a cpo (we will not yet require a union operation in M), and we consider what it means for M to be an approximation to (abstraction of) L.

Before we can consider the notion of abstraction between interpretations we have to discuss the ordering we wish to place on our objects. Here, we will find, there is a difference between the concept of "is less defined than" and that of "will produce a smaller result set". This difference is not present in the

standard semantics of our deterministic recursion equations (since programs always give a single result). Neither is it present in the Cousots' work. However it is present in non-deterministic schemes (see Plotkin [41] or Hennessy [22]).

This difference is represented by the fact that a nd-cpo (here power domain) $(L, \sqsubseteq_L, \cup_L)$ has two different associated orderings: firstly it has a power domain ordering $\sqsubseteq$ and secondly an induced subset ordering $\subset$. We need the power domain ordering to set up our collecting interpretation in order that the least fixpoint functional should be continuous. For example in $(2^{\mathbb{Z}}, \sqsubseteq, \cup)$ we wish that

$\{2,3,\bot\} \sqsubseteq \{2,3\}$

in order to represent an improvement in evaluation.

However, when we wish to discuss the accuracy with which one interpretation models another, we will find that we need to use the subset ordering

$\{4,6\} \subset \{4,6,\bot\}$ and $\{4,6\} \subset \{4,5,6\}$

This is motivated by consideration of an example. Suppose the collecting interpretation of a schema gives {4,6} as the set of possible results. Now, due to our using an approximate version of the collecting interpretation, we may derive a result which corresponds to {4,5,6} in the collecting interpretation. For example consider the evaluations of

$[\![x + x \text{ where } x = \{2,3\}]\!]$ to produce {4,6}

and

$[\![\{2,3\} + \{2,3\}]\!]$ to produce {4,5,6}.

These correspond to the relational and independent attribute

approximations to the collecting interpretation for "+" (see section 2.7.1). Since we want to use the abstract interpretation to infer results about our collecting interpretation, by giving us a superset of possible results, it is clear that we will need to use the subset ordering of power domains to compare interpretations.

To summarise, if M is to model computation in L, we require two different orderings to represent the two ways a computation might be approximated:

- $\sqsubseteq$ models inaccuracy due to insufficient length of computation.
- $\subset$ models inaccuracy due to inaccurate steps in a computation.

2.6.1 Formal Definition of Abstraction

Let $(L, \sqsubseteq_L, \cup_L)$ be a nd-cpo (cpo with a continuous union operation), and $(M, \sqsubseteq_M)$ be a cpo. (Adding the axiom $\cup = \sqcup$ in the following will force L to be a lattice and hence simplify to the standard definition of abstraction.) For the reader acquainted with universal algebra the abstraction we define below is a map between the carriers L and M. We consider maps between the associated functions later.

We now examine the hypotheses that we will impose upon abstraction and concretisation maps: (these conditions given are quite likely to be over cautious, however they will allow us to formulate our abstraction relation)

Abs: $L \rightarrow M$

Conc: $M \rightarrow L$.

Firstly, in order that Abs can model abstraction of computation into a simpler domain we will require that Abs is $(\sqsubseteq_L, \sqsubseteq_M)$ continuous. This is necessary to enable fixpoints over L and M to be related (fixpoints use $\sqsubseteq$). Furthermore we require that Conc is continuous.

We will also require that M has no indistinguishable objects (this is the concept of exactness isolated by Nielson [37]).

$$Abs(Conc(m)) = m \quad \text{for all } m \text{ in } M$$

This implies that Abs is surjective and Conc injective and that Conc o Abs is idempotent. Functions with this property are sometimes called projections in the literature but no standard nomenclature appears to exist. Functions satisfying the additional property that

$$Conc \circ Abs \sqsubseteq \underline{id}_L$$

are often called retractions and those satisfying

$$Conc \circ Abs \sqsupseteq \underline{id}_L$$

are called (upper) closure operators. In our work we will choose a rather different extra property as described in the next paragraph.

For the purposes of abstract interpretation we need to be able to deduce properties of programs over L by considering those over M. In particular we wish the result of a computation over L to be a subset of Conc applied to the result of a corresponding computation over M, in order to consider at least as many values as can actually occur in the L computation. This will require, by considering the empty (identity) computation, that

$$Conc(Abs(l)) \supseteq l \quad \text{for all } l \text{ in } L$$

We will also require that Conc o Abs is $\subseteq_L$ monotonic.

These arguments motivate:

Definition

We say that continuous functions Abs: $L \longrightarrow M$ and Conc: $M \longrightarrow L$ for an nd-cpo $(L, \sqsubseteq_L, \cup_L)$ and a cpo $(M, \sqsubseteq_M)$ are exactly power adjoined in the following circumstances:

- Abs o Conc = $\underline{id}_M$
- Conc o Abs $\supset_L$ $\underline{id}_L$
- Conc o Abs is $\subset_L$ monotonic

Note that now, if we temporarily assume $\subset_L = \sqsubseteq_L$, then L and M are exactly adjoined lattices, as used by the Cousots in [9]. In this situation we would have

$$\text{Conc}(m) = \bigsqcup\{l: \text{Abs}(l) = m\}.$$

Here this is independent, and in fact $\bigsqcup$ will not even be defined for all such sets since we do not insist that L is a lattice. However we do have the corresponding

Theorem

$$\text{Conc}(m) = \bigcup\{l: \text{Abs}(l) = m\}.$$

Proof

Since Abs(Conc(m)) = m we have that the union contains Conc(m) thus giving

$$\text{Conc}(m) \subset \bigcup\{l: \text{Abs}(l) = m\}.$$

Now suppose Abs(l) = m then we have

$$l \subset \text{Conc}(\text{Abs}(l)) = \text{Conc}(m)$$

and hence

$$\bigcup\{l: \text{Abs}(l) = m\} \subset \text{Conc}(m)$$

as required.

2.6.2 An alternative view of Conc and Abs

Since Conc(M) is isomorphic (as a cpo) to M (the isomorphism is given by Abs restricted to Conc(M) and Conc) we can consider Abs and Conc to be a continuous function Clo: L $\rightarrow$ L satisfying the axioms for a (set) upper closure operator: Clo is idempotent, $\subset$ monotonic and Clo $\supset$ $\underline{id}_L$.

2.6.3 The Abstraction of Functions

Suppose $(L_1, \sqsubseteq_{L_1}, \cup_{L_1})$ and $(L_2, \sqsubseteq_{L_2}, \cup_{L_2})$ are nd-cpo's and M_1 and M_2 are respectively abstractions of these domains via exactly power adjoined functions $(Abs_i, Conc_i)$. Then we will say, for continuous functions g and h, that h: $M_1 \rightarrow M_2$ is an _abstraction_ of g: $L_1 \rightarrow L_2$ if we have

$$Conc_2(h(Abs_1(l))) \supset g(l) \quad \forall \ l \text{ in } L$$

or, removing references to elements

$$Conc_2 \circ h \circ Abs_1 \supset g.$$

This condition ensures that any computation performed in the M_i represents a superset of the possible results of the corresponding computation in the L_i, thereby providing sufficiency conditions for correctness, or optimisation of the L_i computation. (For the reader acquainted with universal algebra we are extending the idea of abstraction from carriers to functions.)

Note that such an abstraction, h, always exists, since defining

$$Abs(g) = h = Abs_2 \circ g \circ Conc_1$$

gives

$$\begin{aligned} g' &= Conc_2 \circ h \circ Abs_1 \\ &= (Conc_2 \circ Abs_2) \circ g \circ (Conc_1 \circ Abs_1) \\ &\supset g \circ (Conc_1 \circ Abs_1) \end{aligned}$$

Now that g is monotonic with respect to $\subseteq$. That is

$$a \subseteq b \Rightarrow g(a) \subseteq g(b)$$

since g is a function in the collecting interpretation defined by

$$g(l) = \{f(x): x \in l\}.$$

(When we consider abstractions of interpretations more general than the collecting interpretation we will need to add this as an axiom about g.) Further, since

$$\text{Conc} \circ \text{Abs} \supseteq \underline{id}$$

we have

$$g' \supseteq g \circ (\text{Conc}_1 \circ \text{Abs}_1) \supseteq g$$

as required.

Note that Abs is not an algebra morphism in the usual sense of the word since in general we will not have

$$\text{Abs}(g_2 \circ g_1) = \text{Abs}(g_2) \circ \text{Abs}(g_1).$$

An example of why this is so is given in section 2.7.2.

2.6.4 Correctness of an Abstraction

Let the L_r be $2^{(D^r)}$, then the collecting interpretation defines

$$b_i: L_{r_i} \rightarrow L_1$$

from the definitions of the a_i. Now let M_r be abstractions of the L_r (via functions Abs_r and Conc_r). As in the previous section we have definitions

$$c_i: M_{r_i} \rightarrow M_1$$

induced from the b_i, the Conc_r and the Abs_r.

However we have two possible ways of forming the semantics, h_i of the F_i on the M_r. Firstly they have a natural fixpoint definition induced by our scheme, Σ. Secondly, they can be formed in the above manner, by abstraction of the meanings, g_i, of the F_i

over the L_r. Clearly, from the point of view of computing the h_i we would like to derive them from the fixpoint equations from the c_i, since the M_r are simpler domains. Therefore, as in the flowchart case, correctness is simply a matter of showing that the h_i are always abstractions of the g_i.

We can formulate this (for the interested reader) in the general algebraic framework as:

Correctness Theorem

Let (L,b_i) and (M,c_i) be A-algebras such that (M,c) abstracts (L,b). We have a algebra morphism Σ induced by our recursive program scheme mapping (L,b) and (M,c) onto AF-algebras $(L,b \cup g)$ and $(M,c \cup h)$. Now $(M,c \cup h)$ is an abstraction of $(L,b \cup g)$.

2.6.5 Proof of the Correctness Theorem

The proof of correctness is done inductively: the base case, for system functions, is immediate from the definition in section 2.6.3.

We next show (the inductive step) that composition of the h_i gives an abstraction of composition of the g_i. Let $g, g': L \to L$ be composible with abstractions h and $h': M \to M$. We must show that $h' \circ h$ is an abstraction of $g' \circ g$. We have that

$$g \sqsubseteq Conc \circ h \circ Abs.$$

We have a corresponding inequation for g' and h' (we elide the subscripts on Abs and Conc for simplicity). We now observe that

$$g \circ g' \sqsubseteq g \circ Conc \circ h' \circ Abs$$

since we require g to preserve $\sqsubseteq$, that is

$$x \sqsubseteq x' \Rightarrow g(x) \sqsubseteq g(x'),$$

as in section 2.6.3. Substituting for g in the right hand side of

the above gives

$$g \circ g' \sqsubseteq Conc \circ h \circ Abs \circ Conc \circ h' \circ Abs$$
$$= Conc \circ h \circ h' \circ Abs$$

as required.

Finally we perform the induction by considering the n-depth approximations of functions defined by fixpoints and passing to the limit. Let g^* and h^* be the limits of sequences of g^i and h^i produced by the same fixpoint scheme. Now, the least functions in the domains are given by

$$g^0(x) = \perp$$
$$h^0(x) = Abs(\perp)$$

and h^0 is an abstraction of g^0 since

$$Conc \circ h^0 \circ Abs\ (x) = Conc(Abs(\perp))$$
$$\sqsupseteq \perp = g(x).$$

Moreover we showed above that composition preserved abstraction. Therefore we have that h^i abstracts g^i for all i, since h^{j+1} is defined in terms of a composition possibly including h^{j+1}. Now we turn our attention to the sequence

$$p^i = Conc \circ h^i \circ Abs$$

which is increasing with limit

$$p^* = Conc \circ h^* \circ Abs$$

due to the continuity of Conc (with respect to $\sqsubseteq$). The fact that the h^i abstract the g^i can be written as $p^i \sqcup g^i = p^i$. Continuity of $\sqcup$ implies that $p^* \sqcup g^* = p^*$ or, re-phrasing again, that $g^* \sqsubseteq p^*$ or

$$g^* \sqsubseteq Conc \circ h^* \circ Abs$$

as required.

2.7 Some Example Abstract Interpretations

This section gives three applications of our idea of power domain based abstraction. However, they should not be simply treated as examples since they embellish the theory. For example section 2.7.3 shows how our formulation is more general than the Cousots'.

2.7.1 Application: The Independent Attribute Formulation

As in the flowchart scheme (see section 2.4.1 for definitions), we can define an independent attribute method (IAM) formulation of the collecting interpretation. We will show that this is an abstraction of the given (relational attribute) formulation. (These terms are taken from Jones and Muchnick [29].) Let

$$E = 2^D,$$

then we define the IAM interpretation by defining the base functions

$$c_i: E^* \rightarrow E$$

where

$$\begin{aligned} c_i(X_1 \ldots X_k) &= \{a_i(x_1 \ldots x_k): x_1 \in X_1, \ldots, x_k \in X_k\} \\ &= b_i(X_1 \times \ldots \times X_k) \end{aligned}$$

Again the c_i so defined are continuous with respect to the Egli-Milner ordering (see [41]). Similarly this induces meanings

$$h_i: E^* \rightarrow E$$

via the fixpoint equations.

Note that such a general definition of the c_i would be impossible for the method given in the Cousot paper [10] since the natural image of $\perp$ in their framework of $2^{D-\{\perp\}}$ ordered by inclusion is {}, the empty set. This would imply that any function

UNIV. EDINBURGH LIBRARY

so modelled would be strict because, for c_i so defined,

$$c_i(X_1 \ldots X_k) \;=\; \{\} \text{ if } X_i = \{\} \text{ for any } i.$$

Therefore in such a case we would need to treat non-strict functions as special cases, contrary to our desire to build a natural theory of denotational abstract interpretation incorporating partial functions. We will show later (section 2.7.3) that the Cousot collecting interpretation is an abstraction of (ie less general than) our version.

2.7.2 Application: Abstracting Termination

Here we show that it is possible, in our theory, to abstract termination conditions (just as we will later show that we can abstract value properties ignoring termination when we show that the Cousots' collecting interpretation is an abstraction of ours).

We consider the # and $\flat$ functions to be considered in chapter 3. Both # and $\flat$ are particular versions of the 'Abs' functions discussed and their associated Conc functions will be called #' and $\flat$'.

We will leave chapter 3 in its original form because it is then easier to read independently and provides an alternative to the more abstract definition given below. As in the above, we will use the a_i for semantics of the base function, b_i for the collecting interpretation of these, and c_i for abstractions thereto. Let T = {0,1} be ordered by 0<1. Let D be a flat domain, then the # and $\flat$ functions lift (to $L = 2^D$) the Halt function given by

$$\text{Halt}: D \rightarrow T$$
$$\text{Halt}(\perp) \;=\; 0$$
$$\text{Halt}(x) \;=\; 1 \text{ if } x \neq \perp$$

We define

$\#, \flat: L \rightarrow T$

$S^{\#} = 0$ if $S = \{\bot\}$

$S^{\#} = 1$ otherwise

$S^{\flat} = 0$ if $\bot \in S$

$S^{\flat} = 1$ otherwise

Note that these definitions are only monotonic with respect to $(\sqsubseteq, \leq)$.

The concretisation method given above (section 2.6.1) defines

$\#', \flat': T \rightarrow L$

$1^{\#'} = D$; $\quad 0^{\#'} = \{\bot\}$

$1^{\flat'} = D - \{\bot\}$; $\quad 0^{\flat'} = D$

This enables us to define functions

$c^{\#}$ and $c^{\flat}: T^{*} \rightarrow T$

by

$$c^{\#}(x_1 \dots x_k) = \# \, b(x_1^{\#'} \dots x_k^{\#'})$$
$$c^{\flat}(x_1 \dots x_k) = \flat \, b(x_1^{\flat'} \dots x_k^{\flat'})$$

where

$$b(s_1 \dots s_k) = \{a(x_1 \dots x_k): x_i \in s_i\}$$

We check that the $(\#,\#')$ and $(\flat,\flat')$ pairs are power adjoined, and the c_i abstract the b_i. Then for meanings h of defined functions F, under $\#$ or $\flat$ we have that $h^{\#}$ and $h^{\flat}$ give only valid properties of programs.

At this point it is convenient to give an example ($\#$ above) of an abstraction function which is not an algebra morphism. Consider the function definition

F() = IF(true,$\bot$,91)

we have that (see chapter 3 for more details)

$F^{\#}() = 0$
$IF^{\#}(x,y,z) = x \wedge (y \vee z)$
$true^{\#} = 91^{\#} = 1$
$\perp^{\#} = 0.$

but

$IF^{\#} \circ (true^{\#}, \perp^{\#}, 91^{\#}) = 1.$

This phenomenon occurs in other circumstances than the above use of undefined functions.

2.7.3 Application: Power Sets are an Abstraction of Power Domains

We will here indicate that our power domains generalise the standard power set method used by the Cousots by showing that their collecting interpretation can be represented as an abstraction of ours. We will assume that D is a flat domain both for simplicity and for the reason that the theory has only been developed for such domains. The maps we consider are sufficiently natural (in the mathematical sense) that we would expect the extension to general domains to be straightforward, however power domains at higher types can pose difficulties.

Take our nd-cpo $(L, \sqsubseteq_L, \mathbf{U}_L)$, then, on putting $\mathbf{U} = \sqcup$ we find that L is a lattice (all lowest upper bounds exist by definition) and the rules for Abs and Conc reduce to

$Abs \circ Conc = \underline{id}$
$Conc \circ Abs \sqsupseteq \underline{id}$

This is merely the Cousots' definition of (exactly) adjoined pair on which they base their theory - thus our work does represent a generalisation.

Clearly we have a natural map from

$L_1 = (2^D, \sqsubseteq)$ qua power domain

to

$M_1 = (2^{D-\{\bot\}}, \leq)$ qua power set

given by

Abs: $L_1 \rightarrow M_1$: $l \rightarrow l-\{\bot\}$.

Section 2.6.1 gives conc as

Conc: $M_1 \rightarrow L_1$: $m \rightarrow m \cup \{\bot\}$.

This shows that the Cousot power set collecting interpretation is an abstraction of our power domain one.

Similarly we can set up L_r and M_r to give respectively the power domain of D^r and its strict approximation $2^{[D-\{\bot\}]^k}$. The Abs_i are a kind of smash operation identifying (in the M_r) all elements (in the L_r) which have any undefined component.

Again, merely checking that this does in fact define a valid interpretation will give us the power to deduce properties in L (and hence in D) from those in M. Here however we can use the Cousots' abstraction relation defined on an abstraction, N, say, of M (in their sense) to prove properties of the computation in M from computation in N; and thence, by our abstraction, in L.

This provides the idea of composing abstraction which the Cousots use to derive a lattice of abstractions of a given collecting interpretation. The next section indicates some possible methods which allow us to set up the framework of a lattice of abstract interpretations under our formulation of abstraction.

2.8 The Lattice of Interpretations

As in the Cousots' work, given our abstraction relation as

developed in the previous section, it is natural to want to extend it to compare any two interpretations, rather than simply building the set of all abstractions abstracting the collecting interpretation. There are at least two possibilities for doing this, firstly a simple and general comparison method and secondly a more sophisticated method of composing two abstractions to give a third.[1]

Firstly, there is the general construction which enables us to put an order (the finest possible) on the images under abstraction of a given nd-cpo L. Let M_1 and M_2 be such images abstracted by Abs_1 and Abs_2. We can define $(M_1, Abs_1) \leq (M_2, Abs_2)$ if there is a (continuous) map F, say, $F: M_2 \rightarrow M_1$ such that $Abs_1 = F \circ Abs_2$. However, such maps do not seem to preserve enough structure of L in the M_i.

Alternatively, we might consider the natural suggestion of insisting that Abs and Conc preserve more of the nd-cpo structure, since the problem in wishing to define a chain of abstractions

$$Abs_1: (L, \sqsubseteq_L, \cup_L) \rightarrow (M, \sqsubseteq_M); \quad Abs_2: (M, \sqsubseteq_M) \rightarrow (N, \sqsubseteq_N)$$

is that M does not have the union operator which we require to define abstraction. Otherwise there is no problem - we can compose the Abs_i and $Conc_i$ without restriction. From the view that M is isomorphic to $Conc_1(M)$, the natural union operator $\cup_M$ is given by

$$m_1 \cup_M m_2 = Abs_1(Conc_1(m_1) \cup_L Conc_1(m_2)).$$

---

[1]For universal algebraicists this is just the statement that our A-algebras with our abstraction relation form a category. Similarly we have a category of AF-algebras with morphisms again abstraction. Furthermore the map $\sum$ (induced by the recursive program scheme) is now a functor between these categories.

However in general there seems to be no reason why this should be a union operator (associativity is not guaranteed). A solution is to insist that Abs preserves unions (such functions are called linear by Hennessy and Plotkin [23]), thereby ensuring that such a definition does indeed define a union operator:

$$Abs_1(l_1 \cup_L l_2) = Abs_1(l_1) \cup_M Abs_1(l_2).$$

This is quite consistent with the Cousots' formulation since there we have that Abs is a distributive continuous function with

$$Abs(\bigsqcup\{l_i\}) = \bigsqcup\{Abs(l_i)\},$$

and this work can be considered to be a method of separating the uses of $\cup$ and $\bigsqcup$ which are identified in the their work. Actually we will only use the following weaker conditions relating the union operators:

$$Abs_1(l_1 \cup_L l_2) = Abs_1(l_1) \cup_M Abs_1(l_2) \quad \forall\ l_1, l_2 \text{ in } Conc(M)$$

Abs_1 is $(\subset_L, \subset_M)$ monotonic

Finally, it appears that we also require a condition on Conc, again generalising the Cousots' lattice based theory which has Conc monotonic with respect to $\sqsubseteq$, and accordingly we insist that Conc is $\subset$ monotonic. This enables us to prove

Theorem

The composition of abstraction maps

Abs_1: $(L, \sqsubseteq_L, \cup_L) \rightarrow (M, \sqsubseteq_M, \cup_M)$

Abs_2: $(M, \sqsubseteq_M, \cup_M) \rightarrow (N, \sqsubseteq_N, \cup_N)$

gives an abstraction map

$Abs_2 \circ Abs_1$: $L \rightarrow N$.

Proof

Let $Conc_1$ and $Conc_2$ be the (uniquely determined) concretisation maps corresponding to Abs_1 and Abs_2. Now we will show that

$Conc_1$ o $Conc_2$ acts as an concretisation map for Abs_2 o Abs_1. We must show

- Abs_2 o Abs_1 o $Conc_1$ o $Conc_2$ = $\underline{id}_N$

- Comp = $Conc_1$ o $Conc_2$ o Abs_2 o Abs_1
  has Comp $\supseteq$ $\underline{id}_L$ and Comp is $\subseteq$ monotonic.

- Abs_2 o Abs_1 distributes over $\cup$
  $Conc_1$ o $Conc_2$ is $\subseteq$ monotonic

The first and third of these is trivial (composition preserves the properties). Now

$Conc_2$ o Abs_2 o Abs_1 $\supseteq$ Abs_1

due to the fact that

$Conc_2$ o Abs_2 $\supseteq$ $\underline{id}$.

Using $\subseteq$ monotonicity of $Conc_1$ gives

Comp $\supseteq$ $Conc_1$ o Abs_1 $\supseteq$ $\underline{id}$

as required. To prove $\subseteq$ monotonicity of Comp we note the Abs_i are $\subseteq$ monotonic and so are the $Conc_i$. Hence so is Comp.

One final remark is to the effect that our assumptions as to the $\subseteq$ monotonicity of Abs and Conc render the axiom

Conc o Abs is $\subseteq$ monotonic

(given in section 2.6.1) superfluous.

2.9 Notes on the Abstraction Relation

We here mention a few points which could not conveniently be given in the text due to their ability to confuse.

Firstly, we use the notion of a nd-cpo which is a domain with continuous operation, called union, satisfying the axioms of associativity, commutativity and idempotency. However these are exactly the same axioms that categorise an intersection operation.

Therefore our work has a natural dual (just as the Cousots' work on complete lattices has the $(\sqcup, \sqcap)$ duality). This could be used to infer a subset of the results of a program instead of the above work which aimed to model a superset of the possibilities. This would be useful to prove that run-time errors do occur, and hence the program under analysis is incorrect, just as our work shows that certain states did not occur in order to validate certain optimisations.

Note that our use of cpo's (for semantic domains) rather than complete lattices is absolutely essential in that the power domain construction does not properly work for lattices. For example, it can be shown that if D is a cpo containing three elements related by $a \sqsubseteq b \sqsubseteq c$ then the power domain of D has $\{a,b,c\} = \{a,c\}$. Thus, if we use a lattice then any subset containing the top and bottom elements is equivalent to any other - this fact makes unusable much of the strong abstract interpretations we have developed above for power domains. However, we would not claim that this represents a weakness of our work, but merely indicates how artificial elements (top elements have no semantic basis) can cause artificial problems. We note that the modern style is to use cpo's to set up semantic domains rather than the older Scott style lattice-based semantics.

One final point concerns the fact that our work requires union operators where the Cousots' theory does not. As indicated their theory uses lattices which automatically have two natural union operators ($\sqcup$ and $\sqcap$) and using either of these in the source of

both Abs and Conc ensures that our additional axioms are satisfied.

2.10 Deducing Properties of Applicative Programs

This section gives a quick introduction as to how we might use the work above for transforming programs to improve their efficiency and is not central to the rest of the thesis. We discuss how we can use the non-standard interpretations of user defined function symbols in order to obtain global properties on applicative program execution as the non-standard interpretations are less useful in themselves.

The work on abstract interpretation for the applicative idiom given in section 2.5 concentrates entirely on finding meanings within the abstract interpretation for functions. That is, given $a_i\colon D^{r_i} \rightarrow D$, the meaning of a base function, we deduce an abstract meaning $c_i\colon M_{r_i} \rightarrow M_1$, for a base function symbol, via the collecting interpretation $b_i\colon L_{r_i} \rightarrow L_1$, where $L_n = 2^{D^n}$. We then use this to infer, via the least fixpoint equation in our abstract universe, an approximate meaning $g_i\colon M_{k_i} \rightarrow M_1$ for user function definitions.

We now wish to calculate more directly relevant properties of functions, for example the set of possible parameters supplied to, or results given by, a given function. We do this by noting that the meanings c_i and g_i define an evaluation function, as given previously. Therefore the set of possible parameters to a given function in a given call $[\![F(e_1 \ \ldots \ e_k)]\!]$ is just given by (concretising)

$$S = Eval_M[\![e_1]\!] \times \ldots \times Eval_M[\![e_k]\!]$$

similarly the set of possible results produced by this call is simply Conc(g(S)) where g is the abstract meaning of F.

The set of possible parameters passed to F from all calls within the program is just the union, over all calls to F in the program, of the terms like S given above. The set of all possible results from F within this program is given similarly. Producing highly optimised code for applicative languages can be seen as partitioning this union suitably, and then producing versions for F for each of these cases by partial evaluation of the standard definition.

Note that the optimising described here is at a middle level, intermediate to machine dependent 'peep-hole optimisation' and full program transformation which acts by changing the algorithm as described by Burstall and Darlington [6]. However we would claim that our method has a greater chance of being used automatically than any algorithm changing method.

Chapter 3: The Theory and Practice of Transforming Call-by-need into Call-by-value

3.1 Abstract

Call-by-need (which is an equivalent but more efficient implementation of call-by-name for applicative languages) is quite expensive with current hardware and also does not permit full use of the tricks (such as memo functions and recursion removal) associated with the cheaper call-by-value. However, the latter mechanism may fail to terminate for perfectly well-defined equations and also invalidates some program transformation schemata.

Here a method is developed which determines lower and upper bounds on the definedness of terms and functions, this being specialised to provide sufficient conditions to change the order and position of evaluation keeping within the restriction of strong equivalence. This technique is also specialised into an algorithm analogous to type-checking for practical use which can also be used to drive a program transformation package aimed at transforming call-by-need into call-by-value at 'compile' time.

We also note that many classical problems can be put in the framework of proving the strong equivalence where weak equivalence is easy to show (for example the Darlington/Burstall fold/unfold program transformation).

3.2 Motivation

For a purely applicative language (no assignment or GOTO) call-by-need [51] is a highly desirable parameter passing mechanism, since Vuillemin[50] shows it is a safe evaluation mechanism in that it will give the mathematical result whenever the latter is defined and is more efficient than call-by-name.

Basically call-by-need is the same as call-by-name (passing of an expression bound in the calling environment) but with the proviso that the first reference to the parameter causes not only its evaluation but also the replacement of the parameter with the result of the evaluation thus making subsequent accesses much cheaper. It also has the advantage that it corresponds closely to the method a mathematician would use to evaluate an expression. Note that it retains the advantages of call-by-name in that parameters that are not referenced in a particular activation of the function will not be evaluated: this point is very important since evaluating an argument which should not be evaluated may result in the evaluator looping. To summarise, we have that

- call-by-value evaluates a parameter exactly once,
- call-by-name evaluates a parameter zero or more times,
- call-by-need evaluates a parameter at most once.

The main disadvantage of call-by-value is that it may produce undefined values for (mathematically) well defined expressions, for example consider evaluating

```
f(1,0)  WHERE  f(x,y) =  IF x=0 THEN 0 ELSE f(x-1,f(x,y))
```

using call-by-value.

Note that this point is especially relevant to the typical user of a symbolic algebraic manipulation (SAM) system, who is mathematically sophisticated but computationally naive, because he will write similar (but less contrived) recursive definitions and find the system merely moans that time is up!

For the user of a SAM system it is desirable to use call-by-need as the parameter passing mechanism in order that

1. The recursive definitions are as fully defined as possible.

2. The print program may drive the evaluation process so that printing an infinite expression will run out of time when printing it and not during the evaluation prior to printing.

The counter arguments favouring call-by-value are:

1. Call-by-need is clumsy to implement on current architectures (in that each parameter to a function needs to carry a closure around with it). This leads to differences in efficiency which are put by various sources at factors of between 2 and 10. The situation becomes rather worse in a full lazy evaluator[15, 21] where an evaluation of an expression can be suspended with unevaluated sub-expressions.

2. With call-by-value the system can use memo-functions (due to Michie[32]) to avoid recomputation. These will be (semantically) invisible to the user, and encourage the development of clean "mathematical" rather than "sequential" programs. For example consider:

```
    f(n) = IF n<2 THEN 1 ELSE f(n-1)+f(n-2)
                                        (Fibonacci numbers)
```

or

```
    C(n,r) = 1  IF r=0 OR r=n
           = C(n-1,r-1) + C(n-1,r) OTHERWISE
                                        (Pascal's triangle)
```

Here evaluation (with r = n/2 in the second example) requires in the order of 2^n function calls using the standard implementation. This cost can be made linear in n in exchange for storage by saving the (arguments,result) pairs for previously computed values of f or C. (This technique is called 'memo'ing the function). Unfortunately when using call-by-need, we cannot look at the argument values since to do so causes evaluation effectively at the time of call and hence is equivalent to a call-by-value regime. Thus call-by-value has advantages which extend far beyond current hardware limitations - since exponential costs can rarely be tolerated.

3. Call-by-need does not permit the standard methods of recursion removal to be used, for example:

   ```
   f(x,y) = IF x=0 THEN y ELSE f(x-1,y+1)
   ```

   requires one new closure to be created for y in each recursive call; these all being evaluated 'domino fashion' when y is finally used. For further discussion see Lang[31].

It is worth noting the great similarity between the optimisations furnished by call-by-need over call-by-name and by using memo functions. In both cases the effect is to avoid recalculation of known values, and both are optimisations which can convert an exponential cost into a linear one (unlike traditional compiler optimisations to remove common sub-expressions which can only save at most a linear factor in the cost).

Another reason for using the call-by-need parameter passing mechanism is that call-by-value invalidates some program transformation schemata. For example consider the fold/unfold transformation of Darlington and Burstall[6] which replaces a call

of a function by its body or vice versa.

The program segment

```
IF e1 THEN e2 ELSE e3                                    ... (1)
```

is equivalent to the segment

```
f(e1,e2,e3) WHERE f(x,y,z) = IF x THEN y ELSE z          ... (2)
```

only if the call-by-need (or name) parameter passing regime is used since the early evaluation of e2 or e3 otherwise necessitated by call-by-value in (2) may cause infinite looping. For example compare

```
fact(n) = IF n=0 THEN 1 ELSE n*fact(n-1)
```

with

```
fact(n) = f(n=0, 1, n*fact(n-1))
                    WHERE f(b,x,y) = IF b THEN x ELSE y
```

the latter being undefined for all n when using call-by-value.

The above arguments suggest that call-by-value is more efficient but call-by-need preferable on aesthetic/definedness considerations. So techniques are herein developed which allow the system to present a call-by-need interface to the user but which performs a pre-pass on his program annotating those arguments which can validly be passed using call-by-value. Thus the spirit is similar to, and unifies and implements some of the ideas in Schwarz[44].

Note that the technique only provides the information "It is safe to pass certain parameters by value" and is not claimed to detect all such cases. The problem of detecting all such cases is actually not effectively computable, for example consider:

```
F(x,y) = IF P(x) THEN y ELSE 0
```

where P(x) is true for all values of x. The argument y will, then, always be evaluated and so could be safely passed by value. This fact is impossible to detect uniformly since, in any sufficiently rich domain, there are tautologies which cannot be detected by any (pre-specified) algorithm (eg "The halting problem" for Turing machines). We make no attempt to detect similar tautologies and hence the system "plays safe" and suggests that y is passed by need. In practice this limitation does not stop most cases of call-by-value being detected (see section 3.6 on pragmatics).

There is a analogy between the system described here and the "most general type" inference system used in a language such as ML [17] which even extends to cover the sort of example above; for example consider the declaration

```
    LET x = IF true THEN 1 ELSE NIL
```

then the ML type rules will produce an error for the type of x whereas in fact it is well (but inelegantly) defined.

In order to be able to change the order of evaluation (eg changing call-by-need into call-by-name) without changing the semantics we require referential transparency in the language under study. Applicative languages normally possess this property, with the proviso that error situations (eg 1/0) do not result in 'jumpout' action and merely return a special error value to the calling function. Further discussion of this point may be found in section 3.9.

The central stage in the development of the call-by-value detection system is the definition of maps $\#$ and $\flat$ which are

semi-decision procedures for termination on recursion equations. The idea is that ♯ will map ALL terminating closed forms onto 1, and SOME non-terminating terms onto 0, and ♭ maps ALL non-terminating terms onto 0 and SOME terminating terms onto 1. By investigating the effect of ♯ and ♭ with their semi-homomorphic properties on recursion equations we can see the gross structure of the recursion and occurrences of references to arguments without the clutter of detail present in the original equations.

3.3 Formalism

The formal system in which the theory is developed is that of a scheme, S, of recursion equations together with one standard and two non-standard interpretations.

$$S = \{F_i(X_1 \ldots X_{k_i}) = U_i;\ 1 \leq i \leq n\}$$

where the U_i are (finite) terms defined by the grammar with start symbol T and axioms

- $T ::= X_j$ (individual parameters)
- $T ::= A_j(T_1 \ldots T_{r_j})$ (system functions)
- $T ::= F_j(T_1 \ldots T_{k_j})\ 1 \leq j \leq n$ (user functions)

We insist that U_i contains no X_r for $r>k_i$. Here all base constructs (including the conditional which is normally regarded as syntax) are considered to be members of the (A_i); note that the A_i are base constants when $r_i = 0$.

An interpretation I, of S, consists of a pair $\langle D,(a_j)\rangle$ where D is a domain and the a_i are continuous functions from $D^r \rightarrow D$ where $r = r_i$ is the arity of A_i.

An interpretation I induces for S an interpretation of the function symbols F_i defined in the usual manner as the least fixpoint.

Now let 2 = {0,1} be the two element Boolean lattice ordered by 0<1 and use the standard Boolean connectives (we use 0 and 1 to avoid confusion with elements of D).

3.4 Notation and Definitions

We use the following notation to simplify expressions:

1. [P,Q] is a partition of a set U if U is the disjoint union of P and Q.
2. Let F be a function with arity k then for a partition [P,Q] of {1 ... k} we define F(a/P, b/Q) to mean

   $F(x_1 \ldots x_k)$ where x_i = a if i in P
   = b otherwise

3. We will also use $R(x_Q)$ to mean $R(x_i)$ for all i in Q, where R is a predicate.

We next define two more interpretations in terms of $I = \langle D,(a_j)\rangle$ by

$$I^{\#} = \langle 2, (a_i^{\#})\rangle$$

and

$$I^{\flat} = \langle 2, (a_i^{\flat})\rangle$$

in the following manner: [the definitions are to be seen as monotonic functional extensions of the function

HALT: D $\longrightarrow$ 2 defined by

HALT(x) = 0 if $x = \perp$
= 1 otherwise]

For all partitions [P,Q] of $\{1,2 \ldots r_i\}$ we define

$a_i^{\#}(0/Q, 1/P)$ = 0 if for all $\{x_j$ in D: $1 \le j \le r_i\}$ such that $x_Q = \perp$, $x_P \ne \perp$ we have $a_i(x_1 \ldots x_{r_i}) = \perp$
= 1 otherwise

and

$$a_i^{\flat}(0/Q, 1/P) = 0 \text{ if there exists } \{x_j \text{ in } D: 1 \le j \le r_i\} \text{ such that } x_Q = \bot,\ x_P \ne \bot \text{ and } a_i(x_1 \ldots x_{r_i}) = \bot$$
$$= 1 \text{ otherwise.}$$

Clearly the $(a_i^{\#})$ and $(a_i^{\flat})$ are monotonic since we assume the (a_i) are computable. For any function $g: D^r \rightarrow D$ we will write[3]

$$g^{\#},\ g^{\flat}: 2^r \rightarrow 2$$

to denote the functions constructed from g by the above technique.

3.5 Example

Here we use the standard meaning for IF as the 3 argument sequential conditional; and PLUS as the usual (strict) operation on integers:

$$IF^{\#}(p, x, y) = p \wedge (x \vee y)$$
$$IF^{\flat}(p, x, y) = p \wedge x \wedge y$$
$$PLUS^{\#}(x, y) = x \wedge y$$
$$PLUS^{\flat}(x, y) = x \wedge y$$

It is useful to observe that these equations can be read in English to help understanding: the first one (for $IF^{\#}$ and $IF^{\flat}$) reads

IF(p,x,y) needs to evaluate both p AND at least one of x OR y.
IF(p,x,y) terminates if p AND x AND y do.

We can also cope with parallelism, for example

$$PIF^{\#}(p, x, y) = (p \vee x) \wedge (p \vee y) \wedge (x \vee y)$$

where

$$PIF(p,x,y) = x \text{ if } p = TRUE$$
$$= y \text{ if } p = FALSE$$
$$= x \text{ if } x = y$$
$$= \bot \text{ otherwise.}$$

This approach of identifying all non-undefined values can be

---

[3]Here 2^r means 2 X ... X 2.

further justified by noting that any two partially correct evaluation mechanisms (those that give the same result when both terminate) are weakly equivalent (ie $LUB(E_1,E_2)$ exists where the E_i are the results from the two evaluation mechanisms) and hence it is only necessary to discover places where undefinedness can creep in. In passing we note that this point is still relevant in higher order languages since in a well typed language with flat base domains the universe of discourse is D_n for some n where

D_0 is flat and

$D_{i+1} = D_i + (D_i \rightarrow D_i)$

Note (see section on non-discrete domains) that allowing D to be non-discrete might mean that we fail to obtain a very close bound here without more machinery.

$I^\#$ and $I^\flat$ are (non-comparable) interpretations abstracting I in the sense of Cousot and Cousot[9]. However here we use the two non-standard interpretations to "sandwich" the standard interpretation and thus it is important to note that one of the interpretations is "upside-down" relative to the above work.

We naturally define $f^\#$ and $f^\flat$ corresponding to the F_i as the least fixpoints of their defining equations in S under the interpretations $I^\#$ and $I^\flat$.

Let E, $E^\#$, $E^\flat$ be respectively the denotation functions for terms under I, $I^\#$, $I^\flat$. Then for all terms e (possibly with free variables) we can associate functions

$E[\![e]\!]: D^K \rightarrow D; \quad E^\#[\![e]\!], E^\flat[\![e]\!]: 2^K \rightarrow 2$

where K is a set containing all the free variables of e.

We have that

$$E^{\flat}[\![e]\!] \leq (E[\![e]\!])^{\flat} \leq (E[\![e]\!])^{\#} \leq E^{\#}[\![e]\!]$$

for all terms e, the centre inequality reducing to an equality if e has no free variables. This is a consequence of the general theory of abstract interpretation developed in chapter 2, but can also be shown by a simple "depth of computation" induction left to the reader. The outermost inequalities reduce to equalities if e is of the form $A_i(X_1 \ldots X_{r_i})$.

This result enables us to deduce that the definition of # and $\flat$ on the A_i extends to the F_i to give useful information on termination in I. The result is, for all partitions [P,Q] of $\{1,2 \ldots k_i\}$,

$F_i^{\#}(0/Q,\ 1/P) = 0$ implies
for all (x_j) such that $x_Q = \perp$ we have
$F_i(x_1 \ldots x_{k_i}) = \perp$

and

$F_i^{\flat}(0/Q,\ 1/P) = 1$ implies
for all (x_j) such that $x_P \neq \perp$ we have
$F_i(x_1 \ldots x_{k_i}) \neq \perp$

Note that we lose the half of the if and only if of the definition - this is due to the operation of composition rather than recursion, for example take

$$e = [\![\text{IF true THEN} \perp \text{ELSE } 91]\!]$$

which gives

$$E^{\#}[\![e]\!] = 1$$

in spite of the fact that $E[\![e]\!] = \perp$.

Now these are exactly the two conditions required for the detection of situations where call-by-need may be optimised to

call-by-value. The first gives us conditions on a function such that (some of) its formal parameters may uniformly over calls be evaluated before evaluation of the function body and the second gives us conditions on actual parameters which may be evaluated prior to calling uniformly over the head function symbol. We now consider these remarks in more detail with examples:

A condition for the actual parameter e_i associated to formal parameter x_i in a call $F(e_1 \ldots e_k)$ to be safely (ie without disturbing the meaning of the call - see Vuillemin[50]) evaluated before calling F is precisely that $F(x_1 \ldots x_k)$ is undefined whenever x_i is. Taking $Q = \{i\}$ in the above equation for $F^{\#}$ gives us a useful sufficiency condition for this to hold.

Similarly, to illustrate the use of $F^{\flat}$, suppose we have the following equation:

$$F(x,y) = G(x, y+1) + y$$

Consideration of $F^{\#}$ in the above manner (using the fact that

$$+^{\#}(x,y) = x \wedge y$$

in the usual interpretation of +) enables us to deduce that y may be passed by value to F. Now this fact means that $y \neq \bot$ in the body of F and correspondingly there we have that

$$E^{\flat}[\![y]\!] = 1.$$

Now we use the fact that (giving + its standard meaning)

$$+^{\flat}(x,y) = x \wedge y$$

and hence that

$$E^{\flat}[\![(y+1)]\!] = 1 \quad \text{since} \quad E^{\flat}[\![1]\!] = 1$$

This shows that the second parameter to G in the <u>call</u> $[\![G(x, y+1)]\!]$ always terminates and hence in implementation terms we

may choose to evaluate y+1 prior to the call and give G an evaluated call-by-need thunk rather than the standard unevaluated thunk to be evaluated on its first reference, without disturbing the semantics of the call. A further optimisation is that, if all calls to G have the above property then we know that $x_2 \neq \perp$ in <body> where

$$G(x_1, x_2) = body$$

and accordingly that x_2 may be classed as a value parameter to G. So, having established this we then have

$$E^{\#}[\![x_2]\!] = E^{\flat}[\![x_2]\!] = 1.$$

See also the section on transforming programs to use call-by-value below.

To derive solutions for the $f_i^{\#}$ and $f_i^{\flat}$ which are fixpoints of the systems $\langle S, I^{\#}\rangle$ and $\langle S, I^{\flat}\rangle$ we develop the following theory: (the aim is not to derive solutions by evaluation but rather by examination of their textual definition by forming

$$\lim T^i(\mathrm{BOTTOM})$$

where T is the functional to be defined below).

Define L by:

$$L = (2^{k_1} \rightarrow 2) \times (2^{k_2} \rightarrow 2) \times \ldots \times (2^{k_n} \rightarrow 2)$$

The space L has a natural lattice structure defined componentwise by

$$(p1, \ldots pn) \leq (q1, \ldots qn) \text{ if and only if}$$
$$(pi \mathbin{\&} \sim qi \text{ is identically zero}; \ (1 \leq i \leq n))$$

Now define T, a transformation on L by

$$T: (H_1 \ldots H_n) \rightarrow (H_1' \ldots H_n')$$

where

$$H_i'(x_1, \ldots x_{k_i}) = U_i[H_j/F_j;\ 1 \leq j \leq n;\ a^{\#}/A]$$

Defining

$$\text{BOTTOM} = (\lambda\, x_1 \ldots x_{k_1} \,.\, 0, \quad \lambda\, x_1 \ldots x_{k_2} \,.\, 0, \quad \ldots\ldots \quad \lambda\, x_1 \ldots x_{k_n} \,.\, 0)$$

and TOP = BOTTOM[1/0]

gives the top and bottom of the lattice L, respectively.

The sequence

BOTTOM, T(BOTTOM), T(T(BOTTOM)) ....

gives Kleene's ascending chain (AKC) on the finite lattice L. Hence all these terms are the same from some point onwards with limit value T^*(BOTTOM) say. Define T^*(TOP) similarly. Now by construction T^*(BOTTOM) and T^*(TOP) are fixpoints of $\langle S, I^{\#}\rangle$ with all other fixpoints between these two. The fixpoints of $\langle S, I^{\flat}\rangle$ are similarly defined.

Note now a couple of interesting points;

1. T^*(TOP) and T^*(BOTTOM) are in general distinct
2. Not all points such that $T^*(\text{BOTTOM}) \leq X \leq T^*(\text{TOP})$ are fixpoints of $\langle S, I^{\#}\rangle$

For proof consider

F(x,y,z) = IF x=0 THEN y*z ELSE F(x-1,z,y)

This gives

$$f^{\#}(x,y,z) = x \wedge (y \wedge z \vee f^{\#}(x,z,y))$$

and hence

$$T^*(\text{BOTTOM})(x,y,z) = x \wedge y \wedge z$$
$$T^*(\text{TOP})(x,y,z) = x$$

also

$$H(x,y,z) = x \wedge y$$

is between T^*(TOP) and T^*(BOTTOM) but is not a fixpoint.

The difference between the modes of parameter passing implied by T^*(BOTTOM) and T^*(TOP) is merely the difference in how the calculation proceeds in the evaluation of F(-1, $\bot$, 0); the first case implying passing (x,y,z) by value and the second just (x). In the call-by-value (for x,y,z) manner F is $\bot$ initially (upon evaluation of the value parameter $\bot$), but in call-by-need (for y,z) $\bot$ is never referenced, however the evaluator loops since the termination condition x=0 is never true. This corresponds to the inductive argument that if F is to terminate then the second argument in the initial call must be evaluated and its evaluation terminate. Thus we see that the fact that T has more than one fixpoint allows the system to be undefined in more than one way, but of course any two undefined values are indistinguishable (except by looking at the internal computation history), and hence the minimal fixpoint of T gives a valid mode of evaluation of parameters. In fact it follows (Vuillemin [50]) that any point above the minimal fixpoint defines a mode of evaluation which gives the correct result but there may be differences in the way undefined results are achieved. (Ie which particular infinite computation the system pursues.)

The existence of points (like H in the above example) which are above the minimal fixpoint (and so define safe evaluation strategies) but which are not themselves fixpoints is now explained:

The fixpoints of T correspond to the "consistent" modes of evaluation in the following sense:

A mode of evaluation is consistent if it is safe and no argument which is passed by need to a function will inevitably (after a bounded number of further passing by need) be evaluated.

To return to the case of H we can see that it is not a consistent point of T, and so cannot define a sensible mode of evaluation of parameters for F.

The standard proof that F (as above) terminates only if it references its second and third arguments is based on induction on the computation path. Our # functions, however, has the induction 'built into' the non-standard denotation $F^{\#}$ and so the proof merely consists of case analysis to see how 0 (= $\perp^{\#}$) can propagate.

3.6 Pragmatics

For use of the theory above in an algorithm the iteration produced is refined to be both more convenient and more rapidly convergent.

Define

$$Z(H) = Z_n(Z_{n-1}(\ldots (Z_1(H))\ldots))$$

where

$$Z_i(H_1 \ldots H_n) = (H_1 \ldots H_{i-1}, H', H_{i+1} \ldots H_n)$$

where

$$H'(x_1 \ldots x_{k_i}) = U_i[H_j/F_j;\ 1 \le j \le n;\ a^{\#}/A]$$

Note that Z (like T) is monotonic since it is the result of tupling and composing monotonic functions. We prove that $Z^*(\text{BOTTOM}) = T^*(\text{BOTTOM})$ to show that Z and T give the same result. Z is also

more convenient for implementing the iteration as it can be written as n single assignments in a loop rather than the one n-way multiple assignment required by T.

Further improvement in the speed may be effected by the following technique: firstly associate the call-structure graph with the function definitions (the call-structure graph is the directed graph obtained by considering function names as vertices and having an edge from f to g if and only if f contains a call to g in its body). Now partition this graph into its strongly connected components; giving a directed acyclic quotient graph; the strongly connected subgraphs can be analysed by the use of the Z (or T) iteration and the quotient graph is trivial to analyse - we flatten its partial order into a total order and analyse the strongly connected subgraphs according to this order.

A program has been written by the author (in LISP) to implement the above algorithm. A sample run is given below for a simple example and the system has been used on a text-formatter written by Martin Feather in NPL [5] without knowledge of this system. NPL normally has a call-by-value semantics and as a guide to the utility of the system, 132 of the 188 parameters in the paginator were detected as being safely passable by value upon assuming the program should conform to call-by-need semantics (there were 136 functions covering some 1100 lines of code, the system detecting that 93 of them were strict).

3.7 Transforming programs to use call-by-value

The outstanding cases where the system did not detect call-by-value were due to the following form of recursion in which we test one parameter to give a 'default' value or embark upon a recursive call:

```
LET mult(x,y) = IF x=0 THEN 0 ELSE mult(x-1,y)+y
```

the trouble about this case being that it is impossible (without further knowledge) to discover whether the user intended mult(0, $\perp$) to give 0 or $\perp$ - the call-by-need semantics indicate 0 and so y cannot be passed by value without extra knowledge. Of course for any particular call $\not\vdash$ may be used to detect if the actual parameter terminates and hence optimise the call.

The rest of this section suggests a method by which a program transformation system (for example Burstall and Darlington's fold/unfold method [6]) might be driven in order to transform out such non-strict functions by replacing them with strict functions and the basic non-strict conditional function (which is well known to compile and interpret efficiently).

Note that the "ELSE" branch of the above conditional expression satisfies

$$E^{\#}[\![\text{mult}(x-1,y) + y]\!] \;=\; x \wedge y$$

and hence is strict. So we can replace all calls $\text{mult}(e_1,e_2)$ with

$$\text{IF } e_1\text{=0 THEN 0 ELSE mult1}(e_1,e_2) \quad \text{WHERE mult1}(x, y) = \text{mult}(x-1, y) + y$$

and compile all calls to mult1 using call-by-value. But now a priori all calls to mult have the property that the second actual parameter must terminate (if it does not then neither can mult1 by

considering $\flat$). Hence mult can also be treated as a strict function and compiled appropriately.

We can actually do rather better that this by unfolding the call to mult in mult1 and refolding to use the definition of mult1 to get

```
mult1(x,y)  =  (IF x-1 = 0 THEN 0 ELSE mult1(x-1,y)) + y
```

to obtain a strict version of mult to replace the original non-strict version at the expense of doing the test before calling mult. This cost is significantly cheaper than the cost of merely setting up the closure for the second argument for mult.

I call the above technique rotational refolding of the function mult. This has an intuitive meaning seen by noting that the infinite tree representation for mult has alternate '+' and 'IF' nodes in its infinite backbone. Then the definition of mult1 is just obtained by taking a different ('+' instead of 'IF') starting point for the folding into finite form. The proof of strong correctness for this type of fold/unfold is much easier than the general case.

This idea can be extended to replace *all* uses of call-by-need by call-by-value by the use of appropriate conditionals, and this is the subject of chapter 4.

The following short insert gives a sample run of the implementation of the ideas given above.

```
.R VALARG

*(DEF FACT1 (X)
*     (IF (EQ X 0) 1 (TIMES X (FACT1 (SUB1 X)))))
FACT1

*(DEF FACT2 (X Y)
*     (IF (EQ X 0) Y (FACT2 (SUB1 X) (TIMES X Y))))
FACT2

*(DEF G (X Y Z) (IF X (PLUS Y Z) (DIFFERENCE Y Z)))
G

*(DEF H (X) 3)
H

*(DEF UNDEF (X) (IF (EQ X 0) (UNDEF X) (UNDEF (SUB1 X))))
UNDEF

*(DEF MY-IF (B X Y) (IF B X Y))
MY-IF

*(START) {; see if it all works}
MY-IF : Args (1) may be passed by value
UNDEF : *** totally undefined
H :     *** independent of args
G :     Args (1) (2) (3) may be passed by value
FACT2 : Args (1) (2) may be passed by value
FACT1 : Args (1) may be passed by value

(2 ITERATIONS)
```

3.8 Non-discrete domains

Here we will consider the problems caused by trying to extend the above work to a lazy evaluation system (see for example [15, 21]). In a call-by-need system an expression is either fully evaluated or a fully unevaluated suspension (closure). This corresponds to the set of values a variable may take being an element of a flat (or discrete) domain whose elements, x and y, satisfy

$$x \leq y \iff x = y \text{ OR } x = \bot.$$

However in a lazy evaluator a term can be evaluated to give a CONS node (say) without evaluation of its sub-terms, which will be evaluated when required for printing of deciding program flow. This implies the underlying data domain is not flat which gives us some problems since, using the above notation we get

$CONS^{\#}(x,y) = CONS^{\flat}(x,y) = 1$

$Hd^{\#}(x) = x$

$Hd^{\flat}(x) = 0.$

This unfortunately gives us a very bad bound on definedness and we now need to have some knowledge of list structures as our homomorphic image, instead of just {0,1}, in order to deduce those substructures whose evaluation can be safely moved.

Suitable further research would be to examine the possibility of using the notion of regular trees to approximate the limits of the (possibly infinite) Kleene sequences in the obvious image domain

D where $D = 2 + D \times D$

to tackle this problem.

Since this work was published Jones[27] has shown how to extend the ideas given here to the lambda-calculus by considering the states processed by the mathematical interpreter given by Plotkin[40].

3.9 Discussion of runtime errors in applicative languages.

This section (which is of the nature of an appendix to this chapter) suggests that the best way to handle errors from system functions is by returning special 'error' values. This method has the great advantage of preserving referential transparency and allows code transformations (such as the call-by-need to

call-by-value transformation discussed here) without changing the semantics of the language.

Consider the error which results when some system function is called with argument vector out of the defined range - eg the "division by zero" error from evaluating 1/0. In traditional languages this usually leads to a trap, often at the hardware level and possibly a jumpout to a user provided exit routine to diagnose and correct the error. Indeed in that case this often seems the most appropriate action to take.

However such jumpout action is far removed from the spirit of applicative languages and can destroy the referential transparency which they otherwise possess. Similarly operators which are normally commutative can lose this property and the order of evaluation becomes visible to the user. For example if A and B are (closed) terms whose evaluations lead to distinct errors, then the programs A+B and B+A may yield different results. Note that this last point is important in the system described herein since we want to be able to move a calculation without changing the semantics.

As an alternative the following scheme is much more attractive: Firstly extend the universe of discourse, D, with error elements

{error1, error2, error3 .... }

which are produced by the failure of system (and possibly user) functions. These objects should be treated as "first class citizens" so that evaluation of

[1/0, Hd(NIL), 3+5]

will result in

    [Error: division by zero, Error: Hd of NIL, 8]

as output, rather than a jumpout interrupt so beloved by operating system designers. Note that this scheme is also much more suited to systems with more than one processor.

Another benefit of this scheme is that, when taken to its logical conclusion, it leads to a backtrace of the error being built up automatically. Eg:

    1/0 + 3

might result in

    Error: Arg for PLUS not number: {error} + 3
    Error: Division by zero: 1/0

being printed in a system with an appropriate print program which knows about error objects.

Note that some system functions would allow error objects as parameters which do not change the form of the result. Eg: we would want

    CONS(1/0, CONS(5+4,NIL))

to print as

    [Error: division by zero, 9]

rather than CONS giving an error result. This scheme would probably find favour amongst users who often find systems only allow one error, or upon an error 'correct' it badly so that the output consists of many error reports from one error. Providing the function ISERROR(x) to test whether x evaluates to an error value and if so return some descriptor of the error (possibly a string) would enable provision of the ML [17] failure catching

mechanism.

Finally note that the error caused by non-halting programs is a much nastier object since it is impossible to test for in a uniform manner, and even though we may detect it in certain cases (eg memo-functions may detect a recursive call with the same arguments whilst still evaluating for those arguments) we must not give it the high status of ordinary run-time errors but produce a special error value, {no-halt} say, which cannot be tested for (ie ISERROR(no-halt) = no-halt) to guarantee uniform treatment of $\perp$. By this process we may sometimes anticipate an Operating System time-out without the waste of waiting for it to happen and, in this case, do something else (like evaluating the next expression in the input stream).

Chapter 4: Call-by-Need = Call-by-Value + Conditional

4.1 Introduction

This work demonstrates that a large class of programs designed to be run on a call-by-name interpreter I_n can be simply and effectively transformed into strongly equivalent ones for a call-by-value interpreter I_v in which all functions (including system functions) except the distinguished conditional function are strict.

This class of programs includes all those written in first order applicative languages with sequential base functions (in particular those with strict non-conditional base functions).

The results presented herein can be seen to generalise the results of de Roever [43].

This result has both practical and theoretical importance in that it provides for an alternative to closures (thunks) for implementing call-by-need or call-by-name, and in that it relates the two computation rules in terms of strong equivalence to enable the carrying across of proof techniques. The definition of sequentiality also seems to be of more general use.

4.2 Overview

The overall structure can be visualised as four separate stages:

Firstly we make the observation that a system of call-by-need equations is strongly equivalent to one of the possible computational paths of a non-deterministic similar system. A similar equivalence was also given by de Roever[43].

Secondly we derive a set of 'oracles' for this non-deterministic system which enable us to predict the computation path by insertion of tests (Conditionals). However to do this we must add extra elements to our domain and extend all base functions to cover them.

Thirdly we note that the extra elements introduced in the previous paragraph may be mapped into any set of distinct 'atoms' already present in the original system by using a form of overloading.

Finally we discuss the computational costs of this technique, since it may cause an exponential increase in the size of the program (but not of the running time), however we present arguments (as in chapter 3) to suggest that the actual increase in size is not so large and probably corresponds to less code at current machine level. See section 4.7.

The difficulties involved in the proof of correctness are due to the requirement to prove equivalence between two different program schemes under differing (operational) interpretations, and as such we must adopt a rather indirect technique of showing equivalence for the "before" and "after" versions of calls.

4.3 Basic Definitions

Let $\{F_i;\ 1 \leq i \leq n\}$ be a set of uninterpreted function symbols, with arity k_i; $\{A_i\}$ be base function symbols, with arity r_i; and $\{X_i\}$ be a countable set of individual parameters.

Consider the equations

$$\{F_i(X_1 \ldots X_{k_i}) = U_i;\quad 1 \leq i \leq n\}$$

with the U_i members of WFF(k_i) where the WFF(p) are the set of well formed terms constructed from

$\{A_i; F_j; X_1 \ldots X_p\}$

These equations provide a functional environment for the evaluation of terms from WFF(0) under a given interpretation. However, for definiteness, we will assume that k_1=0 and the 'program' consists of evaluating

$[\![F_1()]\!]$

in this environment.

We will take a domain D and functions $\{a_i\}$ $\{a_i:D^{k_i} \rightarrow D\}$ as the standard semantics of the above equations. This naturally defines a function $Eval_D$ giving meanings to terms. Currently we will also assume D is flat.

We will also use the annotation ':value' on actual parameters to indicate a particular parameter should be evaluated prior to the call of its enclosing function. Similarly ':need' will be used to clarify the default case of call-by-need. The idea of annotations as a means of describing how something is to be done dates from the Algol60 report. Schwarz explores their use for specifying evaluation mechanisms for applicative languages in [44].

We will adopt the standard semantic practice of using $[\![\ldots]\!]$ to enclose program text. Furthermore we will use the notation

$[\![F(u_i\text{:need}; u_j\text{:value}; i \in I; j \in J)]\!]$

or

$[\![F(u_i\text{:need}/I, u_j\text{:value}/J)]\!]$

to stand for

$[\![F(u_1 \ldots u_k)]\!]$

annotated with u_j:value if j is a member of J. This notation will be extended to allow us to write

$[\![F(u_{j_1}/J_1 \ldots u_{j_n}/J_n)]\!]$

to give us a named, rather than positional parameter association for disjoint subsets $J_1 \ldots J_n$ whose union is {1 ... arity(F)}.

4.3.1 Conventions

In the following '?' will be used to denote an arbitrary (but unspecified) non-$\perp$ value of D. The value will not be used in the computation, but is used to simplify functionality considerations.

We will use the following conventions to simplify the formalism and reduce the explicit indication of set membership:

- d_i to range over D
- e_i to range over E
- u_i and v_i to range over terms in any WFF(j)

We will also admit the syntactic sugar of using

```
[[select u from
        u1: v1
        u2: v2
        ..: ..
        uk: vk else w]]
```

to stand for

```
[[if u=u1 then v1
  elseif u=u2 then v2
  .. .. ..
  elseif u=uk then vk
  else w]]
```

We here assume that select is not present in our original language. This is merely a technical convenience to ensure that we are talking about objects introduced by an earlier transformation.

4.3.2 Operational Semantics

Here we give a (very) brief introduction to operational semantics which will be used to justify our transformations. As we indicate elsewhere it would be preferable to use denotational semantics, but the concept of sequentiality which we require does not seem to be easily accessible there.

Operational semantics specify the result of a computation by repeatedly performing re-write rules until reduction to a constant occurs. Excellent descriptions are given by Plotkin [40, 42]. Here we will adopt the notation of writing $\Rightarrow$ for the "re-writes in a single step to give" relation. For example, if $\underline{n}$ is a numeral and n the corresponding element of the domain of numbers, we may write

$$[\![\underline{n}]\!] \Rightarrow n.$$

This is true without any pre-conditions.

We will write $\Rightarrow^*$ for the transitive closure of the $\Rightarrow$ relation and proceed to define the evaluation of larger terms in the following manner, exemplified for '+'.

$$\frac{[\![e_1]\!] \Rightarrow^* n_1, \quad [\![e_2]\!] \Rightarrow^* n_2}{[\![e_1+e_2]\!] \Rightarrow^* n_1+n_2}$$

4.3.3 Definition of Sequentiality

We define a base (system) function symbol A_i to be sequential under an interpretation if we can write the semantics a_i for A_i operationally as

$$\frac{\mathrm{Const}(i)}{[\![A_i(u_1 \ \ldots \ u_{k_i})]\!] \Rightarrow [\![C(i)]\!]}$$

$$\frac{[\![u_p]\!] \Rightarrow^* d, \quad \sim Const(i)}{[\![A_i(u_1 \ldots u_{k_i})]\!] \Rightarrow^* [\![A_j(u_1 \ldots u_{p-1}, u_{p+1} \ldots u_{k_i})]\!]}$$

with A_j sequential

where $j = N(i,d)$ and $p = P(i)$

for some functions

N: Int X D $\rightarrow$ Int

P: Int $\rightarrow$ Int

C: Int $\rightarrow$ D

and some predicate Const. This simply says that the semantics of a function shall be expressible in the form: If a function does not require to evaluate any of its parameters then it is constant; otherwise evaluate the parameter required first (depending on i), and call a new function (dependent on i and the evaluated parameter) with the remaining unevaluated arguments. Thus we could show that '+' is sequential by showing that its semantics can be written in the form (roughly)

$$\frac{[\![e_1]\!] \Rightarrow^* n_1, \quad [\![e_2]\!] \Rightarrow^* n_2}{[\![e_1+e_2]\!] \Rightarrow^* ADD_{n_1}[\![e_2]\!] \Rightarrow^* n_1+n_2.}$$

Thus the evaluation of (our program) $[\![5+3]\!]$ would proceed via $ADD_5[\![3]\!]$ to 5+3 (in our mathematics) which is 8.

We will assume that this operational definition of the semantics agrees with the denotational version given in section 4.3, and feel free to use $Eval_D$ to refer to either.

Note that here we assume that the A_i include members whose interpretations a_i correspond to all possible partial applications of members of A_i. Thus if '+' and '1' are present in the A_i then

we would require that 'ADD1' given by

ADD1(x) = x+1

was present too. This does not reduce the generality since we allow the A_i to be an infinite set (although we can only use a finite number in our program).

This definition seems to be equivalent to the one given by Milner in [33].

4.4 Method

Let C = $[\![F(u_1 \ldots u_k)]\!]$ be a call occurring in the U_i.

We write this as

$[\![F(u_i\text{:need};\ i \in \{1,2 \ldots k\})]\!]$

showing that all arguments have need (non-strict) semantics.

We now note that one of the following must take place on evaluation of the call: (this depends on the assumption of sequentiality)

1. F evaluates (actual) parameter u_j first ($1 \leq j \leq k$)

2. F returns without evaluating any u_i

3. F computes forever without evaluating any of the u_i

Actually, without extra difficulty, we may allow that the order of evaluation of parameters may not only depend on the function, F, and the previously evaluated parameters, but also on the textual form of unevaluated parameters.

This indicates that if we have an oracle F^*, say, (a nullary function) which for the above cases respectively

1. returns j

2. returns 0

3. computes forever

then the call is strongly equivalent to

$$[\![\underline{select}\ F^*()\ \underline{from}\ 1: F(u_1{:}value,\ u_p{:}need;\ p \in \{1...k\}-\{1\})$$
$$..\ ...$$
$$k: F(u_k{:}value,\ u_p{:}need;\ p \in \{1...k\}-\{k\})$$
$$\underline{else}\ F(?{:}value\ ...\ ?{:}value)]\!]$$

where the ?'s stand for any non-$\perp$ value from D. The reason for the use of the '?'s here in the <u>else</u> clause (invoked when the oracle returns 0) is that their values will not be used in the subsequent computation due to the assumed truth of the oracle.

We now show that the above technique can be inductively extended to reduce all the parameters of a call to :value ones, and also how to effectively compute the F^*.

<u>4.4.1 Total reduction to call-by-value</u>

We observe that the above technique is just a special case of the following equivalence:

$$[\![F(u_i{:}need,\ u_j{:}value,\ j \in J;\ i \in I)]\!]$$

is equivalent to

$$[\![\underline{select}\ F_I^*(u_j{:}value;\ j \in J)\ \underline{from}$$
$$..\ ...$$
$$p: F(u_j{:}value,\ u_i{:}need;\ j \in J \cup \{p\},\ i \in I-\{p\})$$
$$..\ ...$$
$$\underline{else}\ F(u_j{:}value,\ ?/I;\ j \in J)]\!]$$

with p varying through I for non-empty I, giving an inductive step for reducing the number of :need parameters. The base case of

$$[\![F(u_1{:}value\ ...\ u_k{:}value)]\!]$$

corresponding to I={}, is already of the required call-by-value

form. Thus we have produced a new program schema

$$G_i(X_1 \ldots X_{k_i}) = V_i$$

from our $\{F_i\}$ schema, by replacing all the F_i in the above terms by G_i. The F_i and G_i schemata will be strongly equivalent under call-by-value and call-by-need interpretations respectively, subject to our defining the oracles F^*_I for all F in $\{F_i\}$ and for all subsets I of $\{1 \ldots k\}$ required by the above process.

The proof of this is given is section 4.4.8 after we have defined the oracles.

Note that this process will (wastefully) re-write the conditional function as a select, however here we are concerned with correctness - efficiency will be considered in section 4.7.

4.4.2 Production of the oracles F^*_I

We will produce the oracles in two stages; first showing that we can introduce oracle-like objects $F^{\#}_I$ at the expense of extending the domain of discourse, D, (and of course also the base function definitions) and then further showing that this extension can be ignored at run time by using a form of overloading to produce the F^*_I (see section 4.5).

4.4.3 The $F^{\#}_I$ exhibited

Consider a call

$$C = [\![F(u_i\text{:need}, u_j\text{:value}; i \in I, j \in J)]\!]$$

which produces, as an intermediate inductive call in the above method

$$[\![\underline{\text{select}}\ F^*_I(u_j\text{:value})\ \underline{\text{from}}\ \ldots]\!]$$

We now define a countable set of new elements

$$E = \{\bot_1, \bot_2 \ldots\}$$

We will only use a bounded number of these elements - the reason for the names will become clearer later.

We will now define

$F_I^{\#}(u_j{:}value) \;=\; F(u_j{:}value,\ \bot_i/I)$

which requires us to extend the semantics $\{a_i\}$ of the $\{A_i\}$ from D to D+E in such a manner to model the computational effects of $\bot$. For example, we want to model the statement that

F(x,y) = e

<u>requires</u> x to be evaluated, by the equation

$F(\bot,u) = \bot$

(modulo some discussion about the termination of F). Here the intention is that the $\bot_i$ will act as bombs which explode when used in a calculation, thus indicating which parameters are evaluated during the call. We will now define an interpretation, by giving its semantics, which will ensure that the $F_I^{\#}$ return $\bot_i$ to indicate that parameter i will be evaluated in the 'real' computation. Thus the $F_I^{\#}$ will be oracles for the extended domain D+E.

4.4.4 Operational extension of a_i

We augment the operational rules (see section 4.3.2 for definitions) for the base function semantics

$$\frac{Const(i)}{[\![A_i(u_1 \ldots u_{k_i})]\!] \;\Rightarrow\; [\![C(i)]\!]}$$

$$\frac{[\![u_p]\!] \;\Rightarrow^* \; d,\ \ \sim Const(i)}{[\![A_i(u_1 \ldots u_{k_i})]\!] \;\Rightarrow^*\; [\![A_j(u_1 \ldots u_{p-1}, u_{p+1} \ldots u_{k_i})]\!]}$$

with A_j sequential

where j = N(i,d) and p = P(i)

with

$$\frac{[\![u_p]\!] \Rightarrow^* [\![\bot_j]\!]}{[\![A_i(u_1 \ldots u_{k_i})]\!] \Rightarrow^* [\![\bot_j]\!]}$$

It is not necessary to define the effect of the $\bot_i$ on the F_i scheme since we can simply use the standard call-by-name substitution semantics as this is equivalent to call-by-need in applicative languages, and use:

$$[\![F_i(u_1 \ldots u_{k_i})]\!] \Rightarrow [\![U_i[u_1/X_1] \ldots [u_{k_i}/X_{k_i}]]\!]$$

4.4.5 Denotational extension of a_i

This section gives an alternative (denotational) definition to the previous one. It is not central to the work, however it does avoid some problems with the over-specification inherent in operational semantics.

Consider a map

$a: D^k \longrightarrow D$

which is the standard interpretation of a symbol, A, say.

We wish to extend this map to the sum domain D+E given by adding the elements of E to the domain D together with the coarsest compatible domain structure (ie the addition of only $\bot < x$ for all x in E, to the partial order). The extension has functionality

$a': (D+E)^k \longrightarrow (D+E)$

thus providing an alternative interpretation for A preserving the behaviour of a on D.

We define the extension componentwise by

$$\begin{aligned} a_1'(e_1, d_2 \ldots d_k) &= a(\bot, d_2 \ldots d_k) \text{ if } a(\bot, d_2 \ldots d_k) \neq \bot \\ &= \bot \text{ if } a(d_1, d_2 \ldots d_k) = \bot \ \forall\ d_1 \in D \\ &= e_1 \text{ otherwise} \end{aligned}$$

and similarly for the other parameters. This definition is monotonic because of the flatness of D and the sequentiality of the $\{a_i\}$. The proof of this is somewhat outside the scope of this chapter.

We can now define a' when a set I, of its parameters are members of E by

$$
\begin{aligned}
a'(e_i/I,\ d_j/J) \ = \ & \\
& \mathrm{JOIN}\ \{\mathrm{STRICT}[a'(e_i/I-\{k\},\ d_j/J,\ x/\{k\})]:\ k \in I\} \\
& \quad \text{if } |I| > 1 \\
& a_i'(e_i/I,\ d_j/J) \\
& \quad \text{if } I = \{i\} \text{ as above} \\
& a(d_j/J) \\
& \quad \text{if } I = \{\}
\end{aligned}
$$

where

$$
\begin{aligned}
\mathrm{STRICT}(f(x)) \ &= \ \mathrm{JOIN}\{f(x):\ x \in D-\{\bot\}\} \\
& \qquad\qquad \text{if } f(x) \in E,\ \forall\ x \in D-\{\bot\} \\
&= \ 0 \text{ otherwise}
\end{aligned}
$$

and

$$
\mathrm{JOIN}\{x_1 \ldots x_n\} \ = \ x_p \text{ for some } x_p \text{ in E.}
$$

There are several notes to be made on this definition. Firstly the definition of JOIN is well, if non-deterministically defined, since again flatness of D and sequentiality imply its argument is a non-empty set. The intention in this denotational definition is to avoid the over-specification of detail present in the operational version: for example consider the definition of '+' operationally as in section 4.3.3. We need to specify in which order the parameters to '+' are evaluated to provide operational semantics, however this is irrelevant to the program (in either its call-by-need or call-by-value form), although it does effect the internal flow of computation in the call-by-value version, since

the oracles have to report the order of examination of parameters. Of course it is reasonable to observe that the oracles could return subsets of the function parameter set which can be evaluated together, thus avoiding this problem and being more in spirit with chapter 3; however this complicates the above work which already has notational problems for no clear gain in flexibility. Also there are problems in doing this operationally. Thus the non-determinism in the definition of JOIN reflects the arbitrary choice of evaluation order in the operational definition of a strict function.

Note that this provides a reason for the names $\bot_i$ for elements of E: they model the behaviour of $\bot$ in a computable (continuous) way in that they model the way $\bot$ propagates through a program.

4.4.6 Definition of F^*_I

It is now obvious that we can define

$$F^*_I(u_j/J) = \text{Index}(F^\#_I(u_j/J))$$

where Index is the D+E disjoint sum extraction operation given by

$$\text{Index}(\bot) = \bot$$
$$\text{Index}(\bot_i) = i$$
$$\text{Index}(x) = 0 \text{ otherwise}$$

This defines the oracle F^*_I to use the original set of equations (F_i), however, we can now see that we can define

$$G^*_I(x_j/J) = \text{Index}(G(x_j/J,\ \bot_i/I))$$

thus giving us an oracle whose (internal) calculations are performed using call-by-value via the G_i schema.

4.4.7 Definition of the system oracles A^*_I

For a system function A_i we will derive the corresponding

oracles

$$A^*_{i,I}$$

directly from the (operational) semantics given in section 4.3.2.

We can define

$$A^*_{i,I}(x_1 \dots x_{r_i}) = 0 \quad \text{if} \quad \text{Const}(i).$$

This merely says that if A_i reduces directly to a constant without evaluating any of its parameters, then the corresponding $A^*_{i,I}$ should return 0 to indicate that this is the case.

On the other hand, if ~Const(i) we have given by the semantics $p = P(i)$ such that u_p is evaluated first in order to see how the computation progresses (that is which A_j, $j = N(i,\text{Eval}(u_p))$ is called on the remaining parameters). In this case we can see that an appropriate definition is

$$A^*_{i,I}(x_1 \dots x_{r_i}) = x_p \quad \text{if } p \in I$$

This case corresponds to evaluating a parameter which $A^*_{i,I}$ has been called to enquire about.

Otherwise x_p corresponds to a value of D (non-$\perp$ due to the use of call-by-value) and hence the operational re-writing rules produce a definition

$$A^*_{i,I}(x_1 \dots x_{r_i}) = A^*_{j,I'}(x_1 \dots x_{p-1}, x_{p+1} \dots x_{r_i})$$
$$\text{where } j = N(i,x_p)$$
$$\text{and } I' = \{k: k \in I, k<p\} \cup \{k-1: k \in I, k>p\}$$

This definition now casts some light on the reasoning behind our rather pedantic form of semantics given in section 4.3.2. For example, the reduction in the number of parameters during re-writing of a system function ensures that the above definition of $A^*_{i,I}$ is primitive recursive, and hence total.

The reason for requiring sequentiality of the base functions is that the oracular versions of the base functions must be provided for the use of the evaluator. This would mean that a non-sequential function, eg the "Parallel IF" function defined by

$$
\begin{aligned}
PIF(p,x,y) &= x \text{ if } p=true \\
&= y \text{ if } p=false \\
&= x \text{ if } x=y \\
&= \bot \text{ otherwise}
\end{aligned}
$$

would require an oracle to interpret the call

$$[\![PIF^{\#}_{\{1,2,3\}}(\bot_1, \bot_2, \bot_3)]\!]$$

in order to predict the actual parameter to PIF which will be evaluated first. This is clearly impossible. Huet and Levy provide more general and detailed argument in [25].

4.4.8 Proof of correctness

We will perform the proof of correctness by showing each of the one-step transformations given in section 4.4.1 produce a strongly equivalent result. Thus after a finite number of such transformations the overall result must be strongly equivalent to the original program. The reasoning given below for a function F applies equally well for system functions A_i and user functions F_i.

We wish to show that

$$Eval_D[\![F(u_i:need/I,\ u_j:value/J)]\!] =$$

$$
\begin{aligned}
Eval_{D+E}[\![& \underline{select}\ Index(F(u_j:value/J,\ \bot_i:value/I))\ \underline{from} \\
& \quad .. \ ... \\
& \quad p:\ F(u_j:value/J \cup \{p\},\ u_i:need/I-\{p\}) \\
& \quad .. \ ... \\
& \quad \underline{else}\ F(u_j:value/J,\ \bot:need/I)]\!]
\end{aligned}
$$

Note that here we use $\bot$'s in the else clause, rather than the

'?'s given in section 4.4.1. We do this to simplify the proof, which will show (case 3 below) that the '?' values are not involved in the calculation.

Firstly we will state a simple observation about the behaviours of $Eval_D$ and $Eval_{D+E}$.

Lemma:

$Eval_D$ and $Eval_{D+E}$ agree on terms not containing any $\perp_i$.

Proof:

Just consider the operational definition of $Eval_{D+E}$ on terms, noting that terms cannot produce $\perp_i$ unless they contain $\perp_i$.

Therefore this implies that

$Eval_D[\![F(u_i\text{:need}/I,\ u_j\text{:value}/J)]\!] \geq$

$Eval_{D+E}[\![$ select Index(F(u_j:value/J, $\perp_i$:value/I)) from
.. ...
p: F(u_j:value/J $\cup$ {p}, u_i:need/I-{p})
.. ...
else F(u_j:value/J, $\perp$:need/I)$]\!]$

where $\geq$ represents domain ordering. This is so since both

$Eval_D[\![F(u_j\text{:value}/J \cup \{p\},\ u_i\text{:need}/I-\{p\})]\!]$

and

$Eval_D[\![F(u_j\text{:value}/J,\ \perp\text{:need}/I)]\!]$

are dominated by

$Eval_D[\![F(u_j\text{:value}/J,\ u_i\text{:need}/I)]\!]$

(see Vuillemin's work [50]) and the calculation of the oracle for the select clause further reduces the result of the select clause in the $\geq$ ordering.

In order to show the converse we will trace the step-by-step evaluation of terms using $Eval_D$ and $Eval_{D+E}$. We require to show

the following consistency conditions on the oracle

1. $\text{Eval}_{D+E}[\![F(u_j\text{:value}/J,\ \bot_i\text{:value}/I)]\!] = \bot$
   $\Rightarrow \text{Eval}_D[\![F(u_j\text{:value}/J,\ u_i\text{:need}/I)]\!] = \bot$

2. $\text{Eval}_{D+E}[\![F(u_j\text{:value}/J,\ \bot_i\text{:value}/I)]\!] = \bot_p$
   $\Rightarrow \text{Eval}_D[\![F(u_j\text{:value}/J,\ u_i\text{:need}/I-\{p\},\ \bot/\{p\})]\!] = \bot$

3. $\text{Eval}_{D+E}[\![F(u_j\text{:value}/J,\ \bot_i\text{:value}/I)]\!] \neq \bot,\ \neq \bot_p$
   $\Rightarrow \text{Eval}_D[\![F(u_j\text{:value}/J,\ u_i\text{:need}/I)]\!] \neq \bot,\ \neq \bot_p$

The corresponding proofs are

1. Eval_{D+E} re-writes its argument infinitely, and Eval_D performs the same re-writings unless a $\bot_i$ is produced by Eval_{D+E}. However this would terminate the Eval_{D+E}. Contradiction.

2. Eval_D and Eval_{D+E} perform corresponding re-writes until Eval_{D+E} first produces $\bot_i$ as a parameter to be evaluated. At this point Eval_D has $\bot$ to evaluate. Thus Eval_D gives $\bot$.

3. Again consider performing the re-writes of Eval_D and Eval_{D+E} step-by-step. They both follow the same reduction sequence and hence, since Eval_{D+E} terminates without encountering a $\bot_i$, Eval_D will terminate without encountering a corresponding $\bot$. (Actually both calculations will produce the same non-$\bot$ value in D.)

4.5 Getting rid of the $\bot_i$

We would now like to remove the explicit tests necessary to determine whether an element is a member of D or E (usually called IsD and IsE) since in any practical implementation the extension to the base functions will be done by first testing (as in the operational definition above) if a parameter is a member of E, and if so taking special action. However these tests will likely be at

least as expensive as the "IsClosure" test implementation of call-by-need. For theoretical reasons it is also rather distasteful to add extra elements to our domain of discourse.

We wish to exhibit the definition of an oracle F_I^* which only uses the elements in D, rather that $F_I^\#$ which requires the $\perp_i$ to be added to the universe of discourse. We will use the integers {1 ... n} to model $\{\perp_1 \ldots \perp_n\}$ and use specific instances of F to ensure type security just like a type-checker would ensure that bit patterns representing objects in D are not used as bit patterns representing objects in E. Thus the code we produce must always ensure we always know whether an integer represents a member of D or E.

We will now exhibit F_I^* in terms of G, the call-by-value version of F produced in section 4.4.1 and given by

$[\![G(x_1 \ldots x_k) = u]\!]$

We will define corresponding versions, for all required subsets I of {1 ... k} by

$$[\![F_I^*(x_j/J) = G_I^*(x_j/J,\ i/I)$$
$$G_I^*(x_1 \ldots x_k) = u_I^*]\!]$$

with

$$u_I^* = P2_I[\![u]\!]$$

where $P2_I$ is given by the following transformation:

The effect on variables is to produce the corresponding integer if the variable corresponds to an oracular parameter, otherwise 0 to indicate that the evaluation does not require parameter evaluation.

$$P2_I[\![x_i]\!] = [\![x_i]\!] \quad \text{if } i \in I$$
$$= [\![0]\!] \text{ otherwise}$$

For function applications, both of user and system functions we define

$$P2_I[\![g(u_s/S)]\!] = T_I[\![g(u_s/S)]\!]\ \{\}\ S$$

where

$$T_I[\![g(u_s/S)]\!]\ J\ K$$
$$= [\![g^*_J(u_s/S-J)]\!] \quad \text{if } K = \{\} \qquad (1)$$
$$= [\![\underline{\text{if}}\ c=0\ \underline{\text{then}}\ v\ \underline{\text{else}}\ w]\!] \text{ otherwise}$$

where

$$p = \max(K)$$
$$c = P2_I[\![u_p]\!]$$
$$v = T_I[\![g(u_1 \ldots u_k)]\!]\ J\ (K-\{p\})$$
$$w = T_I[\![g(u_1 \ldots u_{p-1}, c, u_{p+1} \ldots u_k)]\!]\ (J \cup \{p\})\ (K-\{p\})$$

The intention here is that we scan through the arguments, S, of g, evaluating them in oracle context (as evaluated in the terms given by c) building up a set J of parameters to g which will need to be oracles, and then selecting the appropriate version g^*_J of g^*.

For a select clause it is necessary to choose the correct version of the oracle function since the construction given in section 4.4.1 assumed we could tell the difference between an oracular value and a 'real' value in D. We must also translate the consequents, in order that they perform their calculations in oracle context.

$$P2_I[\![\underline{\text{select}}\ g^*_J(u_i/S)\ \underline{\text{from}}\ 1:v_1 \ldots n:v_n\ \underline{\text{else}}\ v_0]\!]$$
$$= [\![\underline{\text{select}}\ w'\ \underline{\text{from}}\ 1:v'_1 \ldots n:v'_n\ \underline{\text{else}}\ v'_0]\!]$$

where

$$v'_i = P2_I[\![v_i]\!]$$
$$w' = T_I[\![g(u_i/S)]\!]\ J\ S$$

Note that the replacement of g^*_I by g in the definition of w' is not accidental in that line (1) in the definition of T_I re-supplies them.

Hence, the above shows that we can apply a form of overloading (generic definition) in which we define separate functions for different combination of parameter types, and simply return integers (as in section 4.4) indicating which parameter will be evaluated next (actually any set of k distinct objects will do for a k-adic call).

4.5.1 Proof of correctness of the overloading

The proof of the removing the $\perp_i$ in favour of objects already present in our language requires proof. However the details are tedious and not illuminating. Basically the proof is most easily factored into two stages. Firstly we change the $\perp_i$ into corresponding integers, but also add an extra (set) parameter, I, to each function explicitly, thus using

$$F^*(x_1 \ldots x_k, I)$$

instead of

$$F^*(x_1 \ldots x_k).$$

This enables us always to tell whether an integer represents an oracle value or a domain integer. The semantics of these two methods of representing disjoint sums can easily be seen to be equivalent.

Then, we observe that the type parameter, I, can only take on finitely many values, and hence we can produce versions of F^*

$$F^*_I(x_1 \ldots x_k)$$

for all possible I. Thus we have achieved our aim.

Of course in practice we hope not to behave in quite such a profligate manner.

4.6 Example

Consider applying this theory to a call

C = ⟦mult(u,v)⟧

where u and v are closed terms, in an environment given by the definition

⟦mult(x,y) = *if* x=0 *then* 0 *else* mult(x-1,y) + y⟧

We thus expand this call-by-need expression into a call C' with

$$
\begin{aligned}
C' \;=\; [\![& \underline{\text{select}}\ \text{mult}^*_{\{1,2\}}(\bot_1,\bot_2)\ \underline{\text{from}} \\
& \bot_1\text{: } (\underline{\text{if}}\ \text{mult}^*_{\{2\}}(u,\bot_2)=\bot_2\ \underline{\text{then}}\ \text{mult}(u,v) \\
& \qquad\qquad\qquad\qquad\quad \underline{\text{else}}\ \text{mult}(u,?)) \\
& \bot_2\text{: } (\underline{\text{if}}\ \text{mult}^*_{\{1\}}(\bot_1,v)=\bot_1\ \underline{\text{then}}\ \text{mult}(u,v) \\
& \qquad\qquad\qquad\qquad\quad \underline{\text{else}}\ \text{mult}(?,v)) \\
& \underline{\text{else}}\ \text{mult}(?,?)]\!]\text{:value}
\end{aligned}
$$

This can clearly lead to an exponential increase in the size of the code. However, using the techniques described in chapter 3, has the effect of removing the unnecessary tests from C'. (For example, we can see that the term ⟦u⟧ will always be evaluated first in the standard interpretation of functions and hence the *select* will always take the first branch.) This produces C" given by

$$
\begin{aligned}
C'' \;=\; [\![& \underline{\text{if}}\ \text{mult}^*_{\{2\}}(u,\bot_2)=\bot_2\ \underline{\text{then}}\ \text{mult}(u,v) \\
& \qquad\qquad\qquad\qquad\quad \underline{\text{else}}\ \text{mult}(u,?)]\!]
\end{aligned}
$$

and a new version of mult given by

⟦mult(x,y) = *if* x=0 *then* 0 *else* mult(x-1,y) + y⟧:value.

(Here there is no need to expand the recursive call of mult as in C since we may use y equally well as a '?' value, since y must be a non-$\bot$ value in call-by-value.) These definitions rely on the extension of the domain to incorporate the $\bot_i$ values.

We must now define the oracle $\text{mult}^*_{\{2\}}(x,y)$ in terms of mult.

Applying section 4.4.2 gives us

$$[\![mult^{*}_{\{2\}}(x,y) \quad = \quad \underline{if}\ x{=}0\ \underline{then}\ 0\ \underline{else}\ y]\!]$$

Thus we can replace the initial call C:need by C''', where

$$C''' \quad = \quad [\![\underline{if}\ u{=}0\ \underline{then}\ mult(u,?)\ \underline{else}\ mult(u,v)]\!]$$

which provides a general solution to the problem partially solved in section 3.7. Note that the above form for C''' suggests that the term u should be evaluated three times. Again, however, these will produce the same result and hence standard compiling techniques will produce only one evaluation.

We might at this point stop to look at the view, in that the above C''' represents the standard call-by-value version of the non-strict multiplication function.

The reader is strongly advised to work through this particular example ensuring he sees the justification behind each step - it is very easy to succumb to invalid optimisations. A program has been developed which performs a rather more sophisticated algorithm based on this work.

4.7 Computational Costs

As was hinted above, much of this translation technique runs the risk of exponentially increasing the size of a program (although I believe it can only increase the running time by a constant factor). However I now wish to present the argument that, for a large class of programs, the increase in space and time will be fairly small. Certainly it is the case that we write parameters to functions which _can_ possibly be evaluated, for example a dispatch routine which behaves like a conditional, and often we know that

certain parameters will be evaluated. However we do not write parameters which can never be evaluated. Hence the attitude of this work which can be summarised as partitioning parameters into two subsets, those which will, and those which might be evaluated; is justified on pragmatic grounds. We will also probably be quite happy if the increase in complexity is less than a factor of two or so, in that the overhead for compiling traditional call-by-need closures (or 'thunks') is quite considerable compared to call-by-value, both in the space for the code, and in the time taken to switch environments to evaluate a closure.

The following results were derived by analysing the cost of performing the ideas given here on a large (1100 line) applicative program written by Feather[13]. The program is an applicative text formatter (like ROFF or RUNOFF) as described in "Software Tools"[30]. The program size (for an abstract machine) grew from 12746 cells to 26632 cells upon applying the transformation. The computation speed (measured in abstract evaluator cycles for a standard input) was 3417 cycles for the original code under call-by-need and 4995 cycles for the call-by-value code resulting from the transformation. For some indication of efficiency, the original code took 3695 cycles to perform the same action under call-by-value. These figures concur with the suggestions made earlier that the cost is mainly of size rather than execution time, however we should note again that under current machine architecture it is quite possible that the call-by-value transformed code performs better than the original in both space and time. This is so because our abstract call-by-need cycles

represent more real machine cycles than our abstract call-by-value cycles.

One small point to be noted is that the above figures are derived from running on an interpreter which recognises multiple occurrences of a single expression (as can easily be produced by this work) and only evaluates them once. This is justified in that compilers usually perform some common sub-expression analysis.

4.8 Conclusions

I believe the above results show that the work presented is not only a pretty theoretical toy, analogous to "Static and Dynamic Binding Strategies have Equal Power"[20], but also provides a practical alternative to the traditional closure implementation of call-by-need.

However the scope of this work is somewhat restricted and it would be extremely useful to be able to extend it to a full "lazy CONS"[15, 21] language, from our simple call-by-need recursion equations. For example, consider the following simple program in a lazy-evaluation language:

```
let f(n) = n :: f(n+1)
in f(0)
```

(:: is an infix CONS). In a lazy evaluation scheme this program would print

```
[0,1,2,3,4,5,6 .....
```

without stopping. Unfortunately, in a more eager evaluation strategy (even call-by-need), this program would calculate the full result (ie the list of all numbers) before attempting to print it out.

It is desirable (again both for theoretical and practical reasons) to be able to transform this program into the following iterative version (the only non-applicative feature is the existence of a PRINT procedure which is only slightly non-functional):

```
let g(n) = PRINT(n); g(n+1)
in g(0)
```

This is an equivalent program, which can be evaluated quite safely using the most eager of evaluation strategies, call-by-value.

Chapter 5: On Introducing Destructive Operators into Applicative Programs

5.1 Abstract

In this chapter we study methods to introduce destructive operators into applicative programs. The work is based on the concept of "Abstract Interpretation" developed by Cousot & Cousot [8, 9] and generalises previous results of Schwarz [45, 46] and Pettorossi [38, 39]. It also provides a more semantically oriented framework for the work of Jones & Muchnick [28] which only applied to flowchart schemata.

The intention is not to produce a single method of introducing destructive operators, but rather to study a schema or class of methods, and hence we will need general correctness proofs.

We share Schwarz's and Pettorossi's attitude that destructive operators should be used for the means that they provide of optimising otherwise applicative programs, leaving their meaning unchanged, rather than sanctioning their use for causing side effects on other objects, which is often criticised as producing programs which are difficult to read and modify. (See for example Backus's Turing lecture [2]).

This work develops alternative semantics for programs modelling the standard "ad hoc" ideas of collections of CONS nodes being shared or unshared, but the semantic formulation simplifies proof rules. The general proof rules for shared data objects (for example Burstall [4]) can be difficult to apply.

5.2 Developments from previous work

The work of Schwarz [45] was based on an (operational) rewrite rule semantics whereas this chapter will attempt to develop the theory of sharing within a denotational framework. This has the advantages that

1. Fixpoint methods are much more easily discussed denotationally. Accordingly we are able to develop techniques here which exhibit as fixpoints certain properties which had to be supplied by the user in Schwarz's model.

2. Correctness proofs are very much easier denotationally in that we can apply the general abstract interpretation mechanism [8, 9].

We also extend the work of Pettorossi [38, 39] by considering the problem of incorporating destructive operators in situations where substructures may share. In essence this work handles DAGs (directed acyclic graphs) where Pettorossi's only handled trees.

Pettorossi's work was concerned with the problem of introducing destructive versions of system functions (like plus, times etc) which took their arguments by reference and wrote their results in one of the argument locations rather than creating a new location to hold the result. To this end he introduced a marking tuple associated with each system function call, with elements of the tuple indicating whether or not the corresponding parameter may be destroyed. The work was denotationally based and the best (most destructive) safe version could be seen as a fixpoint of a certain set of equations. However the work did not address the problems

involved with the partial degrees of sharing associated with list structures.

Schwarz's work, on the other hand, considered the list based formalism immediately, but via an operational style semantics based on treating all functions as specifying re-write rules. The operational formulation inhibited the view of certain properties as fixpoints and so the programmer had to provide sharing and destructiveness declarations for each function he provided. Another problem is that it is difficult to see how to give any form of correctness proof - Schwarz gives none. On the positive side, the concepts of structure usage given by non-standard interpretations E_{uses}, E_{exam}, E_{isol} are very similar to the ones we use in a denotational setting here.

We also give credit to Wegbreit [52] who suggested that one possible use for non-standard interpretations was to model storage allocation in the real world.

5.3 General Overview of the Development

First we introduce our langauge of recursion equations and its associated semantics, together with such auxiliary notions as an occurrence within an expression. We also discuss the most appropriate form for destructive operators.

Next we introduce two non-standard interpretations E_{uses} and E_{exam}. The former gives (upper bounds on) the set of CONS cells which the standard interpretation would build into the result of an expression and the latter indicates similarly which cells are

traversed in order to build this result. These two interpretations are then specialised into (non-standard) semantic functions USES-L and USES-R respectively. These are merely derived for their convenience of use over E_{uses} and E_{exam}.

We now introduce the notion of 'isolation classes' which represent the extent of sharing of a given structure. We then lift this idea to an interpretation E_{isol} which specifies the sharing properties of an expression in terms of the sharing properties of its free variables.

These results are then used to justify the validity of a small number of transformation rules which insert destructive operators into expressions without changing their semantics.

5.4 Formalism and General Ideas

We will define a language called LISP-D (D for destructive) which consists of a first order language of recursion equations in a LISP-like syntax based on the signature

- $\{A_i : i>0\}$ Atoms
- $\{X_i : i>0\}$ Variables
- $\{B_i : i>0\}$ Base functions (of arity r_i).
- $\{F_i : 0<i\leq n\}$ Defined functions (of arity k_i).

We will identify certain distinguished elements of the $\{B_i\}$ which will be written as

{CONS, CAR, CDR, ATOM, IF, FREE}

which are the standard list processing primitives for building (CONS), selecting (CAR, CDR), discriminating (ATOM), the three

parameter conditional (IF), together with an operator (FREE) for destroying list cells. We will use a single constructor function (CONS). The extension to multiple constructor functions poses no theoretical problems, though it does require a more complicated algorithm to analyse them. Similarly we will use a single destructive operator (FREE) and the motivation for this will be detailed in section 5.4.2.2. We will also define a language LISP-A which is identical to LISP-D except that the signature will not contain any destructive base functions and hence will be purely applicative. It is possible to give LISP-A a semantics which does not involve stores and locations, and this is discussed in section 5.12.3.

We choose the somewhat barbarous LISP syntax, rather than a higher level applicative language because LISP already has a well established set of names and intuitive concepts for destructive operators, and also because LISP is quite near to machine level which enforces us to make explicit certain choices glossed over in a purely applicative language. (For example the amount of sharing present and the method used for building objects with mainly constant data as discussed in the SUBST worked example in section 5.11.1.) We will discuss semantics in more detail later.

The aim throughout will be to treat system functions in as general a manner as possible in order to permit defined functions having similar properties to be treated similarly.

We now define a program scheme to be a set of equations

$\{(F_i\ X_1\ \ldots\ X_{k_i}) = U_i :\ 1 \leq i \leq n\}$

where the U_i are terms respecting the arities of the $\{F_i\}$ and $\{B_i\}$ and the free variables of U_i are contained in $\{X_1 \ldots X_{k_i}\}$.

We will follow standard applicative semantic practice and associate a standard semantics with this syntax by first choosing a domain D (including $\perp$, the undefined value, and ?, an error value) and then associating functions $b_i:D^{r_i} \rightarrow D$ with the B_i in the usual manner. This then induces meanings $f_i:D^{k_i} \rightarrow D$ for the F_i by the usual fixpoint method. However the wish to discuss destructive operators will mean that our denotations will actually have an extra store component (both as an argument and result) to model the side effects of a function thus:

$$f_i: D^* \times \text{Store} \rightarrow D \times \text{Store}$$

Elements of Store will actually be triples $\langle s_1, s_2, m\rangle$ with $s_1(l)$ giving the CAR of a location l, $s_2(l)$ giving its CDR and m giving the next location to be allocated by CONS. The standard LISP-D semantics models a subset of LISP and is given denotationally in section 5.12.

We adopt the (slightly unorthodox) convention of writing the objects (locations) constructed by (CONS e_1 e_2) as $[v_1 . v_2]$ where the e_i evaluate to the v_i, to avoid confusion between program text and values.

Since the intention is that our external (applicative) world will solely concern itself with values, not locations, our semantics contains such a "print" semantic function which "forgets" locations, abstracting only their values.

This work requires some model of sharing, and we will choose to follow Schwarz and call the elements of these models isolation classes. The result of an expression will be categorised as being in a certain isolation class if certain sharing properties hold (see section 5.7).

5.4.1 Does CONS have a side effect?

This is an interesting question, especially in the context of the recent upsurge of interest in applicative languages, and more so due to the fact that there are two opposing viewpoints. It would seem to be clear that, mathematically at least, CONS merely produces a value which is already present in the universe of discourse and just happens to be a pair of two other values. Similarly it is just as clear from the description of CONS as "producing a new object whose components are its parameters" must have a side effect on something so that we can elaborate the details of "new".

The resolution of this apparent paradox is that CONS necessarily has a side effect in any direct style semantics involving locations, for example the LISP-D semantics given in section 5.12. However the semantics given in section 5.12.3 uses mathematical tupling rather than locations and so has no side effect.

A closely related point to be noted is that any LISP-A program can be considered as a program in LISP-D, however in the former CONS is pure, and in the latter CONS has side effects, whereas (hopefully) they should both give the same results. This paradox is resolved by noting that the effect of the 'Print' semantic

function is to hide the mode of evaluation, by mapping the two different actual results of calculation (one involves locations, the other tuples) onto the same output form on paper. In general this "print the same result" is the notion of equivalence we seek. More discussion of the effect of printing can be found in section 5.9.1.

5.4.2 Destructive Operators

The intention is to introduce 'standard' destructive operators, for example RPLACA or RPLACD instead of CONS (the reader is referred to section 5.12), or NCONC instead of APPEND to implement safely these operators, but with a saving of space, time or garbage collection. However it seems that to do so directly is more complicated than adopting the strategy given below in section 5.4.2.2. First however we will consider a little worked example.

5.4.2.1 Simple Example

Suppose we define a function F, say, by

```
(F Z)  =  (IF (ATOM Z) (ERROR)
                (CONS (G (CAR Z)) (CDR Z)))
```

where G is a previously defined function. Now suppose that for a class of calls to F the actual parameter is of the form

(CONS E_1 E_2)

where E_1 and E_2 are expressions. This parameter necessarily evaluates to give a location l containing, say,

[a . b]

However we know that CONS has the property that l does not occur either as a variable binding, r, or within the store, s, which is passed to CONS. Therefore, for this class of calls, we have that Z will be bound to the only pointer to l.

Clearly then, the evaluation of F also produces a new location m, say, containing

```
[c . b]
```

where c is the result of evaluating (G (CAR Z)). However it is apparent that the node l will have no further pointers to it when F returns and Z is no longer bound to l. Therefore we may use a destructive version of CONS, which is traditionally called RPLACA (see section 5.12), and write

```
(F Z)  =  (IF (ATOM Z) (ERROR)
              (RPLACA Z (G (CAR Z))))
```

We may also view this as an optimisation of the following code, which is simpler is some respects and will be discussed in more detail later:

```
(F Z)  =  (IF (ATOM Z) (ERROR)
              (CONS (G (CAR Z))
                    (PROG1 (CDR Z) (FREE Z))))
```

where FREE is the function which requires a CONS location for a parameter and returns it to the CONS free list.

Note that our new destructive code relies heavily on call-by-value semantics (so that (CAR Z) is evaluated before the RPLACA) and left-to-right evaluation (in the use of PROG1).

The arguments presented here will be formalised by the rest of this chapter.

5.4.2.2 The choice of destructive operators

Here I wish to deviate from what seems to be the standard technique of introducing many destructive operators, and instead merely introduce a single destructive operator just as we introduced a single constructor (CONS). The intention is to factor

the problems of actually inserting destructive operators into two parts, by separating the concept of detecting unused nodes from that of re-allocating them.

Firstly introduce a single destructive operator

(FREE n)

where n is a term we will require to evaluate to a CONS node. The intended interpretation of FREE is to supply the information "This node is available for re-use" to the run-time system. However it will be necessary for us to define its semantics somewhat differently in the standard interpretation in order to ensure that it is only used in situations where this is indeed the case. In fact the standard semantics will merely mark a cell as having been FREE'd and produce an error upon further reference to the contents (as opposed to location) of that cell.

Secondly, we will regard all the other destructive operators as compound forms of FREE. The idea is that we shall regard the conversion of FREE into RPLACA, NCONC etc. as merely one of local allocation, which is (currently at least) conceptually much simpler and more in the province of traditional compiler analysis. For example, given that we are using RPLACD to optimise store re-use rather than for its side effects, we can define

(RPLACD x y) = (DCONS x (CAR x) y)

where

(DCONS x xc y) = (PROG2 (FREE x) (CONS xc y))

Here PROG2 represents sequencing (see section 5.4.3), and returns the value of its second term. In fact, our semantics will provide

exactly the same result[2] for the two definitions provided the first parameter to RPLACD is never referenced again (excepting via the location returned by RPLACD). In an actual implementation, we would of course define FREE to return CONS nodes to the free list rather than the approach taken in section 5.12 of simply marking the node as unusable and causing an error when such a node is referred to. This definition was adopted to provide a means to ensure that our program respects the given (applicative) semantics. This two level scheme corresponds to the factorisation of the correctness proof, which is otherwise much more involved.

It is important to note that since all destructive operators can be built from RPLACA and RPLACD, the above definition allows us to define any destructive operator in terms of FREE.

Another advantage of using FREE as our destructive operator is that we can always insert it to release a CONS node we can show to be unused. This is not the case for RPLACA/D since we must not only find such a CONS node but also an occurrence of CONS to re-use it in.

5.4.3 Order of evaluation

Note that our (applicative) LISP-A is independent of order of evaluation provided we treat the IF form correctly. This is because changing the Seq form which handles the case of multiple parameters to a single function then can only give rise to a

---

[2]modulo changes of locations which are not directly visible to the external world

permutation of addresses in Loc, and the act of printing "forgets" the actual locations used.

However in LISP-D we have to specify an order of evaluation in order to define the semantics, due to the fact that we have introduced a function FREE which has a side-effect on Store. We here will (somewhat arbitrarily) choose left to right evaluation by default. This is the purpose of Seq in the semantics.

Thus we may see that, when using left to right evaluation

E⟦(PROG2 (FREE X) (CAR X))⟧(r,s) = (?,s')

whereas

E⟦(PROG1 (CAR X) (FREE X))⟧(r,s) = (A_1,s")

in a store s and an environment r where X is bound to

[A_1 . A_2] = E⟦(CONS A_1 A_2)⟧.

PROG1 and PROG2 are the natural projections which return (the value of) respectively their first or second parameter. The second term above may also be considered to be

E'⟦(PROG2 (FREE X) (CAR X))⟧(r,s)

evaluated using a right to left strategy, E'.

Note that we here consider PROG1 and PROG2 as pure functions, defined by

```
(PROG1 X Y)  =  X
(PROG2 X Y)  =  Y,
```

and treat the sequencing normally associated with them as part of the semantics of constructing an argument tuple, thus making this sequencing explicit for all functions.

A point worth examining in a little more detail is the possibility of exploiting the flexibility inherent in the

independence of LISP-A semantics with respect to order of evaluation. We could then produce a LISP-D version whose evaluation strategy is effectively to evaluate the parameters to a function in such a manner as to maximise the re-use of store. This is worthy of further research and not considered here.

5.4.4 Occurrences and other Basic Ideas

This section introduces the idea of labelling a particular occurrence (see Donzeau-Gouge [12]) of a term within an expression. We need this to talk about program transformations to insert destructive operators. It is also required to enable us to model denotationally the effect of execution ordering. In order to discuss the effect of sharing we define the idea of active terms in an expression (these are the only ones which can contribute to the result).

We define the notion of occurrence of a term in an expression. Occurrences are tuples, written $\langle p_1 \ldots p_n \rangle$ with $\langle p | q \rangle$ denoting the tuple whose first element is p and the remainder given by the tuple q. We will write @ for infixed append on tuples. We will say q is an initial segment of p if p = q @ r for some r. The occurrence $\langle p_1 \ldots p_n \rangle$ in an expression e is defined recursively in the following manner.

$occ(\langle\rangle, [\![e]\!]) = e$
$occ(\langle p|q \rangle, [\![(G\ e_1 \ldots e_k)]\!]) = occ(q, [\![e_p]\!])$ if $1 \le p \le k$
$occ(p, [\![e]\!])$ is otherwise not defined

This defines the set of valid occurrences and their corresponding terms in an expression. Now define an ordering on Occ, the set of valid occurrences within an expression, by the standard

lexicographic post-ordering on tuples given by

$x \leq \langle\rangle$

$\langle a|x\rangle \leq \langle b|y\rangle$ if $a \leq b$ or ($a = b$ and $x \leq y$)

Note that this also specifies our left to right execution ordering except for the case of the of the IF form, which differs because the 2nd and 3rd sons are exclusively evaluated rather than being sequentially executed. We use the post ordering to account for the fact our semantics uses the LISP style depth-first (call-by-value) evaluation order. The concept of execution ordering is defined to be that modification of the lexicographic ordering to consider the IF form correctly. We define p to be executed before q, written $p \ll q$, if p preceeds q in the above lexicographic ordering and p and q are not (sub-terms of) different consequent branches of any conditional. Formally, $p \ll q$ if $p \leq q$ and for no common initial segment, r, of p and q do we have

$$\begin{aligned} occ(r,U) &= (\text{IF } e_1\ e_2\ e_3) \text{ with} \\ p &= r @ \langle 2\rangle @ l \text{ and} \\ q &= r @ \langle 3\rangle @ m. \end{aligned}$$

This defines $\ll$ to be a partial order. We must take care with this definition of execution ordering - it neither implies that evaluation of p inevitably precedes that of q, nor that evaluation of q inevitably follows that of p, even for terminating evaluations. The set of execution paths through an expression U is the set of maximal chains of occurrences with respect to the execution ordering on occurrences.

For an occurrence p we define the set of active occurrences at p in U by

$$\text{Active}(p,U) = \{p\} \cup \text{Uncles}(p,U)$$

$Uncles(\langle\rangle, U) = \{\}$

$Uncles(q \mathbin{@} \langle i\rangle, U) = Uncles(q,U) \cup Brothers(q,i,U)$

$Brothers(q, i, U) = \{\}$ if f=IF & (i=2 or i=3)

$\qquad\qquad\qquad\qquad = \{q \mathbin{@} \langle j\rangle : 1 \le j \le r \;\&\; j \ne i\}$ otherwise .

where $[\![(f\ e_1\ \ldots\ e_r)]\!] = occ(q,U)$.

The active occurrences wrt p in U are the brothers of initial segments of p, together with p itself. For example, if p labels the occurrence of (CAR X) in

```
(FUN (CDR W)
     (IF (P X) (CONS (CAR X) Y) (FOO Y))
     (ATOM Z))
```

then the active occurrences with repect to p in this term are (labels of) the terms

(CDR W), (CAR X), Y, (ATOM Z).

The reason for the importance of active occurrences is that any CONS node associated with a variable X_i must appear in the result of an active occurrence if it is to form part of the result of U, given that execution passes through p. That this is so is a consequence of the fact that in our first order applicative language CONS nodes returned as a result of functions must either be new CONS nodes created by the function or appear as part of one of its actual parameters. Inductively this means that CONS nodes associated with the X_i can only occur in the result of U if they occur as part of the result of an active occurrence. The special treatment of the conditional is merely an optimisation based on the fact that the result of a conditional can only involve CONS nodes occurring in the current consequent. It is helpful to observe that

q being active at p implies $p \underline{\ll} q$ or $q \underline{\ll} p$.

For transformations we will need the concept of substituting one term for another at a given occurrence within an expression. This will be done by

subst[p,U,⟦e⟧]

which gives a term identical to U except that the (valid) occurrence p in U is replaced with e. We will refer to this as replacing occ(p,U) by e.

5.5 The Extent of Possible Use of Arguments

Following Schwarz's terminology, though not his definitions, we will now define two abstract interpretations called E_{uses} and E_{exam}. These will respectively describe (upper bounds on) the extent to which a term may build in structure from its parameters, and examine parts of its parameters in order to produce a result. Archetypal examples of these two notions are respectively the second and first positional parameters in the three parameter conditional

(IF condition trueresult falseresult).

We introduce a semantic function to describe which CONS nodes, present in structure bound to variables, are built into the the result of the real computation before introducing (superset) approximations to these. Let v be a value and s a store, then the set of CONS nodes present in the structure of v is given by Nodes(v,s) where Nodes(v,s) is the smallest set of locations satisfying

$v \in Loc \Rightarrow v \in Nodes(v,s)$

$n \in Nodes(v,s) \ \& \ s_1(n) \in Loc \Rightarrow s_1(n) \in Nodes(v,s)$

$n \in Nodes(v,s) \ \& \ s_2(n) \in Loc \Rightarrow s_2(n) \in Nodes(v,s)$

where $\langle s_1,s_2,m\rangle = s$. Therefore, we can now define BuiltIn⟦e⟧(r,s) to be the set of CONS nodes present in structure bound to variables which will be incorporated into the result of evaluating e in environment r with store s. It is given by

$$\bigcup_i \text{Nodes}(r[\![X_i]\!],s) \;\cap\; \text{Nodes}(E[\![e]\!](r,s)).$$

Now we wish to have some <u>algebraic</u> formulation of the concept of CAR/CDR selected sub-structure of parameters which may be traversed at run time. We will define the set of <u>paths</u> to be that produced by the grammar

$$\{h,t\}^* \text{ Var } \{h,t\}^*$$

where h (head) and t (tail) are regarded as free symbols, and Var the set of variables. The intent is that the h's and t's before the variable indicate routes to the variable, and those after show how the variable is selected upon with CAR and CDR. Thus for example (see the semantics below) we would have that

```
(CONS (CAR (CDR Y)) (CONS (CAR Y) Z))
```

has paths {h.Y.t.h, t.h.Y.h, t.t.Z}.

Of course, we will only be able to derive an approximation to the set of paths which will actually exist at run time, but as usual in abstract interpretation (see chapter 2) the paths we infer will be a superset of those which can occur at run time.

In the following X will range over Var. The evaluations are

$E_{uses}[\![A]\!] = \{\}$
$E_{uses}[\![X]\!] = \{X\}$
$E_{uses}[\![(CAR\ e)]\!] = HD(E_{uses}[\![e]\!])$
$E_{uses}[\![(CDR\ e)]\!] = TL(E_{uses}[\![e]\!])$
$E_{uses}[\![(CONS\ e_1\ e_2)]\!] = h.E_{uses}[\![e_1]\!] \cup t.E_{uses}[\![e_2]\!]$
$E_{uses}[\![(ATOM\ e)]\!] = \{\}$
$E_{uses}[\![(IF\ e_1\ e_2\ e_3)]\!] = E_{uses}[\![e_2]\!] \cup E_{uses}[\![e_3]\!]$
$E_{uses}[\![(F\ e_1\ \ldots\ e_k)]\!] = compose[F^{uses}, \langle E_{uses}[\![e_i]\!]\rangle]$
$F^{uses} = E_{uses}[\![U]\!]$ where $(F\ X_1\ \ldots\ X_k) = U$

where

$HD(S) = \{x: h.x \in S\} \cup \{X.x.h: X.x \in S\}$
$TL(S) = \{x: t.x \in S\} \cup \{X.x.t: X.x \in S\}$
$h.S = \{h.x: x \in S\}$
$t.S = \{t.x: x \in S\}$
$compose[S, \langle T_i\rangle] = \cup \{comp1[s,\langle T_i\rangle]: s \in S\}$
$comp1[x.X_i.y, \langle t_1\ \ldots\ t_k\rangle] = x.comp2[y, t_i]$
$comp2[h.y, t] = comp2[y, HD(t)]$
$comp2[t.y, t] = comp2[y, TL(t)]$
$comp2[(), t] = t$

These provide recursive definitions for the F_i^{uses} in the domain of sets of paths ordered by inclusion, with least element {}. We must show that the system functions given above abstract the standard semantics in the sense of chapter 2. This is merely a matter of ensuring the consistency of the E_{uses} evaluation with the standard evaluation by showing that every possibility of the real calculation is modelled in the abstract system.

Note that the general theory of abstract interpretation establishes this correctness result for user defined functions, given the corresponding correctness proof for the base functions.

We show the consistency for the IF function above. Suppose that we have a term $[\![(IF\ e_1\ e_2\ e_3)]\!]$, then E_{uses} says that each term

built into the result comes from the evaluation of e_2 or e_3. This is clearly the case as the standard semantics provide no way for any CONS node occurring in the result of e_1 to occur in the result of IF unless it also occurs in the result of e_2 or e_3. However we might note that our approximation, although safe, is not accurate since in $[\![$(IF True e_2 e_3)$]\!]$ CONS nodes occuring in e_3 cannot be incorporated in the result of the IF unless they also occur in e_2. More detailed discussion of the correctness of the E_{uses} interpretation will be given after the introduction of E_{exam}.

The examines interpretation is similarly defined to take account of nodes passed through for the purpose of determining the result of an expression. For example in

$[\![$(IF (ATOM (CAR X)) (CDR Y) Z)$]\!]$

the CONS nodes referred to by X and Y will need their contents to be intact in order that the standard semantics give the correct result for this evaluation. We will later seek ways to return a node to a CONS free-list and this will be viewed as destroying its contents, but otherwise leaving it alone. The semantic equations for the examines interpretation, E_{exam}, are given below. We use the E_{uses} interpretation as the work-horse but require a separate interpretation to ensure that the IF form and call-by-value are treated correctly.

$$E_{exam}[\![A]\!] = \{\}$$
$$E_{exam}[\![X]\!] = \{X\}$$
$$E_{exam}[\![(CAR\ e)]\!] = E_{exam}[\![e]\!] \cup HD(E_{uses}[\![e]\!])$$
$$E_{exam}[\![(CDR\ e)]\!] = E_{exam}[\![e]\!] \cup TL(E_{uses}[\![e]\!])$$
$$E_{exam}[\![(CONS\ e_1\ e_2)]\!] = E_{exam}[\![e_1]\!] \cup E_{exam}[\![e_2]\!]$$
$$E_{exam}[\![(ATOM\ e)]\!] = E_{exam}[\![e]\!]$$
$$E_{exam}[\![(IF\ e_1\ e_2\ e_3)]\!] = E_{exam}[\![e_1]\!] \cup E_{exam}[\![e_2]\!] \cup E_{exam}[\![e_3]\!]$$
$$E_{exam}[\![(F\ e_1\ \ldots\ e_k)]\!] = compose[F^{exam}, \langle E_{uses}[\![e_i]\!]\rangle] \cup \bigcup_{1 \le i \le k} E_{exam}[\![e_i]\!]$$
$$F^{exam} = E_{exam}[\![U]\!] \quad \text{where} \quad (F\ X_1\ \ldots\ X_k) = U$$

These equations again give a fixpoint equation for the meanings, F_i^{exam}, of the F_i, in the domain of sets of paths with {} as the bottom element.

It will be useful to separate out the concept of arguments (ie paths of the form X.x) being used or examined from the general schemes given above which described how arguments and new CONS nodes constructed in defining equations are used or examined. Thus we will define

$$USED\text{-}L[e,X] = \{y.z:\ x.X.y \in E_{uses}[\![e]\!],\ z \in \{h,t\}^*\}$$
$$USED\text{-}R[e,X] = \{y:\ x.X.y.z \in E_{exam}[\![e]\!],\ \text{for some } z \ne ()\}.$$

In the case of USED-L we include all CONS nodes which can be reached from a path given by the E_{uses} interpretation. We do this on the grounds that although a function may only return a single structure, the calling environment may then extract any sub-structure. By similar reasoning we wish to include in USED-R the selector path leading to our desired node, however we do not wish to include the node itself (this is the purpose of $z \ne ()$), since the contents of such a node are not extracted unless it is described elsewhere in USED-R. For example in

$$e = [\![(CONS\ (ATOM\ (CAR\ X))\ (CDR\ Y))]\!]$$

we have that

$$USED\text{-}L[e,X] = \{\}$$
$$USED\text{-}R[e,X] = \{()\}$$
$$USED\text{-}L[e,Y] = t.\{h,t\}^{*}$$
$$USED\text{-}R[e,Y] = \{()\}$$

We will further say the path X.y.z is USED-L in a term, e, if there is a path x.X.y in $E_{uses}[\![e]\!]$. Similarly we will say X.y is USED-R in e if there is a path x.X.y.z ($z \neq ()$) in $E_{exam}[\![e]\!]$. These terms are taken from the classical concepts of L-mode referring to a location and R-mode referring to its contents.

5.5.1 Correctness of the Uses and Examines Interpretations

The statement of correctness of the uses interpretation is that no CONS node accessible from an environment through the store can possibly occur in the result of an expression (evaluated in this environment and store) unless it is represented in $E_{uses}[\![e]\!]$. In other words, recalling that Nodes(v,s) is the set of CONS nodes accessible from a value v, then what we want is that for all r, s, and e;

$$BuiltIn[\![e]\!](r,s) = \bigcup_i Nodes(r[\![X_i]\!],s) \cap Nodes(E[\![e]\!](r,s))$$

is contained in

$$\bigcup_i Rep(r[\![X_i]\!], s, USED\text{-}L([\![e]\!],[\![X_i]\!]))$$

where Rep(v,s,P) defined similarly to Nodes(v,s) but restricting our attention only to nodes obtained by following CAR/CDR chains given by the paths in P. It is formally defined by

$$Rep(v,s,P) = \bigcup \{Rep2(v,s,y): y \in P\}$$

$$
\begin{aligned}
Rep2(v,s,y) &= Rep3(v,s,y) \quad \text{if} \quad v \in Loc \\
&= \{\} \quad \text{otherwise} \\
Rep3(v,s,y) &= Rep2(s_1(v),s,x) \quad \text{if} \quad y=h.x \\
&= Rep2(s_2(v),s,x) \quad \text{if} \quad y=t.x \\
&= \{v\} \quad \text{otherwise} \\
&\quad \text{where } s = \langle s_1,s_2,l\rangle.
\end{aligned}
$$

Note that $Nodes(v,s) = Rep(v,s,\{h,t\}^*)$.

The statement of correctness of the examines interpretation is that if we FREE all CONS nodes specified by variables in a term e, which do not correspond to any member of the $E_{exam}[\![e]\!]$ then we make no difference to the evaluation of e. More formally

$$E[\![e]\!](r,s) \;=\; E[\![e]\!](r,s')$$

where s' is a version of $s = \langle s_1,s_2,m\rangle$ modified by replacing $s_1(l)$ and $s_2(l)$ by '?', the error value, for all locations l in

$$\bigcup_i Nodes(r[\![X_i]\!],s) \;-\; \bigcup_i Rep(r[\![X_i]\!],\ s,\ USED\text{-}R([\![e]\!],[\![X_i]\!])).$$

5.6 Usage counts

Usage counts are a method of associating integers with every CONS node to count the number of pointers to that node. The intention is that upon creation of a new pointer to a CONS node, we increment the usage count associated with that node. Similarly upon destroying a pointer to a node, we decrement the usage count, the node being returned to free storage upon the counter being decremented to zero.

In general reference counts provide a sufficient condition for returning a node to free space, however the existence of circular or re-entrant structures created by destructive operators could

mean that a set of nodes can no longer be referred to in spite of their reference counts all being non-zero. In (current) applicative languages we cannot create circularities in this manner, and a usage count provides a necessary and sufficient condition for the re-use of store.

In our semantic model (section 5.12) it is possible to define usage counts for CONS nodes in the following manner

$$U: \text{Loc} \times \text{Store} \times \text{VarEnv}^* \rightarrow \text{Number}$$

$$U(l,s,r^*) = U_1(l,s) + U_2(l,s) + U_V(l,r^*)$$

where

$$U_1(l,\langle s_1,s_2,m\rangle) = \text{Card}\{x \in L: s_1(x) = l\}$$

$$U_2(l,\langle s_1,s_2,m\rangle) = \text{Card}\{x \in L: s_2(x) = l\}$$

$$U_V(l,r^*) = \sum_{r \in r^*} \text{Card}\{X \in \text{Var}: r[\![X]\!] = l\}$$

$$L = \bigcup \{\text{Nodes}(r[\![X]\!],s): r \in r^*, X \in \text{Var}\}$$

The only problem in this formulation is that our semantics as given in section 5.12 only allows us to access the current environment and provides no method by which we can access the set of environments which are currently dynamically active (r^* in the above). The simple solution to this problem is to change the environment syntactic category (Env) to include a component which contains environments which are active but inaccessible from the current function. This is not done here to permit the semantics to be as simple as possible.

An alternative to the above method is to include a usage count component in the store, as shown in section 5.12.2.1.

We are careful to break down the usage count into its components

given by other CONS nodes (U_1,U_2), and those given by variables U_V. We emphasise this because variables represent a disciplined form of sharing (being of limited scope), and also because as noted in section 5.7.4 all sharing in structures originates from multiple uses of variables. This aspect of usage counts makes our model a little more difficult to handle than Schwarz's rewrite model [45] which only uses variables to indicate substitutions, and hence requires no variable binding component in the usage count formulae. However, I believe that the extra benefits of denotational style greatly outweigh the disadvantages.

5.7 Isolation classes: abstract interpretations modelling usage counts

In order to insert destructive operators, we must have some notion of how shared an object might be. Isolation classes provide this notion.

We will first introduce a simple isolation class _one_, taken from Schwarz. The meaning of saying an expression, e say, is in (isolation) class _one_ is that the result of e should either be a non-CONS object, or should have the property that the usage count of the CONS node given by e is 1 (excepting irrelevant paths - see section 5.7.6).

For inductive style reasoning we will need rather more expressive concepts than merely being able to say that a result of a term is isolated at the top level. For example it is desirable to express the idea that a whole structure has no external pointers. To this end we introduce a set of isolation classes.

For us the set of isolation classes are the set of subsets of $\{h,t\}^*$, where $\{h,t\}^*$ gives the set of finite strings of h's and t's. We will call the elements of isolation classes paths. To tie this in with the idea given above, we will say that a value, v, in store, s, is in isolation class I, if all members of Rep(v,s,I) are of isolation class one. This is just another way of saying that for all paths x in I, when v.x exists it is in isolation class one. Note that we can recover the isolation class one given above as the isolation class {()} where () denotes the empty string. Henceforth we will use one for either. We will give names to some other isolation classes

arb = {}
ti = $\{h,t\}^*$
onelist = $\{t\}^*$
onehlst = $\{h\}^*$

These will enable us to discuss (arb) objects with no restrictions, (ti) objects totally unshared from other objects and (onelist and onehlst) objects representing lists in CAR or CDR directions with unshared tails.

The isolation classes that we use will be consistent in the following sense: if x.y is a member of I then x is a member of I. We will find no use for CONS nodes that have a single pointer to them, but that pointer comes from a CONS node with a high usage count.

It is clear that the run-time behaviour described by the above isolation classes cannot be exactly modelled at compile-time due to problems of computability (we can simulate numbers as linear lists

within the model). Therefore we will be interested in deriving methods which give sufficiency conditions to show that particular nodes in the run-time state will satisfy these requirements. The works of Cousot & Cousot [8, 9] give us a very general model for this which is discussed in detail in chapter 2. One possible candidate for consideration is to use the sets of objects which can be described by regular trees [49] for the actual isolation class models we can handle at compile time. Regular trees are a generalisation of regular expressions, and have many similar computable properties.

We will use $\sqsubseteq$ as an ordering on isolation classes. It is defined by $I \sqsubseteq J$ if and only if $I \supset J$. Our least element will be *ti*.

5.7.1 The Isolation Ordering

The problem of "which way up" to arrange our abstract value domain of isolation classes is rather a tricky one, especially so due to the fact that we have simultaneously two different concepts of ordering which often indicate their presence by suggesting that the whole lattice should be upside down. For example we can choose one of the two following (*dual*) configurations:

It is very tempting (and fatal) to follow Schwarz and opt for the second structure which orders isolation classes by set

inclusion. The reason why this is wrong is concerned with deriving fixpoint expressions for our sharing structure in that we will want a recursive function definition to start off from the premise that its result is isolated, and change this if it is contradicted by the definition. For example

```
(F X)  =  (IF (P X) (F (Q X))
                    (CONS X NIL))
```

never returns a shared CONS node. The first model domain given above has the property that we can use <u>ti</u> to start our fixpoint iteration and thus derive a LEAST fixpoint. However if we use the second model, we will find that the least fixpoint of such a structure would produce <u>arb</u> for the isolation property of F, due to the fact that <u>arb</u> would be the initial value for the fixpoint iteration and that the IF function must be pessimistic about sharing and satisfy if(x,<u>arb</u>,y) = <u>arb</u> (see section 5.7.2). The alternative solution of using a MAXIMAL fixpoint approach is valid but suffers from the great disadvantage of being much harder to prove correct, since our standard semantics uses minimal fixpoints.

A connected matter is the inclusion of the non-CONS value sets Atom and {?} in all isolation classes (that this is so is a consequence of Rep(v,s,I) being empty if v not a location). The reasoning is similar to that given in the above in that we might wish to argue that whenever a function returns a CONS result then this node is isolated. Certainly we do not wish to consider a function as possibly returning a shared node merely because we cannot show it can never result in a run time error (? in the semantics), similarly list processing functions often have to

return NIL for some inputs, and we will want such functions to return isolated results. It is much simpler, both for the development and proof, to consider non-CONS items as being of class <u>one</u>. What the class <u>one</u> really means, then, is that the object described cannot be a shared CONS node, and we gain efficiency (more results for little analysis) by lumping together (in <u>one</u>) all objects that are not shared CONS nodes. Furthermore this formulation shows that the above definition of <u>one</u> is natural, rather than being ad hoc as at first appearance.

5.7.2 Isolation properties of functions

In order to use the isolation classes just introduced, we must define how they behave in expressions. To this end we will seek non-standard interpretations F^{isol} describing the isolation class of the result of a function in terms of the classes of its parameters. We define the isolation class interpretation for a given base function, B, say, following the Cousots' formulation detailed in chapter 2. Let Isol be our set of isolation classes and

Abs: Val $\rightarrow$ Isol

give the isolation class of a value. Then we define the isolation semantics B^{isol} by

$$B^{isol}(x_1 \dots x_k) = \bigsqcup\{Abs[b(y_1 \dots y_k)]: Abs(y_i) = x_i\}$$

where b is the standard semantics of B. That is, B^{isol}, given by $B^{isol}(x_1 \dots x_k)$, is the least upper bound (in the isolation class ordering) of the isolation classes whose values represented include all the elements which can be constructed by $b(y_1 \dots y_k)$ as the tuple (y_i) ranges over all values represented by the isolation

class tuple (x_i).

Henceforth we will not refer to the standard semantics again and accordingly use b instead of B^{isol} (and similarly for other functions). Thus the above defines

$$cons(I,J) = \{()\} \cup h.I \cup t.J$$
$$car(I) = \{x: h.x \in I\}$$
$$cdr(I) = \{x: t.x \in I\}$$
$$if(I,J,K) = J \cap K$$
$$atom(I) = \underline{ti}.$$

For example

$$cons(\underline{arb}, \underline{onelist}) = \underline{onelist}.$$

We will now formalise this introduction into an interpretation by defining a new abstract interpretation, E_{isol}. Firstly, however, we will include an isolation environment which associates variables (parameters) to isolation classes, defined by

$$IsolEnv = Var \rightarrow Isol.$$

This enables us to define functions F^{isol} for user functions. Unfortunately the semantics given below is only correct for terms in which variables only occur at most once. This will be corrected in section 5.7.5 after we find the difficulties in the 'obvious' interpretation. The functionality of E_{isol} is now

$$E_{isol}: Exp \rightarrow IsolEnv \rightarrow Isol$$

with definitions

$$E_{isol}[\![A_i]\!]R = \underline{ti}$$
$$E_{isol}[\![X_i]\!]R = R[\![X_i]\!]$$
$$E_{isol}[\![(\mathrm{CAR}\ e)]\!]R = car(E_{isol}[\![e]\!]R)$$
$$E_{isol}[\![(\mathrm{CDR}\ e)]\!]R = cdr(E_{isol}[\![e]\!]R)$$
$$E_{isol}[\![(\mathrm{CONS}\ e_1\ e_2)]\!]R = cons(E_{isol}[\![e_1]\!]R,\ E_{isol}[\![e_2]\!]R)$$
$$E_{isol}[\![(\mathrm{ATOM}\ e)]\!]R = \underline{ti}$$
$$E_{isol}[\![(\mathrm{IF}\ e_1\ e_2\ e_3)]\!]R = E_{isol}[\![e_2]\!]R \sqcup E_{isol}[\![e_3]\!]R$$
$$E_{isol}[\![(F\ e^*)]\!]R = F^{isol}(E_{isol}[\![e_1]\!]R \ldots E_{isol}[\![e_k]\!]R)$$
$$F^{isol} = \lambda x^*.E_{isol}[\![U]\!](\lambda X^*.x^*)$$
$$\text{where } (F\ X^*) = U.$$

This provides a fixpoint definition of the isolation class of a function in terms of the isolation classes of its parameters and its textual definition. We use the textual definition for user functions rather than the approach taken for system functions, since computability restrictions imply the inability to calculate the standard denotation of a user function at compile-time.

However, as indicated, the above semantics is wrong because variables might occur twice and produce the problems given in the next section.

5.7.3 The problems of variables

Consider the two program fragments:

$$[\![(F\ X) = (G\ X\ X);\ (F\ (\mathrm{CONS}\ A_1\ A_2))]\!] \qquad (1)$$

$$[\![(G\ (\mathrm{CONS}\ A_1\ A_2)\ (\mathrm{CONS}\ A_1\ A_2))]\!] \qquad (2)$$

with G defined elsewhere.

In (2) G may freely destroy the CONS node which constitutes the top-level of either (or both) of its parameters. However, in (1) G may only destroy the CONS node corresponding to one of its parameters, and even then only if (and when) it has finished using

the other one. Therefore we find that our abstract interpretation cannot be referentially transparent due to the fact that a location-CONS based semantics cannot of itself be referentially transparent (although the external world view can be - provided that we ensure that the exact locations used in Loc cannot be distinguished by the program or the printing routines).

5.7.4 The treatment of variables

The problem of variables not fitting into the standard framework of abstract interpretation is due to the non-referential transparency of a location-CONS based semantics. However as we will see, it is possible to use the abstract interpretation idea by treating variables rather cautiously and using an environment which takes account of multiple uses of a single variable. This will be discussed in section 5.7.5. It is worth spending some time considering variables for the reason that variables are the cause of all sharing in a program[3] in the sense that, if any two

[3] To ensure that variables are the only possible cause of sharing it is necessary to put some (mild) restriction on the objects we are willing to accept as system functions. These restrictions are not central to the work on introducing destructive operators, but are given merely for the correctness of the view that variables cause all sharing. A small example will show that sharing may arise without using variables if some restriction is not placed on system functions. Suppose we had a system function CONSXX whose semantics were the same as those given by the user function

    (CONSXX Y) = (CONS Y Y)

We would then be able to produce shared substructure without using variables (system functions are defined by their effect rather than by their textual definition). Thus, to make the above claim watertight, it is necessary to include an assumption to the effect that all system functions have the property that every CONS node existing before the call to that function does not have its usage count increased by the call.

substructures share then at one time they must have been associated with the same variable. That this is the case can be seen by observing that our language only permits a function to return a (possibly structured) single result. Hence the only way to produce two references to the same structure (not copies), is to associate the result of a function with a variable, and then use the variable two or more times.

Furthermore, variables represent a much more disciplined form of sharing than does arbitrary sharing of CONS nodes by different substructures. Variables are of limited scope and so it is always possible to identify points at which they release their grip on list structure. Moreover the information on uses of a variable is explicitly available in the textual form of the program, whereas that for CONS nodes has to be deduced from a rather more detailed analysis of program structure.

5.7.5 The details of variable sharing

Returning to the problem raised in section 5.7.3 in which X is effectively a *where* variable, we will develop the following solution, which makes use of the fact that the isolation properties of a variable depend on the other occurrences of that variable.

We will explain our method with reference to a simple example:

```
(G Y Z)  =  U
(F X)   =  (G (CAR X) X)
```

where U is a term not further specified. We will further suppose that for some class of calls to F we have that the parameter X will be of class *ti*. That is, the isolation environment, R, passed to F

will have $R[\![X]\!] = \underline{ti}$. Now we will consider the implied isolation properties of Y and Z for the calls of G from the above calls to F. Firstly, the isolation interpretation for CAR has $car(\underline{ti}) = \underline{ti}$ however, we see that X.h (which is the path for (CAR X)) is also USED-L by X (the second parameter to G), and thus the sharing properties of the two occurrences of X should really modify the isolation class for (CAR X) to

$$E_{isol}[\![(CAR\ X)]\!]\ R \ = \ \underline{arb}.$$

(We cannot do any better than this because every node of the structure Y accessible by G in U can also be accessed by Z).

Similarly when we look at the second argument, Z, of G and its corresponding actual parameter, X, we see that it is indeed shared (with Y), but only paths of the form X.h.y have this property. Indeed we still have that the paths X.h.z, and X.t.y are unshared for any y, z in $\{h,t\}^*$ (class ti precludes sharing within X). Hence we could say that

$$E_{isol}[\![X]\!]\ R \ = \ \underline{ti} - h.\underline{ti} \ = \ cons(\underline{arb},\underline{ti}).$$

If our choice of isolation classes was {arb, one, onelist, onehlst, ti} the nearest to this value we can represent is onelist.

As the above example indicates the abstract evaluation function E_{isol} for variables will have to take into account the number of times and in which contexts a variable occurs. Specifically we want something like

$$E_{isol}[\![X_i]\!]\ R \ = \ R[\![X_i]\!] - Shared(p,U)$$

where p labels the particular occurrence of X_i in the term U which is the (whole) right hand side of the definition in which p occurs.

Shared(p,U) will indicate which portions of X might be shared. The first apparent objection to this definition is that it is not denotational in that the meaning of a term containing variables is not just dependent on its sub-terms but also on its enclosing expression. We will counter this objection by changing E_{isol} to be of higher order, and passing the enclosing expression around between all recursive uses, thus making it available when required. Similarly the occurrence p used above can be carried around explicitly as a parameter to E_{isol} - no magic is involved. This technique of making functions higher order to have terms available when required is standard in denotational semantics and for further information the reader is directed to Mike Gordon's book on semantics [16], and in particular to chapter 11 where he examines the subject of Algol-60 OWN variables which require 'position dependent denotations' and receive very similar semantic treatment.

Thus we are led to introduce a semantic function 'Deisolate' which modifies isolation environments according to the possible sharing induced by multiple uses of a particular variable. It is this trick which enables us to model the non-referential transparency of our language in a direct manner. I would claim that the invention of the following denotational formulation is one of the major developments of this chapter over the work of Schwarz.

$$E_{isol}[\![A_i]\!]R\ p\ U \ = \ \underline{ti}$$
$$E_{isol}[\![X_i]\!]R\ p\ U \ = \ R[\![X_i]\!]$$
$$E_{isol}[\![(CAR\ e)]\!]R\ p\ U \ = \ car(E_{isol}[\![e]\!](Deisolate\ R\ p\ 1\ U)\ p@\langle 1\rangle\ U)$$
$$E_{isol}[\![(CDR\ e)]\!]R\ p\ U \ = \ cdr(E_{isol}[\![e]\!](Deisolate\ R\ p\ 1\ U)\ p@\langle 1\rangle\ U)$$
$$E_{isol}[\![(CONS\ e_1\ e_2)]\!]R\ p\ U \ = \ cons(E_{isol}[\![e_1]\!](Deisolate\ R\ p\ 1\ U)\ p@\langle 1\rangle\ U,\ E_{isol}[\![e_2]\!](Deisolate\ R\ p\ 2\ U)\ p@\langle 2\rangle\ U)$$
$$E_{isol}[\![(ATOM\ e)]\!]R\ p\ U \ = \ \underline{ti}$$
$$E_{isol}[\![(IF\ e_1\ e_2\ e_3)]\!]R\ p\ U \ = \ E_{isol}[\![e_2]\!](Deisolate\ R\ p\ 2\ U)\ p@\langle 2\rangle\ U \sqcup E_{isol}[\![e_3]\!](Deisolate\ R\ p\ 3\ U)\ p@\langle 3\rangle\ U$$
$$E_{isol}[\![(F\ e^*)]\!]R\ p\ U \ = \ F^{isol}(E_{isol}[\![e_1]\!](Deisolate\ R\ p\ 1\ U)\ p@\langle 1\rangle\ U\ \ldots\ E_{isol}[\![e_k]\!](Deisolate\ R\ p\ k\ U)\ p@\langle k\rangle\ U)$$
$$F^{isol} \ = \ \lambda x^*.E_{isol}[\![U]\!](\lambda X^*.x^*)\ \langle\rangle\ [\![U]\!] \quad \text{where } (F\ X^*) = U.$$

where

$$Deisolate\ R\ p\ i\ U \ = \ R[r_1\ \ldots\ r_k/X_1\ \ldots\ X_k]$$
$$\text{where } r_j \ = \ Irrelevant(R[\![X_j]\!],p@\langle i\rangle,U,X_j)$$

and

$$Irrelevant(I,p,U,X) \ = \ I - \bigcup\{USED\text{-}L[occ(q,U),X]:\ q \in Active(p,U)-\{p\}\} - \bigcup\{USED\text{-}R[occ(q,U),X]:\ q \in Active(p,U)-\{p\},\ q \underline{\gg} p\}.$$

We separate the notion of Irrelevant so that it can be used later. The name Irrelevant is used because we only consider X as being isolated along paths which are irrelevant to the side computations of occurrence p. This definition of E_{isol} is essentially the same as before and the only serious change (apart from the ubiquitous introduction of U and p) is to change the isolation environment to account for other uses of a variable. Given a variable, X say, we reduce its isolation properties for sub-expressions of a given term

by treating as shared all paths of X which can possibly affect the computation (USED-R) or be built into the result (USED-L). We will now justify the choice of occurrences, q, ranged over in the above definition of R'. Now, as noted in the definition of active occurrences, given we are evaluating a term at occurrence p within expression U, the only way for a CONS node which forms part of one of the variables to appear in the result of U, is by it appearing in the result of an active occurrence. We exclude p itself since the recursive use of E_{isol} will deal with it. Therefore we infer that we should treat as shared any node that might occur in the result of any active occurrence except p itself. However it is worth noting that the union of USED-L terms can be restricted to Brothers(p,i,U), if we desire, since brothers of initial segments of p will have already been considered by the uses of Deisolate on the terms corresponding to those initial segments in the recursive application of E_{isol}.

Similarly, we now consider in which circumstances the examining of the contents of a CONS node can require us to treat the node as shared. The point to note is that passing a parameter to a function with the information that it lies within a certain isolation class is a invitation for the corresponding structure to be destroyed. Hence we must ensure that the corresponding CONS nodes are not USED-R after the call of the (possibly destructive) function. We have set up our E_{exam} interpretation in such a manner that if a path X.y is USED-R in a subterm of a given term then it is USED-R in the given term. Therefore the required condition is

that paths should be treated as shared if they are USED-R at any active occurrence q wrt p which can follow the execution of p. Due to our definition of execution ordering this condition can be simply expressed as $q \underline{\gg} p$. Again it would be equivalent for the formula for Deisolate to merely consider the Brothers of p rather than all active occurrences since the recursive formulation of E_{isol} ensures that an earlier Deisolate will have considered them.

5.7.6 Irrelevant paths

One point which should be made now, is that, contrary to what we suggested in the naive introduction to FREE, FREE can never operate on a CONS node of usage count zero since the structure being FREE'd has to be referenced to be passed as a parameter. For example, consider evaluating

```
(F (CONS e1 e2))
```

in

```
(F X)  =  (G X)
(G Y)  =  U
```

for some term U. We would like to argue that Y is of isolation class one within U in this invocation of G via F. However during the evaluation of U the usage count of the node referred to by Y is at least 2 (X and Y each refer to it). So what we see ourselves as doing, then, is to propagate backward the (notional) unbinding of variables to the point at which they can last affect the computation. This situation not only affects variable bindings, but can also be exhibited with CONS structure. Consider

```
(F X)  =  (H1 (H2 (CAR X)) (CDR X))
```

and suppose that the parameter X to F can be shown to be ti. We wish to argue that the parameter to H2 is isolated but we cannot

argue that X is used for the last time at the call to H2, because it is also required for the later occurrence of (CAR X) in H1. What we can say is that the paths X.h.y become irrelevant after calling H2 and so H2 can destroy objects on those paths. Similarly all paths X.y become irrelevant upon completing the call to H1. Since we do not want to destroy structure shared by the function calling F we had better insist that structure can only be irrelevant if it corresponds to a path within the isolation class for X.

This notion of irrelevance has been described previously. We will say that a path X.y for a variable X of isolation class I is irrelevant at an occurrence p within a term U if y is a member of

Irrelevant(I,p,U,X).

Note well that this backwards propagation of unbinding information is dependent on the order of evaluation (section 5.4.3), and provides another reason for the strict semantic formalism.

5.8 Useful Transformations

This section introduces the transformations which we will use in our simple examples. However, it is claimed that they are also useful for larger programs, and accurately capture typical uses of destructive operators when inserted by hand.

5.8.1 Transformations to insert FREE

We will now define the transformations of the code of a definition of F, say, given by

$(F\ X_1 \ldots X_k) = U.$

In the following we will assume that X_1 has been shown to be always

bound to a value in a particular non-arb isolation class, (for example one), for each of the calls of F under consideration. Similar transformations apply to the other X_i.

Our first transformation replaces occurrence p given by

$$occ(p,U) \;=\; (G\ e_1\ \ldots\ e_k)$$

with the term

$$(G\ e_1\ \ldots\ e_{r-1}\ (PROG1\ e_r\ (FREE\ X_1))\ e_{r+1}\ \ldots\ e_k)$$

which FREE's X_1 after e_r, for any non-IF function G, under the conditions that

- X_1 is not USED-L at any active occurrence (wrt p@<r>).
- X_1 is not USED-R at any active occurrence q (wrt p@<r>) such that q $\underline{\gg}$ p@<r>, q $\neq$ p@<r>.

The first condition is there to ensure that (the possible location referred to by) X_1 cannot occur in the result of U. The fact that X_1 is not arb ensures that it is not shared (at top level) with any other variable which does occur in the result. The second ensures that the contents of X_1 are not corrupted by the FREE in the case that, say, (CAR X_1) is tested later in the execution of U. Note that we do not need to worry about X_1 being USED-R before or at p@<r> since such references to the contents of X_1 have already made their effect on the computation. Note that these conditions are equivalent to the path X.() for X being Irrelevant at p@<r> in U, with the added restriction that X.() being not USED-L in occ(p@<r>,U).

5.8.2 Transformations for IF

The above transformation is valid if G = IF, but we can find a stronger transformation which takes advantage of the special

properties of the conditional. Suppose that there is a occurrence p in U given by

$$occ(p,U) = (IF\ e_1\ e_2\ e_3)$$

then we can replace occ(p,U) in U by the term

$$(IF\ e_1\ (PROG2\ (FREE\ X_1)\ e_2)\ \ e_3)$$

which FREE's X_1 before e_2, under the following conditions:

- X_1 is not USED-L at any active occurrence (wrt p@<2>).

- X_1 is not USED-R at any active occurrence q (wrt p@<2>) such that q $\underline{\gg}$ p@<2>.

Note that these are essentially the same conditions that allow us to replace

$$[\![(G\ e_1\ e_2)]\!]$$

by

$$[\![(G\ (PROG1\ e_1\ (FREE\ X_1))\ e_2)]\!].$$

This represents the fact that the conditional can only return results via the selected consequent. We have to move the (FREE X_1) to immediately before e_2 rather than immediately after e_1 in order that X_1 is not affected in the case that the e_3 branch is taken. Moreover, in normal left to right sequencing we have that

$$(G\ e_1\ \ldots\ (PROG1\ e_r\ (FREE\ X_1))\ e_{r+1}\ \ldots\ e_k)$$

is equivalent to

$$(G\ e_1\ \ldots\ e_r\ (PROG2\ (FREE\ X_1)\ e_{r+1})\ \ldots\ e_k)$$

from the semantic definitions, and also intuitively from the observation that both terms FREE X_1 between e_r and e_{r+1}.

Similarly we can replace occ(p,U) in U by the term

$$(IF\ e_1\ e_2\ (PROG2\ (FREE\ X_1)\ e_3))$$

mutatis mutandis.

5.8.3 Replacing FREE with RPLACA/D

One transformation, which will be used in the examples in spite of the fact that this chapter does not address it in detail is the following simple case of store re-use. Suppose we transform U into V, say, then we can optimise FREE/CONS pairs in V as indicated in the following. Let p and q be respectively occurrences of (FREE X_1) and (CONS e_1 e_2) with the property that p$\underline{\leq\leq}$q and that p appears in every execution path in which q appears. Our transformation is given by removing the FREE and changing occ(V,q) into

(DCONS X_1 e_1 e_2)

where

(DCONS X Y Z) = (PROG2 (FREE X) (CONS Y Z)).

In a 'real' FREE implementation (section 5.9.1) we would use

(DCONS X Y Z) = (RPLACD (RPLACA X Y) Z).

Removing the FREE is simpler than at first sight because it always occurs within a PROG1 or PROG2 in the form

(PROG1 e (FREE X_1)) or (PROG2 (FREE X_1) e)

and such a removal merely consists of replacing the occurrence of PROG1 or PROG2 by e. The main reason why we do not consider in detail such transformations is that it is not clear, in general, how to optimally associate FREE/CONS pairs.

5.9 Correctness with respect to the semantics

I have not worked out the full details of the following sketch of how a proof of correctness of the transformations described here would go, however it is hoped that the following gives some intuitive insight into their correctness.

The first step in the proof is to consider a program, P say, and transform it into a program Q by the above techniques. We now consider the equivalence of P and Q. Because the definition of FREE which we give in our semantics (see section 5.12) merely marks the FREE'd CONS node and gives an error on further reference, we thus have that the result of running P is identical to that of running Q (even down to which locations are used) excepting the possibility that Q gives a run-time error whilst P does not. (This is due to the fact that all functions are strict with respect to '?', the value of a run-time error.) Therefore the only thing to be proved is that Q cannot produce an error when P does not. We discuss the effect of using a 'real' FREE which returns items to a free-list in the next section.

Now we will argue that the possible run-time error referred to above cannot actually occur if we use our transformations when they are valid. To do this we will use the definition of a CONS node being irrelevant (see section 5.7.6). We recall that a node is irrelevant at occurrence p if it is of isolation class one and it cannot further affect computation in the current function, nor can it be returned as a result of the current function except via p. However, and this is the cornerstone of the proof, the fact that the node has been passed to the current function as a member of isolation class one means that it is irrelevant at the occurrence of the call in the calling function (otherwise Deisolate would have reduced it from class one to arb). Now we have two cases to consider. Firstly the node may be USED-L in the current function.

In this case the rules for inserting destructive operators will forbid the use of FREE on this node. Therefore a run-time error cannot arise from this cause. Secondly if the mode is not USED-L in the current function it may be destroyed, however it can never occur in the result in this function, nor therefore in the result of any calling function. To complete the argument we observe that

the release of the node in the current function implies that it is not USED-R there after the FREE, and hence an error cannot occur there. Since we therefore can never take the contents of this node we cannot produce the error given by accessing a FREE'd CONS node. Therefore CONS nodes FREE'd by our transformations really cannot affect the computation.

5.9.1 Proof of correctness of 'real' FREE

Having a general setup which enables us to prove the correctness of the transformations given above for the FREE function (given in the semantics) which only marks its argument as being unusable is one thing, but for practical use it is now necessary to show the correctness when using a _real_ FREE function. We will say a FREE function is _real_ if it returns its (location) argument to the free list in order that it might be re-allocated for future CONS'ing.

Here we will discuss in detail the _formulation_ of the proofs of equivalence, and also consider the question of how much remains to be proved for a particular choice of isolation classes.

It is important to see that the reason that this work is ever valid (except in a totally vacuous sense of never being applicable) is in the nature of the PRINT function. The PRINT function has the

property of losing much of the information of machine representation (including which locations are used and which substructures share). We can see its behaviour as a many-one homomorphism from DAGs labelled with addresses, to trees, the standard method of printing a structure. Trees (or rather their flattened forms) are the only permitted method of printing objects in a applicative language whose denotations do not include locations (even though clearly their implementations might use locations). In a sense the PRINT program merely forgets locations. Of course this is why debugging a system using sharing often requires a DUMP containing rather more information than that present in a PRINT.

This fact, together with the requirement for the ability to prove equivalence, explains why the LISP-D semantics has been given in such detail in section 5.12. For example, if we had omitted to specify the semantics of the print function then this work would have been open to the objection of being made inapplicable by choosing a suitable (location preserving) print function.

Let us consider the step by step evaluation of a program resulting from our transformation, using for one a real FREE and for the other our marking FREE, clearly the exact locations involved in the computations will differ, but under the assumption that both programs are implementations of an original applicative program, there is no possibility of the execution sequence depending on this (since we do not introduce any functions for testing equality of locations). Hence the two programs will follow

corresponding sequences of interpreter states with the only possibility of divergence occuring at a reference to a location which has been FREE'd. Upon referencing the contents of such a location the marking FREE interpreter will give a run time error, whereas the real FREE interpreter will extract whatever contents (in D) are present in the cell. However upon assuming the absence of such a run time error in the marking FREE version, as implied by the correctness of inserting marking FREE's, then we are led to the conclusion that the execution paths can never diverge, and hence that real FREE is equivalent to marking FREE for all programs resulting from transformations of an originally applicative version.

5.10 Producing destructive versions of user functions

This section details the considerations necessary to decide which functions we will build destructive versions of, and also applies to non-primitive system functions (eg APPEND).

Consider again the example given earlier

```
(F (CONS X Y))  =  (CONS A Y)
```

in which we argued that F could destroy its parameter and hence be modelled by

```
(F Z)  =  (RPLACA Z A)
```

in the event that we could show that (the location denoted by) the actual parameter to F could never be further referenced in the computation.

However, the following complication may arise where we have two calls

$C_1 = [\![(F\ u_1)]\!]$
$C_2 = [\![(F\ u_2)]\!]$

where u_1 and u_2 are terms such that u_1 always produces an isolated result and u_2 may sometimes produce a result which is shared.

The problem is that we would like to use a destructive version of F for C_1, but that it is unsafe (incorrect) to do so for C_2. This gives us a choice of pursuing either of the following strategies.

Firstly we may adopt the policy of using the destructive version of F for C_1 and the non-destructive version of F for C_2. This produces more code, especially in the case of a large function which is only used twice and performs very little CONS'ing.

Secondly we may choose not to produce a destructive version of F (because it cannot be used everywhere), but instead use our knowledge of sharing to FREE the parameter to C_1 immediately <u>after</u> the call to F. (Of course we cannot FREE the parameter in C_2.) This is most easily done by changing the call C_1 into $[\![(\text{F-DEST}\ u_1)]\!]$ where

(F-DEST X) = (F X)

and then using our FREE inserting transformation to produce

(F-DEST X) = (PROG1 (F X) (FREE X))

since X can no longer be used. This method avoids the risk of producing several large versions of each function, but, unfortunately, it can increase the storage requirement during the evaluation of F in C_1 above that which would be required by a destructive version of F. For example, consider the following

function

```
    (MAPA X)  =  (IF (ATOM X) NIL
                      (CONS A (MAPA (CDR X))
```

which has the effect of producing a result list of the same length as its argument, L say, but with all the elements replaced with A's. If the argument to MAPA is not shared with other structure then it is permissible to merely replace in situ (with RPLACA) the elements of L with A's. However, in the above case, where we are required to use a non-destructive version of MAPA on an unshared list we will want to FREE each CONS node on L after the call to MAPA. This requires (LENGTH L) extra CONS nodes to perform the calculation of MAPA, whilst the destructive version of MAPA requires no working space.

It is now fairly clear that the choices given above represent extremes of a range of choices by which we insert destructive operations, and neither can be considered absolutely 'better' than the other. We should note at this point that we can define a continuum of behaviours intermediate to these two extremes in that we are free to choose between the two strategies given above at different points within a list structure. A further development is to achieve this effect by adding an extra parameter to each (potentially) destructive operation indicating the extent of destructiveness to be allowed on each particular invocation of the function. Thus we may have a parameterised JOIN(l,m,d) whose extremal behaviours are those of APPEND(l,m) and NCONC(l,m), as directed by the parameter d. Again this leaves the question of how detailed the information carried by d is to be; making d take only

two values could model the above situation, whereas computatibility restrictions imply the inability to model the run time behaviour exactly. Here we only observe these possibilities and do not consider their development. However the worked examples will be examined under the former strategy given above - that is, we will produce destructive versions of functions and retain their applicative version for use in situations where the destructive ones cannot be used. We note that this flavour of idea has been explored by Lang [31] in the case of the continuum of behaviours between fully eager and fully lazy evaluation.

5.11 Worked example: Derivation of NCONC from APPEND

APPEND may be defined by

```
(APPEND X Y)  =  (IF (ATOM X) Y
                     (CONS (CAR X)
                           (APPEND (CDR X) Y))).
```

Now let us suppose we have discovered, using the techniques given earlier, that each of a particular set of calls to APPEND has a first parameter which evaluates to a value which is in isolation class onelist. That is, the usage counts for the nodes X, (CDR X), ..., (CD...DR X), ... are all equal to 1. (For example the result of a MAP function is in general of class onelist.) We will now derive a destructive version of APPEND suitable for use in this case, and will call it NCONC in accordance with the standard parlance.

Firstly, let us consider the occurrence, p say, of (CDR X) in the true branch of the above. The active occurrences wrt p are (CAR X), Y and (CDR X) itself. Since X is not USED-L in any of

these, and it is not USED-R in any later occurrence (there are none), then we can embed p within (PROG1 ... (FREE X)) to derive

```
(NCONC X Y)  =  (IF (ATOM X) Y
                    (CONS (CAR X)
                          (APPEND (PROG1 (CDR X) (FREE X))
                                  Y))).
```

Now we consider the question of which version of APPEND we should use in the recursive call in this definition. Since NCONC has been defined so that X is always bound to a value in isolation class onelist, we have that the isolation environment for NCONC has R⟦X⟧ = onelist and therefore E_{isol}⟦(CDR X)⟧R in the above will also yield onelist. The other occurrences of X within the body of APPEND do not affect R since (ATOM X) is not active and

USED-L[X,(CAR X)] = h.ti

Thus our use of APPEND in the above definition can be replaced by NCONC, as we are deriving a version of APPEND which can only be applied to items of class onelist. Hence we have

```
(NCONC X Y)  =  (IF (ATOM X) Y
                    (CONS (CAR X)
                          (NCONC (PROG1 (CDR X) (FREE X))
                                 Y))).
```

This version may seem more complicated than the original version, but it has been achieved with simple transformations and is just as efficient in CONS use as the versions we will now develop, which return something of the elegance and simplicity of APPEND.

Having achieved the above we can now see that the FREE in NCONC is perfectly able to supply the location required by the CONS (replacing FREE/CONS by DCONS), and so we can write

```
(NCONC X Y)  =  (IF (ATOM X) Y
                    (DCONS X (CAR X)
                             (NCONC (CDR X) Y)))
```

thus explicitly using the node X to hold the result which was previously produced in a new CONS node.

We can produce a further optimisation of this version by using the property of the ('real') definition of DCONS in terms of RPLACA and RPLACD, given by

```
(DCONS X Y Z)  = (RPLACD (RPLACA X Y) Z).
```

This gives the identities

```
(DCONS X (CAR X) Y)  =  (RPLACD X Y)
(DCONS X Y (CDR X))  =  (RPLACA X Y),
```

thus we derive

```
(NCONC X Y)  =  (IF (ATOM X) Y
                    (RPLACD X (NCONC (CDR X) Y))).
```

This is very close to the standard definition of NCONC which can be given as NCONC-S defined by

```
(NCONC-S X Y)  =  (IF (ATOM X)
                      Y
                      (PROG2 (NCONC-S1 X Y) X))

(NCONC-S1 X Y)  =  (IF (ATOM (CDR X))
                       (RPLACD X Y)
                       (NCONC-S1 (CDR X) Y)).
```

This again requires no working space in CONS cells, but has the further advantages of using tail recursion (thus requiring no stack space either) and also uses the fact that only the last element of the list X has to be smashed with a RPLACD. NCONC-S relies much more heavily on non-applicative properties, in particular, on how smashing the final element in a list affects any sharing list (the second X in the PROG2 above). Discussion of how to achieve this final definition is outside the scope of this chapter which only considers the optimisation of CONS nodes. However it does seem to be an interesting area for research.

I am indebted to Neil Jones for the suggestion of deriving NCONC from APPEND, using the techniques developed here.

5.11.1 Producing more efficient versions of SUBST

We will now show that our techniques are capable of handling the more sophisticated example of deriving destructive versions of SUBST, and at the same time illustrate one possible extra research direction which could be pursued to enhance our capabilities for optimising store usage in a completely orthogonal direction from that given in this work.

We may define

```
(SUBST U X E)  =  (IF (ATOM E)
                      (IF (EQUAL E X) U E)
                      (CONS (SUBST U X (CAR E))
                            (SUBST U X (CDR E))))
```

or, in words, (SUBST U X E) produces a new structure in which each occurrence of the atom X in the structure E is replaced with the structure U. Note that the applicative version given here does this as wastefully as possible in that sharing is neither considered (in the sense that E may be modified in place) nor exploited (in the sense that SUBST need not create a copy of any part of E which contains no occurrences of X). This second point will be illustrated later (in MSUBST).

One possible reason why E may be unshared (and this provides another reason why we adopted a language close to LISP) is that it is often more convenient to write

```
(SUBST U 'X '(F Y (G X Y) X))
```

rather than

```
(LIST 'F 'Y (LIST 'G U 'Y) U)
```

in LISP, to produce

[F Y [G <U> Y] <U>] where <U> represents the value of U.

Note that choosing a different applicative language which allows us to write (say)

[F Y [G U Y] U]

in an environment where F, G, and Y are constants does not enable us to express these choices in our program. However the implementation must still address the space-time trade-off inherent in the choice and this provides another method by which this work could help to optimise programs in a purer applicative language.

Now let us suppose that we have identified a class of calls to SUBST in which the third argument has no external pointers (that is it is in class <u>ti</u>), then by similar arguments to those used for APPEND we may derive

```
(DSUBST U X E)  =  (IF (ATOM E)
                       (IF (EQUAL E X) U E)
                       (DCONS E (DSUBST U X (CAR E))
                                (DSUBST U X (CDR E)))))
```

which has the effect of constructing the new tree while destroying the old one.

Whilst this is as good as we can get by merely considering improving algorithms using the technique of spotting where garbage is produced, we should really note that other techniques might be useful here too (for example the production of a minimal CONS'ing SUBST routine from the applicative routine given above). The idea is that we might want a version of SUBST defined by

```
(MSUBST U X E)  =  (IF (ATOM E)
                       (IF (EQUAL E X) U E)
                       (MSUBST-1 E (MSUBST U X (CAR E))
                                   (MSUBST U X (CDR E))))

(MSUBST-1 E N1 N2)  =  (IF (AND (EQ N1 (CAR E))
                                (EQ N2 (CDR E))
                           E
                           (CONS N1 N2))
```

where we borrow from LISP the EQ function which tests for equality of locations (another dirty function which we would like to incorporate automatically rather than pollute our applicative language design by introducing the foreign concept of locations) and also the boolean AND function. This version has the great advantage of only constructing new CONS nodes for those parts of the structure which must be created - all other parts are shared with the argument E. It is true that such effects can be created by making CONS into a memo-function (originally due to Michie [32]), a technique often called hash-cons'ing because it usually requires a hash to make the associative lookup tolerable (see Goto [19] for more details). However the great drawbacks of hash-cons'ing are that they are expensive, and also one can never guarantee that a new CONS is unshared, thereby invalidating most of the work presented here as well as producing 'stray' sharing. One very promising solution which needs to be investigated is the development of work similar to that presented here, but which performs compile time hash-cons'ing. It is clear how such a scheme would work in a simple case: for example

```
(COPY X)  = (IF (ATOM X)
                X
                (CONS (COPY (CAR X)) (COPY (CDR X))))
```

can be transformed to the identity function in an applicative

language because

```
(CONS (CAR X) (CDR X))
```

is EQUAL to X is such regimes.

5.12 Syntax and Semantics of LISP-D

This section gives a (possible) semantics for our toy language LISP-D. For further details and background Mike Gordon's book [16] is to be recommended.

The semantics given below is somewhat complicated (but made more general) by the fact that we have included error handling cases in the semantics rather than merely producing $\perp$ on an error. The reader is welcome to read ? and ?? as $\perp$ if he does not care to distinguish failure, as in (CAR X) where X is an atom, from looping.

5.12.1 Notation

If X is a data class we will use X^* to stand for the class of sequences of elements of X. Correspondingly we will use <> to represent the empty sequence and <a¦x> to represent the sequence whose first element is a and x is the sequence of the remaining elements.

5.12.1.1 Data Classes

Atom = {True, False, A_1, A_2 ...}

Var = {x_1, x_2 ...}

Fun = {f_1, f_2 ...}

Token = Atom + {"[", ".", "]", Error}

Val = Atom + Loc + {?}

TupleVal = $(\text{Val} - \{?\})^*$ + {??}

Loc = {Loczero} + Succ(Loc)

Store = (Loc → Val) X (Loc → Val) X Loc

Env = VarEnv X FunEnv

VarEnv = (Var → Val)

FunEnv = Fun → (Val^* X Store) → (Val X Store)

5.12.1.2 Syntactic equations

```
Exp ::= Var | Atom |
        (CAR Exp) | (CDR Exp) | (CONS Exp Exp) |
        (ATOM Exp) | (IF Exp Exp Exp) |
        (ERROR) | (FREE Exp) |
        (Fun Exp*)


Dcl ::= (Fun Var*) = Exp
Prog ::= Dcl* Exp.
```

5.12.1.3 Semantic Functions

E: Exp → (Env X Store) → (Val X Store)

Seq: Exp^* → (Env X Store) → (TupleVal X Store)

P: Prog → Token^*

D: Dcl^* → FunEnv

Bind: Var^* → Val^* → VarEnv

Print: Val X Store → Token^*

5.12.2 Semantic Equations

$E[\![A]\!](r,s) = (A,s)$

$E[\![x]\!](\langle r_1,r_2\rangle, s) = (r_1(x), s)$

$$\begin{aligned}
E[\![(\text{CAR } e)]\!](r,s) &= \bot \text{ if } E[\![e]\!](r,s) = \bot \\
&= (s_1(v),\ s') \text{ if } v \in \text{Loc} \\
&= (?,s') \text{ otherwise} \\
&\text{where } (v,s') = E[\![e]\!](r,s) \\
&\text{and } \langle s_1,s_2,m\rangle = s'
\end{aligned}$$

$$\begin{aligned}
E[\![(\text{CDR } e)]\!](r,s) &= \bot \text{ if } E[\![e]\!](r,s) = \bot \\
&= (s_2(v),\ s') \text{ if } v \in \text{Loc} \\
&= (?,s') \text{ otherwise} \\
&\text{where } (v,s') = E[\![e]\!](r,s) \\
&\text{and } \langle s_1,s_2,m\rangle = s'
\end{aligned}$$

$$\begin{aligned}
E[\![(\text{CONS } e_1\ e_2)]\!](r,s) &= \bot \text{ if } \text{Seq}[\![\langle e_1,e_2\rangle]\!](r,s) = \bot \\
&= (m,\ \langle s_1[v_1/m],\ s_2[v_2/m],\ \text{succ}(m)\rangle) \\
&\qquad\qquad \text{if } v = \langle v_1,v_2\rangle \\
&= (?,\ s') \text{ otherwise} \\
&\text{where } (v,\ s') = \\
&\qquad \text{Seq}[\![\langle e_1,\ e_2\rangle]\!](r,s) \\
&\text{and } \langle s_1,s_2,m\rangle = s'
\end{aligned}$$

$$\begin{aligned}
E[\![(\text{ATOM } e)]\!](r,s) &= \bot \text{ if } E[\![e]\!](r,s) = \bot \\
&= (\text{True},\ s') \text{ if } v \in \text{Atom} \\
&= (\text{False},\ s') \text{ if } v \in \text{Loc} \\
&= (?,s') \text{ otherwise} \\
&\text{where } (v,s') = E[\![e]\!](r,s)
\end{aligned}$$

$$\begin{aligned}
E[\![(\text{FREE } e)]\!](r,s) &= \bot \text{ if } E[\![e]\!](r,s) = \bot \\
&= (v,\ \langle s_1[?/v],\ s_2[?/v],\ m\rangle) \text{ if } v \in \text{Loc} \\
&= (?,s') \text{ otherwise} \\
&\text{where } (v,s') = E[\![e]\!](r,s) \\
&\text{and } \langle s_1,s_2,m\rangle = s'
\end{aligned}$$

$$
\begin{aligned}
E\llbracket(\mathrm{IF}\ e_1\ e_2\ e_3)\rrbracket(r,s) &= \bot \text{ if } E\llbracket e_1\rrbracket(r,s) = \bot \\
&= E\llbracket e_2\rrbracket(r,s') \text{ if } v = \mathrm{True} \\
&= E\llbracket e_3\rrbracket(r,s') \text{ if } v = \mathrm{False} \\
&= (?,s') \text{ otherwise} \\
&\text{where } (v,s') = E\llbracket e\rrbracket(r,s)
\end{aligned}
$$

$$
E\llbracket(\mathrm{ERROR})\rrbracket(r,s) = (?,s)
$$

$$
\begin{aligned}
E\llbracket(F\ e^*)\rrbracket(r,s) &= \bot \text{ if } Seq\llbracket e^*\rrbracket(r,s) = \bot \\
&= (?,s') \text{ if } Seq\llbracket e^*\rrbracket(r,s) = (??,\ s') \\
&= r_2(F)[Seq\llbracket e^*\rrbracket(r,s)] \text{ otherwise} \\
&\text{where } \langle r_1,r_2\rangle = r
\end{aligned}
$$

$$
\begin{aligned}
Seq\llbracket\langle\rangle\rrbracket(r,s) &= (\langle\rangle,\ s) \\
Seq\llbracket\langle e \mid e^*\rangle\rrbracket(r,s) &= \bot \text{ if } E\llbracket e\rrbracket(r,s) = \bot \\
&= (??,s') \text{ if } E\llbracket e\rrbracket(r,s) = (?,s') \\
&= \bot \text{ if } Seq\llbracket e^*\rrbracket(r,s') = \bot \\
&= (??,s'') \text{ if } Seq\llbracket e^*\rrbracket(r,s') = (??,s'') \\
&= (\langle v \mid v^*\rangle,\ s'') \text{ otherwise} \\
&\text{where } (v,s') = E\llbracket e\rrbracket(r,s) \\
&\text{and } (v^*,s'') = Seq\llbracket e^*\rrbracket(r,s')
\end{aligned}
$$

$$
\begin{aligned}
Print(a,s) &= \langle\rangle \text{ if } a = \bot \\
&= \langle a\rangle \text{ if } a \in Atom \\
&= \langle\text{"["}\rangle\ @\ Print(s_1(a),s)\ @\ \langle\text{"."}\rangle\ @ \\
&\qquad Print(s_2(a),s)\ @\ \langle\text{"]"}\rangle \text{ if } a \in Loc \\
&= \langle\text{"Error"}\rangle \text{ otherwise} \\
&\text{where } \langle s_1,s_2,m\rangle = s \\
&\text{and } @: Token^* \times Token^* \rightarrow Token^* \\
&\qquad \langle\rangle\ @\ x = x \\
&\qquad \langle a \mid b\rangle\ @\ x = \langle a \mid b@x\rangle
\end{aligned}
$$

$$P[\![d^* e]\!] = \text{Print}(E[\![e]\!]\langle\langle r_0, D[\![d^*]\!]\rangle, \langle s_0, s_0, \text{LocZero}\rangle\rangle)$$

$$\text{where } r_0(x) = ?$$

$$\text{and } s_0(x) = \lambda\langle v^*, s\rangle . \langle ?, s\rangle$$

$$D[\![d^*]\!] = \text{Fix}_{\text{FunEnv}}[\lambda\, r.\ \lambda\, f.$$

$$\lambda\langle v^*, s\rangle.\ \text{Valid}(f, d^*) \rightarrow$$

$$E[\![e]\!](\langle \text{Bind}[\![X^*]\!]v^*, r\rangle, s),$$

$$(?, s)$$

where $\text{Valid}(f,d^*)$ is true iff there exists (in d^*)

a unique $d = [\![(f\ X^*) = e]\!]$

and e is as given by Valid

and $\text{Bind}[\![X^*]\!]v^*$ gives the environment induced by the match.

Although we do not allow programs to use RPLACA and RPLACD they will be used in the discussions. Their semantics are given by

$$E[\![(\text{RPLACA } e_1\ e_2)]\!](r,s)$$

$$= \perp \text{ if } \text{Seq}[\![\langle e_1, e_2\rangle]\!](r,s) = \perp$$

$$= ? \text{ if } v^* = ??$$

$$= (v_1, \langle s_1[v_2/v_1], s_2, m\rangle) \text{ if } v_1 \in \text{Loc}$$

$$= ? \text{ otherwise}$$

$$\text{where } (v^*, s') = \text{Seq}[\![\langle e_1, e_2\rangle]\!](r,s)$$

$$\text{and } \langle s_1, s_2, m\rangle = s'$$

$$\text{and } \langle v_1, v_2\rangle = v^*$$

$$E[\![(\text{RPLACD } e_1\ e_2)]\!](r,s)$$

$$= \bot \text{ if } Seq[\![<e_1,e_2>]\!](r,s) = \bot$$
$$= ? \text{ if } v^* = ??$$
$$= (v_1, <s_1, s_2[v_2/v_1], m>) \text{ if } v_1 \in Loc$$
$$= ? \text{ otherwise}$$
$$\text{where } (v^*,s') = Seq[\![<e_1,e_2>]\!](r,s)$$
$$\text{and } <s_1,s_2,m> = s'$$
$$\text{and } <v_1,v_2> = v^*$$

5.12.2.1 Semantic modifications to add usage counts

The semantic modifications to describe usage counts directly associated with each CONS node are very simple and merely consist of the following additions.

Firstly we must change the definition of Store to

Store = (Loc → Val) X (Loc → Val) X (Loc → Number) X Loc.

This provides a Store value with a component which gives the number of pointers to each location.

Secondly we must alter all semantic clauses involving Store to increment or decrement the relevant entry in the third component of Store accordingly. Since our semantic Loc is infinite we do not have to worry about actually using the information given by the usage counts, unlike the situation in a real machine.

5.12.3 A Store-less semantics for LISP-A

This section sketches the changes that have to be made to the semantic functions and objects in order to give LISP-A a semantics not involving stores.

Firstly the following changes are required to semantic categories

Val = Atom + (Val X Val) + {?}

VarEnv = Var $\rightarrow$ Val

Funenv = Fun $\rightarrow$ Val^* $\rightarrow$ Val

This of course requires that the functionality of the semantic functions changes to (we give only the important changes)

E: Exp $\rightarrow$ Env $\rightarrow$ Val

Seq: Exp^* $\rightarrow$ Env $\rightarrow$ TupleVal

Print: Val $\rightarrow$ $Token^*$

Most semantic functions remain unchanged except for the need to remove the Store components of their parameters and results. The only functions to undergo radical change are CONS (as might be expected) and FREE (which we can no longer discuss). We derive

$$\begin{aligned} E[\![(\text{CONS } e_1\ e_2)]\!]\ r &= \bot \text{ if } Seq[\![\langle e_1, e_2\rangle]\!] = \bot \\ &= ? \text{ if } Seq[\![\langle e_1, e_2\rangle]\!] = ?? \\ &= Seq[\![\langle e_1, e_2\rangle]\!] \text{ otherwise} \end{aligned}$$

Chapter 6: Conclusions

We hope to have convinced the reader, in the last few chapters, that there are techniques, well founded in theory, by which we can undertake systematic optimising transformations of applicative programs. It is now time to address side issues such as where we progress from here.

6.1 Efficiency in Applicative Languages

Much of the material in the preceding chapters has been oriented towards improving the efficiency of applicative languages. It is now desirable that we consider why this is useful.

Firstly, let us observe, for the purposes of compilation (or sophisticated interpretation), that applicative languages are a double-edged sword.

On one hand (the traditional viewpoint) applicative languages are difficult to compile efficiently because they are rather distant from the notion of a von Neumann architecture machine, the basis of all current computers.

On the other, applicatives languages offer us much greater potentialities for improved compilation due to their reduced specification of exactly how (operationally) a computation is to be performed. This is only possible for pure applicative languages and not for an applicative subset of a language (such as LISP) because in the latter case the applicative semantics has to be rather constrained in order to tie in with the sequential semantics in the rest of the language. For example, we cannot remove, or

optimise the order of evaluation of, a certain piece of code due to (the risk of) side-effects. However applicative languages greatly ease the problems of considering whether a proof or transformation is valid. Many of the most effective theorem proving or program transformation systems have an applicative target language, for example Feather's ZAP system [13], Boyer and Moore's theorem prover [3] or the Edinburgh LCF system [18].

The objection to permitting 'mixed' languages, as in the above notion of 'applicative subset', also extends to considerations of parallelism, and we will spend a few moments considering the potential of ADA [26] in this respect. ADA is an imperative language which allows the programmer to specify how a program is to be broken up into a set of co-operating processes. However the number of processes chosen will very probably depend on the particular target machine the programmer has in mind. Hence, acceptable efficiency may only be achieved on a single machine. Moreover, there are several research projects in progress developing machines containing thousands of processors. It will surely be impossible to write an ADA program which uses such a machine to its full potential. It does not even appear possible to decompose a multi-tasking program into a larger number of tasks due to the complicated semantics of full tasking (ADA's tasking will not even be formally specified). We are much more likely to be able to decompose a sequential program into tasks than one already unsuitably partitioned. We would even go further than this and argue that it will be practically impossible to incorporate

automatically a reasonable degree of tasking within a given sequential imperative program due to the difficulties of detecting whether two computations can, in fact, be performed in parallel without invalidating the semantics. The objections to the programmer performing this breakdown are firstly the machine dependency implied and secondly the extra work involved.

In an applicative language, with no notion of a global von Neumann store, we are free to evaluate any two expressions in parallel, and the implied data dependencies provide the communication between processes. We can summarise this by saying that it is very difficult to achieve acceptable parallelism in imperative languages, whereas the main problem in applicative languages seems to be that of cutting down the vast number of parallel processes that the above method of using a task for each sub-expression would generate.

6.2 Suggestions for Further Work

The first observation to be made is that the techniques described here are too oriented towards 'toy' systems. For example, the techniques for implementing call-by-need using call-by-value described in chapter 4 have not been tested by a practical implementation in a large system. Similarly, the merit of the theory of inserting destructive operators into applicative programs (chapter 5) must be decided by its application to a large system (such as the HOPE [7] system here at Edinburgh). It is not sufficient to argue that the techniques are correct - we must also show that they are applicable sufficiently often to make a

significant improvement to the performance of the system as a whole. It is to the author's regret that there has not yet been an opportunity to make such a large-scale trial.

On the other hand, we can find places where the theoretical basis is not quite satisfactory. This is not meant to imply that we consider the work unrigorous, but rather that the theory of sharing given in chapter 5) leans too heavily on computational insights rather than on an independent basis. Similarly the theoretical work on abstract interpretation for the applicative idiom described in chapter 2 is presented as a first development of the theory of abstract interpretation for this style, and inevitably will lack the elegance of a full theory developed with hindsight.

We will turn now to chapters 3 and 4 which discuss the use of eager evaluation strategies to implement lazier ones at a gain of efficiency with respect to current machine architectures. Our work presents results only for the case of a flat domain, where a parameter (or sub-computation) is either unevaluated, or evaluated to completion. As we observed in these chapters it would be highly desirable to extend such results to a fully lazy evaluation scheme where expressions may be evaluated to yield a partial result, leaving (possibly many) unevaluated sub-expressions.

References

1. Abbott, J.C. Sets, Lattices and Boolean Algebras. Allyn and Bacon, 1969.

2. Backus, J. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. Comm. ACM 21, 8 (August 1978), 613-641.

3. Boyer, R.S. and Moore, J.S. A Computational Logic. Academic Press, 1980.

4. Burstall, R.M. Some Techniques for Proving Correctness of Programs which alter Data Structures. Machine Intelligence 7 (1972), 23-50.

5. Burstall, R.M. Design Considerations for a Functional Programming Language. Infotech State of the Art Conference: The Software Revolution, Copenhagen, October, 1977.

6. Burstall, R.M. and Darlington, J. A Transformation System for Developing Recursive Programs. J. ACM 24, 1 (January 1977), 44-67.

7. Burstall, R.M., MacQueen, D. and Sannella, D.T. HOPE: an Experimental Applicative Language. Conference Record of the 1980 LISP Conference, 1980. Also internal report CSR-62-80, Dept. of Computer Science, Edinburgh University.

8. Cousot, P. and Cousot, R. Static Determination of Dynamic Properties of Programs. Proc. 2nd Int. Symp. on Programming, 1976.

9. Cousot, P. and Cousot, R. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. Proc. 4th ACM Symp. on Principles of Programming Languages, Los Angeles, 1977.

10. Cousot, P. and Cousot, R. Static Determination of Dynamic Properties of Recursive Procedures. Proc. IFIP conf. on Formal Descriptions of Programming Concepts, 1978, pp. 237-277.

11. Donzeau-Gouge, V. Utilisation de la Sémantique Dénotationelle pour l'étude d'Interprétations non-standard. IRIA-Laboria, 78150-Rocquencourt, France, 1978.

12. Donzeau-Gouge, V. Denotational Definition of Properties of Program Computations. In Muchnick, S.S. and Jones, N.D., Ed., Program Flow Analysis, Prentice-Hall, 1981.

13. Feather, M.S. A System for Developing Programs by Transformation. Ph.D. Th., University of Edinburgh, 1979.

14. Floyd, R.W. Assigning Meanings to Programs. Amer. Math. Soc. 19 (1967), 19-32.

15. Friedman, D.P. and Wise, D.S. CONS should not Evaluate its Arguments. Proc. 3rd Int. Colloq. on Automata, Languages and Programming, Edinburgh, 1976.

16. Gordon, M.J.C. The Denotational Descripion of Programming Languages. Springer-Verlag, 1979.

17. Gordon, M.J.C., Milner, A.J.R.G., Morris, L., Newey, M. and Wadsworth, C. A Metalanguage for Interactive Proof in LCF. Proc. 5th ACM Symp. on Principles of Programming Languages, Tucson, Arizona, 1978.

18. Gordon, M.J., Milner, A.J.R. and Wadsworth, C.P. Edinburgh LCF: Lecture Notes in Computer Science. Springer-Verlag, 1979.

19. Goto, E. Monocopy and Associative Algorithms in an Extended LISP. University of Tokyo, May, 1974.

20. Harel, H. On Folk Theorems. Comm. ACM 23, 7 (July 1980), 379-389. (Folk theorems have no known authors, but are widely known results.)

21. Henderson, P. and Morris, J. A Lazy Evaluator. Proc. 3rd ACM Symp. on Principles of Programming Languages, Atlanta, Georgia, 1976, pp. 95-103.

22. Hennessy, M.C.B. Power Domains and Non-determinstic Recursive Definitions. In preparation (submitted to 5th Int. Symp. on Programming).

23. Hennessy, M.C.B. and Plotkin, G.D. Full Abstraction for a simple Parallel Programming Language. Proc. 7th Int. Symp. on Mathematical Foundations of Computer Science, 1978, pp. 108-120.

24. Hoare, C.A.R. An Axiomatic Basis for Computer Programming. Comm. ACM 12, 10 (1969), 576-583.

25. Huet, G. and Levy J.-J. Call-by-need Computations in Non-ambiguous Linear Term Rewriting Systems. IRIA-Laboria, 78150-Rocquencourt, France, July, 1979.

26. Ichbiah, J.D. et al. Preliminary ADA Reference Manual. SIGPLAN Notices 14, 6A (June 1979), .

27. Jones, N.D. Flow Analysis of Lambda Expressions. DAIMI report PB-128, Dept. of Computer Science, Aarhus University, January, 1981.

28. Jones, N.D. and Muchnick, S.S. Flow Analysis and Optimisation of LISP-like Structures. TR 78-2, Dept. of Computer Science, University of Kansas, 1978.

29. Jones, N.D. and Muchnick, S.S. Complexity of Flow Analysis, Inductive Assertion Synthesis, and a Language Due to Dijkstra. Proc. 20th Conf. on Foundations of Computer Science, 1979, pp. 185-190.

30. Kernighan, B.W. and Plauger, P.J. Software Tools. Addison-Wesley, 1976.

31. Lang, B. Threshold Evaluation and the Semantics of Call by Value, Assignment and Generic Procedures. Proc. 4th ACM Symp. on Principles of Programming Languages, Los Angeles, 1977.

32. Michie, D. Memo Functions: a Language Feature with Rote Learning Properties. Research Memorandum MIP-R-29, Machine Intelligence Research Unit, Edinburgh University, 1967.

33. Milner, R. Fully Abstract Models of Typed Lambda Calculi. Theor. Comp. Sci. 4, 1 (February 1977), 1-23.

34. Mycroft, A. The Theory and Practice of Transforming Call-by-need into Call-by-value. Proc. 4th Int. Symp. on Programming: Lecture notes in Computer Science number 83, Paris, April, 1980, pp. 269-281.

35. Mycroft, A. Call-by-Need = Call-by-Value + Conditional. Internal report CSR-78-81, Dept. of Computer Science, Edinburgh University, 1981. Presented at IUCC conference at Exeter, Sept 1980.

36. Naur, P. Checking of Operand Types in ALGOL compilers. BIT 5 (1965), 151-163.

37. Nielson, F. Semantic Foundations of Data Flow Analysis. DAIMI report PB-131, Dept. of Computer Science, Aarhus University, February, 1981.

38. Pettorossi, A. Destructive Marking: A Method and some Simple Heuristics for Improving Memory Utilisation in Recursive Programs. Informatica proceedings, Bled, 1978.

39. Pettorossi, A. Improving Memory Utilisation in Transforming Recursive Programs. Proc. 7th Int. Symp. on Mathematical Foundations of Computer Science, Zakopane, Poland, 1978.

40. Plotkin, G.D. Call-by-name, Call-by-value and the Lambda Calculus. Theor. Comp. Sci. 1, 2 (December 1975), 125-159.

41. Plotkin, G.D. A Powerdomain Construction. SIAM J. Comput. 5, 3 (1976), 452-487.

42. Plotkin, G.D. A Structural Approach to Operational Semantics. Lecture notes 1981, Dept. of Computer Science, Aarhus University.

43. de Roever, W.P. First Order reduction of Call-by-name to Call-by-value. Proving and Improving Programs, Arc et Senans, 1975.

44. Schwarz, J. Using Annotations to make Recursion Equations behave. DAI research report 43, Dept. of Artificial Intelligence, Edinburgh University, 1977. Revised 1981 at Bell Labs, to appear.

45. Schwarz, J. Verifying the Safe Use of Destructive Operators in Applicative Programs. Proc. 3rd Int. Symp. on Programming, Paris, 1978. Also published as DAI research report 55, Dept. of Artificial Intelligence, Edinburgh University.

46. Schwarz, J. Destructive Operations in Applicative Languages. Unpublished manuscript.

47. Sintzoff, M. Calculating Properties of Programs by Valuations on Specific Models. Proceedings on an ACM conference on Proving Assertions about Programs, Las Cruces, Mexico, January, 1972.

48. Strachey, C. and Wadsworth, C.P. Continuations - a Mathematical Semantics for Handling Full Jumps. Technical monograph PRG11, Programming Research Group, Oxford University, 1974.

49. Thatcher, J.W. Tree Automata: an Informal Survey. In Currents in the Theory of Computing, Prentice-Hall, 1973, pp. 143-172.

50. Vuillemin J. Correct and optimal implementations of recursion in a simple programming language. Journal of Computer and System Sciences 9 (1974), 332-354. Also PhD thesis: Proof techniques for recursive programs (chapter 2).

51. Wadsworth, C.P. *Semantics and Pragmatics of the Lambda Calculus*. Ph.D. Th., Programming Research Group, Oxford University, 1971.

52. Wegbreit, B. Property Extraction in Well Founded Property Sets. *IEEE Trans. on Software Eng. SE-1*, 3 (September 1975), 270-285.