Compile-Time Optimisation of Store Usage in Lazy Functional Programs

Geoffrey William Hamilton

Department of Computing Science and Mathematics

University of Stirling

Submitted in partial fulfilment
of the requirements for the
degree of Doctor of Philosophy

October 1993

"For Summer has o'erbrimm'd their clammy cells.
Who hath not seen thee oft amid thy store?"

John Keats, *Ode to Autumn*

Abstract

Functional languages offer a number of advantages over their imperative counterparts. However, a substantial amount of the time spent on processing functional programs is due to the large amount of storage management which must be performed. Two apparent reasons for this are that the programmer is prevented from including explicit storage management operations in programs which have a purely functional semantics, and that more readable programs are often far from optimal in their use of storage. Correspondingly, two alternative approaches to the optimisation of store usage at compile-time are presented in this thesis.

The first approach is called *compile-time garbage collection*. This approach involves determining at compile-time which cells are no longer required for the evaluation of a program, and making these cells available for further use. This overcomes the problem of a programmer not being able to indicate explicitly that a store cell can be made available for further use. Three different methods for performing compile-time garbage collection are presented in this thesis; compile-time garbage marking, explicit deallocation and destructive allocation. Of these three methods, it is found that destructive allocation is the only method which is of practical use.

The second approach to the optimisation of store usage is called *compile-time garbage avoidance*. This approach involves transforming programs into semantically equivalent programs which produce less garbage at compile-time. This attempts to overcome the problem of more readable programs being far from optimal in their use of storage. In this thesis, it is shown how to guarantee that the process of compile-time garbage avoidance will terminate.

Both of the described approaches to the optimisation of store usage make use of the information obtained by *usage counting analysis*. This involves counting the number of times each value in a program is used. In this thesis, a reference semantics is defined against which the correctness of usage counting analyses can be proved. A usage counting analysis is then defined and proved to be correct with respect to this reference semantics. The information obtained by this analysis is used to annotate programs for compile-time garbage collection, and to guide the transformation when compile-time garbage avoidance is performed.

It is found that compile-time garbage avoidance produces greater increases in efficiency than compile-time garbage collection, but much of the garbage which can be collected by compile-time garbage collection cannot be avoided at compile-time. The two approaches are therefore complementary, and the expressions resulting from compile-time garbage avoidance transformations can be annotated for compile-time garbage collection to further optimise the use of storage.

Declaration

I hereby declare that this thesis has been composed by myself, that the work reported has not been presented for any university degree before, and that the ideas I do not attribute to others are due to myself.

Geoffrey Hamilton
October 1993

Acknowledgements

The completion of this thesis was dependent on many different people. I would like to thank everyone who encouraged me, but I apologise if I do not mention them by name.

Firstly, Simon Jones, must be acknowledged for his supervision and encouragement during the course of this work. The Computing Science department at Stirling University must also be thanked for providing a friendly working atmosphere. Special thanks go to the technical support team, Graham, Sam and Catherine, and to the secretaries, Jane, Moira and Muriel for their help over the years.

The functional programming group in Glasgow is acknowledged for providing a stimulating atmosphere for research, and for accepting me into their fold. In particular, I would like to acknowledge Phil Wadler, whose ideas have inspired much of the work in this thesis.

The Department of Education for Northern Ireland must also be acknowledged for their financial support during the course of this work, and for funding my trips abroad.

On a personal note, I would like to thank my friends at Stirling for their help and encouragement, and for keeping me sane. Paul Gibson deserves a special mention for his friendship over the years, and for helping me to complete this thesis by proof reading it for me, and for giving me a roof over my head while I finished it off.

I would also like to thank my father for his encouragement from across the Irish Sea over the years. He was always ready to listen and to help in whatever way he could. I don't think I could ever repay him.

Finally, and most importantly, I would like to thank my fiancée Sam for her strength and devotion over the years. She was always prepared to listen to my problems, and help me through them, even in the darkest moments. This thesis would not have been completed without her help. I don't know why she has stuck by me over the years, but I'm just glad that she has. This thesis is dedicated to her.

Contents

1 Introduction **1**

1.1 Compile-Time Optimisation . . . 2
1.1.1 Static Analysis . . . 2
1.1.2 Program Transformation . . . 3
1.1.3 Desirable Criteria for Compile-Time Optimisations . . . 3
1.2 Compile-Time Optimisation of Store Usage . . . 4
1.2.1 Compile-Time Garbage Detection . . . 4
1.2.2 Compile-Time Garbage Collection . . . 4
1.2.3 Compile-Time Garbage Avoidance . . . 5
1.3 Thesis Contribution . . . 5
1.3.1 Compile-Time Garbage Detection . . . 6
1.3.2 Compile-Time Garbage Collection . . . 6
1.3.3 Compile-Time Garbage Avoidance . . . 6
1.4 Thesis Outline . . . 7

2 Language **9**

2.1 Notation . . . 10
2.2 Syntax . . . 10
2.3 Standard Semantics . . . 14
2.4 Store Semantics . . . 16
2.5 Congruence . . . 21
2.6 Related Work . . . 23
2.7 Conclusion . . . 24

3 Compile-Time Garbage Detection **25**

3.1 Usage Counting Store Semantics . . . 26
3.2 Usage Patterns . . . 28
3.3 Operations on Usage Patterns . . . 34
3.4 Usage Counting Analysis . . . 37
3.5 Proof of Correctness . . . 40
3.6 Examples . . . 43
3.7 Related Work . . . 45
3.7.1 Abstract Interpretation . . . 45
3.7.2 Backward Analysis . . . 46
3.7.3 Type Inference . . . 47
3.8 Conclusion . . . 48

4 Compile-Time Garbage Collection **49**
4.1 Run-Time Garbage Collection . . . 50
4.1.1 Reference Counting Garbage Collection . . . 50
4.1.2 Mark/Scan Garbage Collection . . . 52
4.1.3 Copying Garbage Collection . . . 52
4.2 Compile-Time Garbage Marking . . . 54
4.2.1 Annotating Programs for Compile-Time Garbage Marking . . . 54
4.2.2 Compile-Time Garbage Marking Store Semantics . . . 55
4.2.3 Correctness . . . 57
4.3 Explicit Deallocation . . . 61
4.3.1 Annotating Programs for Explicit Deallocation . . . 61
4.3.2 Explicit Deallocation Store Semantics . . . 62
4.3.3 Correctness . . . 64
4.4 Destructive Allocation . . . 67
4.4.1 Annotating Programs for Destructive Allocation . . . 68
4.4.2 Destructive Allocation Store Semantics . . . 69
4.4.3 Correctness . . . 71
4.5 Related Work . . . 74
4.5.1 Compile-Time Garbage Marking . . . 74
4.5.2 Explicit Deallocation . . . 75
4.5.3 Destructive Allocation . . . 76
4.6 Conclusion . . . 78

5 Compile-Time Garbage Avoidance **79**
5.1 Deforestation . . . 80
5.1.1 Treeless Form . . . 81
5.1.2 The Deforestation Algorithm . . . 82
5.1.3 The Deforestation Theorem . . . 86
5.2 Extended Deforestation . . . 90
5.2.1 Transient Structures . . . 90
5.2.2 Accumulating Parameters . . . 93
5.2.3 Shared Values . . . 94
5.2.4 Extended Treeless Form . . . 95
5.2.5 The Extended Deforestation Theorem . . . 95
5.3 Generalised Deforestation . . . 99
5.3.1 Generalised Treeless Form . . . 100
5.3.2 The Generalised Deforestation Algorithm . . . 101
5.3.3 The Generalised Deforestation Theorem . . . 103
5.4 Related Work . . . 106
5.4.1 Deforestation . . . 106
5.4.2 Extended Deforestation . . . 107
5.4.3 Generalised Deforestation . . . 108
5.5 Conclusion . . . 109

6 Conclusion **110**
6.1 Summary of Thesis . . . 111
6.1.1 Language . . . 111
6.1.2 Compile-Time Garbage Detection . . . 111
6.1.3 Compile-Time Garbage Collection . . . 112
6.1.4 Compile-Time Garbage Avoidance . . . 112
6.2 Further Work . . . 113
6.2.1 Compile-Time Garbage Detection . . . 113
6.2.2 Compile-Time Garbage Collection . . . 113
6.2.3 Compile-Time Garbage Avoidance . . . 114
6.3 General Conclusions . . . 115

References **116**

Appendices **123**

A Proofs for Language Semantics **124**
A.1 Congruence of Expressions . . . 124
A.2 Congruence of Function Variable Environments . . . 129

B Proofs for Compile-Time Garbage Detection **131**
B.1 Correctness of Usage Counting Analysis . . . 131
B.2 Correctness of Usage Counting Analysis Function Variable Environment . . . 138

C Proofs for Compile-Time Garbage Avoidance **140**
C.1 Proof of Deforestation Theorem . . . 140
C.1.1 Proof of Lemma 5.1.3 . . . 140
C.1.2 Proof of Lemma 5.1.4 . . . 145
C.1.3 Proof of Lemma 5.1.5 . . . 148
C.1.4 Proof of Lemma 5.1.10 . . . 152
C.1.5 Proof of Lemma 5.1.11 . . . 155
C.2 Proof of Extended Deforestation Theorem . . . 159
C.2.1 Proof of Lemma 5.2.12 . . . 159
C.2.2 Proof of Lemma 5.2.13 . . . 163
C.3 Proof of Generalised Deforestation Theorem . . . 173
C.3.1 Proof of Lemma 5.3.3 . . . 173
C.3.2 Proof of Lemma 5.3.4 . . . 175
C.3.3 Proof of Lemma 5.3.8 . . . 176

List of Figures

2.1 Abstract Syntax . . . 11
2.2 Example Function Definitions . . . 12
2.3 Standard Semantic Domains . . . 13
2.4 Standard Semantic Functions . . . 14
2.5 Standard Semantics . . . 15
2.6 Standard Semantics (auxiliary functions) . . . 16
2.7 Store Semantic Domains . . . 17
2.8 Store Semantic Functions . . . 18
2.9 Store Semantics . . . 19
2.10 Store Semantics (continued) . . . 20
2.11 Store Semantics (auxiliary functions) . . . 21

3.1 Usage Counting Store Semantic Domains . . . 27
3.2 Usage Counting Store Semantic Functions . . . 28
3.3 Usage Counting Store Semantics . . . 29
3.4 Usage Counting Store Semantics (continued) . . . 30
3.5 Usage Counting Store Semantics (auxiliary functions) . . . 31
3.6 The Domain of Usage Patterns $U(list\ T_A)$ . . . 33
3.7 Usage Counting Analysis Domains . . . 37
3.8 Usage Counting Analysis Functions . . . 38
3.9 Usage Counting Analysis . . . 39

4.1 Annotation of *accreverse* (*append xs ys*) *zs* for Compile-Time Garbage Marking 55
4.2 Compile-Time Garbage Marking Store Semantic Domains . . . 56
4.3 Compile-Time Garbage Marking Store Semantic Functions . . . 57
4.4 Compile-Time Garbage Marking Store Semantics . . . 58
4.5 Compile-Time Garbage Marking Store Semantics (continued) . . . 59
4.6 Compile-Time Garbage Marking Store Semantics (auxiliary functions) . . . 60
4.7 Annotation of *accreverse* (*flatten xss*) *ys* for Explicit Deallocation . . . 62
4.8 Explicit Deallocation Store Semantic Domains . . . 63
4.9 Explicit Deallocation Store Semantic Functions . . . 64
4.10 Explicit Deallocation Store Semantics . . . 65
4.11 Explicit Deallocation Store Semantics (continued) . . . 66
4.12 Explicit Deallocation Store Semantics (auxiliary functions) . . . 67
4.13 Annotation of *append* (*reverse xs*) (*Cons x Nil*) for Destructive Allocation . . . 69
4.14 Destructive Allocation Store Semantic Domains . . . 70
4.15 Destructive Allocation Store Semantic Functions . . . 71

4.16 Destructive Allocation Store Semantics 72
4.17 Destructive Allocation Store Semantics (continued) 73
4.18 Destructive Allocation Store Semantics (auxiliary functions) 74

5.1 Transformation Rules for Deforestation 82
5.2 Deforestation of *append* (*append xs ys*) *zs* 83
5.3 Result of Deforestation of *append* (*append xs ys*) *zs* 84
5.4 Modified Transformation Rules for Deforestation 85
5.5 Definition of the Size of Expressions 88
5.6 Grammar of Expressions Encountered During Deforestation 89
5.7 Deforestation of *append* (*flatten xss*) 91
5.8 Result of Deforestation of *append* (*flatten xss*) *ys* 92
5.9 Deforestation of *accreverse xs ys* 93
5.10 Grammar of Expressions Encountered During Extended Deforestation 98
5.11 Additional Transformation Rules for the Generalised Deforestation Algorithm . 101
5.12 Generalised Deforestation of *accreverse* (*flatten xss*) *ys* 102
5.13 Result of Generalised Deforestation of *accreverse* (*flatten xss*) *ys* 103

List of Tables

3.1 Usage Counting Analysis of the Function *append* 43
3.2 Usage Counting Analysis of the Function *reverse* 44
3.3 Usage Counting Analysis of the Function *accreverse* 44
3.4 Usage Counting Analysis of the Function *flatten* 45

Chapter 1

Introduction

In recent years, there has been a growing interest in *functional languages.* Functional languages offer a number of advantages over their imperative counterparts (Hughes, 1989). The special characteristic of functional languages which gives them such desirable properties is the fact that they contain no *side-effects.* This means that a function call has no effect other than to calculate its result. An expression can be evaluated at any time, since no side-effect can change its value. Expressions can therefore be evaluated in any order, and the programmer does not need to worry about the flow of control. Thus programs may be written which resemble the structure of the original problem without making detailed implementation decisions. Also, programs will be *referentially transparent.* This means that variables in an expression can be replaced by their values, and vice versa. Functional programs are therefore easier to reason about mathematically, and are more amenable to transformation, than traditional imperative languages.

The use of lazy evaluation (Henderson & Morris, 1976; Friedman & Wise, 1976) within functional languages offers additional advantages. For example, it allows for greater modularisation within programs (Hughes, 1989). The majority of functional languages which have appeared in recent years use lazy evaluation (for example LML (Augustsson, 1984), Miranda[1] (Turner, 1985) and Haskell (Hudak & Wadler, 1990)).

However, functional languages also have their disadvantages. For example, a substantial amount of the time spent on processing functional programs is due to the large amount of heap storage management which must be performed. The aim of this thesis is to investigate how the amount of store usage implied by lazy functional programs can be reduced at compile-time.

[1]Miranda is a trademark of Research Software Ltd.

1.1 Compile-Time Optimisation

As mentioned earlier, functional programs are easy to reason about mathematically, and are amenable to transformation. Compile-time optimisations can therefore make use of static analysis and program transformation.

1.1.1 Static Analysis

Static analysis involves the analysis of programs to determine their properties without actually executing them. This is done by defining abstract domains which are simpler than the standard semantic domains of the program language. These abstract domains have a structure which reflects the property of the program which is being analysed, and usually give the minimum information required to encapsulate this property. There are three main frameworks which can be used to perform static analysis of functional programs. These are abstract interpretation (or forward analysis), backward analysis[2] and type inference. The framework which is used in this thesis is the backward analysis framework.

Abstract interpretation (Cousot & Cousot, 1977; Mycroft, 1981; Abramsky & Hankin, 1987) involves associating an abstract function with each function in a program. These abstract functions are applied to information about their arguments to give information about their results. Thus the flow of information is forwards, from function arguments to function results.

Backward analysis (Hughes, 1988) also involves associating abstract functions with each function in a program. These abstract functions are applied to information about their results to give information about their arguments. Thus the flow of information is backwards, from function results to function arguments.

Using type inference to perform static analysis involves defining a non-standard type system to infer the required information from a program. This approach has the advantage that there already exist efficient algorithms for checking and inferring types (Milner, 1978; Hindley, 1979; Damas & Milner, 1982). Examples of type inference schemes for performing static analysis are described in (Wadler, 1990c) and (Kuo & Mishra, 1989). Since program logics are used to define these type inference schemes, the flow of information takes place in both forward and backward directions.

[2]The distinction between forward and backward analysis is not clear, since a backward analysis can be expressed as an abstract interpretation in which the abstract values of expressions are functions from contexts to the variables in them.

1.1.2 Program Transformation

Program transformation involves transforming programs to other programs which exhibit the same semantic behaviour, but which have hopefully been improved in some way. There are two different approaches to program transformation. These are the algebraic approach and the operational approach.

The algebraic approach to program transformation is based on the application of axioms and theorems which equate expressions and function definitions having certain structures. Thus, in a functional program, expressions may be re-written by more efficient equivalent expressions which are given by one of these theorems. This approach requires a new theorem to be invented for each new class of transformation which is required.

The operational approach to program transformation involves using a small set of meaning preserving rules for generating new recursion equations. An example of this approach is the unfold/simplify/fold program transformation methodology described in (Burstall & Darlington, 1977). Unfolding replaces a function call with the function body containing the appropriate parameter substitutions. Folding replaces an expression which matches a function body with a corresponding function call. Simplification is achieved through the application of a small set of meaning preserving rules for generating new equations which are hopefully more efficient than the original recursion equations.

The operational approach to program transformation is taken in this thesis to reduce the amount of garbage produced at run-time. Examples of algebraic transformation methods which seek to reduce the amount of garbage produced at run-time are described in (Wadler, 1981; Bellegarde, 1986; Gill *et al.* , 1993).

1.1.3 Desirable Criteria for Compile-Time Optimisations

Compile-time optimisations should satisfy the following criteria:

Termination : the process of optimisation must be finite;

Automatability : it must be possible to perform the optimisations automatically;

Correctness : unoptimised and optimised programs must produce the same results.

Termination and automatability must be guaranteed if the optimisations are to be of any use during compilation. Correctness is a vital criterion because users will have no confidence in the optimisations being performed if this is not assured. To show that the compile-time

optimisations of store usage which are presented in this thesis are correct, a reference is required against which their correctness can be proved. Non-standard store semantics are therefore defined which model the use of store in possible implementations of the language for which the described optimisations are performed.

1.2 Compile-Time Optimisation of Store Usage

Two apparent reasons why functional programs are such heavy consumers of storage are that the programmer is prevented from including explicit memory management operations in programs which have a purely functional semantics, and more readable programs are often far from optimal in their use of storage. Consequently, two alternative approaches to the optimisation of store usage at compile-time are presented. These are *compile-time garbage collection* and *compile-time garbage avoidance*. Before these optimisations are performed, the cells which will become garbage within a program are determined. This is called *compile-time garbage detection*.

1.2.1 Compile-Time Garbage Detection

Compile-time garbage detection involves determining at compile-time which cells in a program will become garbage. A cell will become garbage during the evaluation of an expression if it is unshared when it loses a reference. To determine whether a cell is unshared, a *usage counting* analysis is defined. This analysis determines the number of times a cell will be used in future computations within a program. If a cell is used only once after it has been created, then it is unshared.

1.2.2 Compile-Time Garbage Collection

Compile-time garbage collection involves determining at compile-time which store cells are no longer required for the evaluation of a program, and making these cells available for further use. This overcomes the problem of a programmer not being able to indicate explicitly that a memory cell can be made available for further use. Programs are annotated at compile-time to allow garbage cells to be collected automatically at run-time. The garbage collection itself does not actually take place at compile-time, so the term 'compile-time garbage collection' is misleading. However, this is the term which has been used for this kind of optimisation in the past, so it is used again in this thesis. Three methods for performing compile-time garbage

collection in lazy languages are presented. These are called *compile-time garbage marking*, *explicit deallocation* and *destructive allocation*. Compile-time garbage marking involves marking those cells which will become garbage after their first use. Explicit deallocation involves explicitly returning cells to the memory manager at a particular point in a program. Destructive allocation involves reusing cells directly for further allocations within a program.

1.2.3 Compile-Time Garbage Avoidance

Compile-time garbage avoidance involves transforming programs to other programs which exhibit the same semantic behaviour, but produce less garbage at run-time. This overcomes the problem of more readable programs being far from optimal in their use of storage. As mentioned earlier, the use of lazy evaluation allows for greater modularisation within programs. Functions can be defined in terms of smaller and simpler functions which are 'glued' together to give the required definition. These smaller functions are easier to define and reuse, but they often form a structure as a result, or decompose a structured argument into its constituent elements, or both. When these functions are put together to form compound expressions, many structures are formed only to be decomposed again. As described in (Wadler, 1990b), these intermediate structures are the 'glue' which hold the functions together. The use of these intermediate structures aids clarity, but it results in inefficiency at run-time. Each intermediate structure must be allocated, traversed and subsequently deallocated. These compound expressions can be transformed instead to avoid the building of intermediate structures. This is the approach which is taken in this thesis using the *deforestation* algorithm presented in (Wadler, 1990b).

1.3 Thesis Contribution

The main contribution of this thesis is to show how the amount of store usage implied by lazy functional programs can be reduced at compile-time by making use of the information obtained by usage counting analysis. This is intended to be a theoretical study rather than a practical study. The implementation and efficiency of each process of optimisation is therefore not considered. The methods which are used to optimise store usage are compile-time garbage collection and compile-time garbage avoidance. The kinds of expressions which can be optimised by each method are characterised, thus allowing comparisons to be drawn between them. The contributions of the thesis to the areas of compile-time garbage detection,

collection and avoidance are summarised below.

1.3.1 Compile-Time Garbage Detection

In most of the previous work in the area of compile-time garbage detection, the correctness of the static analyses which are used to detect garbage is not considered. In this thesis, a reference semantics is defined against which the correctness of these static analyses can be proved. A static analysis which can be used at compile-time to detect which cells in a program will become garbage is then defined, and is proved to be correct with respect to this reference semantics. It is then shown how the information obtained by this analysis can be used to allow various optimisations of store usage to be performed.

1.3.2 Compile-Time Garbage Collection

Most of the previous work in the area of compile-time garbage collection has been for strict languages. Not so much work has been done for lazy languages. Three different methods for performing compile-time garbage collection in lazy languages are therefore presented; compile-time garbage marking, explicit deallocation and destructive allocation. Of these three methods, it is found that destructive allocation is the only method which is of practical use. The correctness of the methods for performing compile-time garbage collection described in this thesis is considered. In the majority of previous work in the area of compile-time garbage collection, the correctness of the optimisations which are performed is not considered.

1.3.3 Compile-Time Garbage Avoidance

It has already been shown in (Wadler, 1990b) how compile-time garbage avoidance can be performed for lazy languages using the deforestation algorithm, and a sketch proof was given for the deforestation theorem stated in that work. This sketch proof is fleshed out in this thesis. It was also noted in the conclusion of (Wadler, 1990b) that the class of expressions for which the deforestation algorithm is guaranteed to terminate could be extended. This is what has been achieved in this thesis by making use of the information obtained by usage counting analysis. The work in this thesis therefore contributes to the understanding of when the deforestation algorithm will terminate.

1.4 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2 : the syntax and semantics of the language which will be used throughout this thesis are presented. The language is a simple lazy first order language with recursion equations and list operators. Non-standard store semantics are then defined for the language. Since these semantics will be used as a reference against which the store-related analysis and optimisations presented in this thesis can be proved correct, they are shown to be congruent to the standard semantics of the language.

Chapter 3 : it is shown how the cells which will become garbage within a program can be detected at compile-time. The store semantics defined in the previous chapter are augmented to incorporate usage counting. This involves counting the number of times each value in the store is used. Usage counting values in these semantics are then abstracted to usage patterns to allow usage counts to be determined at compile-time. A usage counting analysis is defined using these patterns to determine at compile-time the number of times each part of a value will be used in future computations. This analysis is then proved to be correct with respect to the usage counting store semantics.

Chapter 4 : it is shown how information obtained from usage counting analysis can be used to annotate programs for compile-time garbage collection. Three different methods for compile-time garbage collection are presented. The first method is called compile-time garbage marking, which involves marking cells at their allocation to indicate that they will become garbage after their first use. The second method is called explicit deallocation, which involves explicitly returning cells to the memory manager at a particular point in a program. The third and final method for compile-time garbage collection is called destructive allocation, which involves reusing cells directly for further allocations within a program. Store semantics are defined for programs which have been annotated for each of these methods for compile-time garbage collection, and the correctness of these store semantics is considered.

Chapter 5 : it is shown how information obtained from usage counting analysis can be used to guide the transformation when compile-time garbage avoidance is performed. The method which is used for avoiding the production of garbage at compile-time is the deforestation transformation algorithm described in (Wadler, 1990b). A treeless form of

function definition which does not create any intermediate structures is characterised in (Wadler, 1990b), and a sketch proof is given that the deforestation algorithm will always terminate for expressions in which all functions have definitions which are in this treeless form. This sketch proof is fleshed out in this chapter. The deforestation algorithm will also terminate for expressions in which some functions have definitions which are not in this treeless form. It is shown how this treeless form can be extended by making use of information obtained by usage counting analysis. It is then proved that the deforestation algorithm will always terminate for expressions in which all functions have definitions in this extended treeless form. Some intermediate structures can still be eliminated from expressions in which some functions have definitions which are not in this extended treeless form. It is therefore shown how any function definition can be generalised in such a way that it will be in extended treeless form. The deforestation algorithm is also extended to be able to cope with these generalisations. It is then proved that this generalised deforestation algorithm will always terminate for expressions in which all functions have definitions which have been generalised in the described manner.

Chapter 6 : a summary of the achievements of this thesis is given, directions for further work are discussed, and general conclusions are drawn.

Chapter 2

Language

In this chapter, the syntax and semantics of the language which will be used throughout this thesis are presented. The language is a simple first order lazy functional language with list operators and recursion equations. To show that the store-related analysis and optimisations presented in this thesis are correct, a reference must be provided against which their correctness can be proved. The standard semantics of the language do not model the use of store, so they cannot be used to provide this reference. Non-standard store semantics are therefore defined for the language and are shown to be congruent to the standard semantics.

The remainder of this chapter is structured as follows:

- **Section 2.1**: some of the notation which is used throughout this thesis is described.
- **Section 2.2**: the abstract syntax of the language is defined.
- **Section 2.3**: the standard semantics of the language are defined.
- **Section 2.4**: non-standard store semantics are defined for the language.
- **Section 2.5**: the store semantics defined in the previous section are shown to be congruent to the standard semantics for the language.
- **Section 2.6**: related work is considered.
- **Section 2.7**: a summary of this chapter is given.

2.1 Notation

In this section, some of the notation which is used throughout this thesis is described. It is assumed that the reader is familiar with domain theory. For a given domain D, the bottom element of the domain is represented by $\bot_D$, and the elements of the domain are ordered by the partial order $\sqsubseteq_D$. The notation $D_\bot$ represents the lifting of the domain D to add a new bottom element $\bot$. The operators $\oplus$, $\times$ and $\rightarrow$ are the coalesced sum, product and function space constructors respectively.

Tuples of elements are represented by $(v_1, \ldots, v_n)$. Elements of a tuple can be accessed using the $\downarrow$ operator, where T $\downarrow$ n denotes the nth element of the tuple T.

The notation D^* represents zero or more function arguments which are elements of the domain D. Thus the function type $\text{D}^* \rightarrow \text{E}$ is a shorthand notation for $\text{D} \rightarrow \ldots \rightarrow \text{D} \rightarrow \text{E}$.

2.2 Syntax

In this section, the abstract syntax of the language which is used throughout this thesis is defined. The language is a simple first order lazy functional language with list operators and recursion equations. The abstract syntax is shown in Figure 2.1. Programs in the language consist of an expression to evaluate and a set of function definitions. Nested function definitions are not allowed in the language. Programs involving nested function definitions

$pr \in$ Prog	$::=$	e		
		where		
		$f_1\ v_{11} \ldots v_{1k_1} \quad = \quad e_1$		Program
		$\vdots$		
		$f_n\ v_{n1} \ldots v_{nk_n} \quad = \quad e_n$		
$e \in$ Exp	$::=$	k		Expression
	$\mid$	v		
	$\mid$	$b\ e_1 \ldots e_n$		
	$\mid$	$c\ e_1 \ldots e_n$		
	$\mid$	$f\ e_1 \ldots e_n$		
	$\mid$	**case** e_0 **of** $p_1 : e_1 \mid \ldots \mid p_k : e_k$		
$k \in$ Num	$::=$	$0 \mid 1 \mid -1 \mid \ldots$		Constant
$v \in$ Bv				Bound Variable
$b \in$ Bas	$::=$	$+ \mid - \mid < \mid = \mid \ldots$		Basic Function
$c \in$ Con	$::=$	$\mathit{True} \mid \mathit{False} \mid \mathit{Nil} \mid \mathit{Cons}$		Constructor
$f \in$ Fv				Function Variable
$p \in$ Pat	$::=$	$c\ v_1 \ldots v_n$		Pattern

Figure 2.1: Abstract Syntax

can be transformed into this restricted form of program using a technique called *lambda lifting* (Johnsson, 1985). Some example function definitions are given in Figure 2.2.

The language is monomorphically typed, and it is assumed that all programs in the language are well-typed. Values in the language can have the following types:

$$
\begin{array}{lcll}
T & ::= & \mathit{int} & \text{Integers} \\
 & \mid & \mathit{bool} & \text{Booleans} \\
 & \mid & \mathit{list}\ T & \text{Lists}
\end{array}
$$

The only constants in the abstract syntax of the language are integers. Bound variables and function variables in the language are represented by strings of characters, and are elements of the domains Bv and Fv respectively. The basic functions are the built-in functions of the language and operate on integers only. The comparison of lists using the basic equality function is therefore not allowed, but it is possible to determine the equality of lists recursively

$$
\begin{array}{lcl}
\mathit{append} & : & \mathit{list\ int} \rightarrow \mathit{list\ int} \rightarrow \mathit{list\ int} \\
\\
\mathit{append\ xs\ ys} & = & \textbf{case}\ \mathit{xs}\ \textbf{of} \\
 & & \quad \begin{array}{lcl} \mathit{Nil} & : & \mathit{ys} \\ \mathit{Cons\ x\ xs} & : & \mathit{Cons\ x\ (append\ xs\ ys)} \end{array} \\
\\
\mathit{flatten} & : & \mathit{list\ (list\ int)} \rightarrow \mathit{list\ int} \\
\\
\mathit{flatten\ xss} & = & \textbf{case}\ \mathit{xss}\ \textbf{of} \\
 & & \quad \begin{array}{lcl} \mathit{Nil} & : & \mathit{Nil} \\ \mathit{Cons\ xs\ xss} & : & \mathit{append\ xs\ (flatten\ xss)} \end{array} \\
\\
\mathit{reverse} & : & \mathit{list\ int} \rightarrow \mathit{list\ int} \\
\\
\mathit{reverse\ xs} & = & \textbf{case}\ \mathit{xs}\ \textbf{of} \\
 & & \quad \begin{array}{lcl} \mathit{Nil} & : & \mathit{Nil} \\ \mathit{Cons\ x\ xs} & : & \mathit{append\ (reverse\ xs)\ (Cons\ x\ Nil)} \end{array} \\
\\
\mathit{accreverse} & : & \mathit{list\ int} \rightarrow \mathit{list\ int} \rightarrow \mathit{list\ int} \\
\\
\mathit{accreverse\ xs\ ys} & = & \textbf{case}\ \mathit{xs}\ \textbf{of} \\
 & & \quad \begin{array}{lcl} \mathit{Nil} & : & \mathit{ys} \\ \mathit{Cons\ x\ xs} & : & \mathit{accreverse\ xs\ (Cons\ x\ ys)} \end{array}
\end{array}
$$

Figure 2.2: Example Function Definitions

within the language. Basic function applications will be expressed in infix notation throughout the course of this thesis.

Booleans are represented by the values *True* and *False*. Note that booleans are considered to be constructors in the abstract syntax of the language. This is so that pattern matching can be performed upon them, since pattern matching is allowed only on constructors. The conditional can therefore be expressed as follows:

$$\textbf{case}\ e_0\ \textbf{of}\ \mathit{True} : e_1 \mid \mathit{False} : e_2$$

This has the same meaning as the more traditional form of conditional:

$$\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2$$

Empty lists are represented by *Nil* and non-empty lists are represented by an expression of the form *Cons* e_1 e_2, where the head of the list is denoted by e_1, and the tail of the list is denoted by e_2. Lists are decomposed using a **case** expression of the following form:

$$\textbf{case } e_0 \textbf{ of } \textit{Nil} : e_1 \mid \textit{Cons } v_1 \; v_2 : e_2$$

In the expression e_2, the head of the list e_0 is represented by the variable v_1, and the tail of this list is represented by the variable v_2. There is therefore no need to add explicit head and tail operators to the basic functions.

Within **case** expressions of the following form:

$$\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k$$

e_0 is called the *selector*, and $p_1 : e_1, \ldots, p_k : e_k$ are called the *branches*. The branches in a **case** expression can either be separated by the | character or by a newline character. The patterns used in the branches of **case** expressions may not be nested. Methods to transform **case** expressions with nested patterns into ones without nested patterns are described in (Augustsson, 1985) and (Wadler, 1987b).

The intended evaluation mechanism for the language is lazy evaluation. However, the basic functions are strict in all their arguments. Also, pattern matching is strict, and when a **case** expression is evaluated, the selector is evaluated to head normal form before the appropriate branch of the **case** expression is evaluated.

$$\begin{array}{lcl}
x \in \mathrm{Val}_{\mathcal{E}} & = & \mathrm{Atom} \oplus \mathrm{List} \\
\mathrm{Atom} & = & \mathrm{Int} \oplus \mathrm{Bool} \\
\mathrm{Int} & = & \{0\}_\perp \oplus \{1\}_\perp \oplus \{-1\}_\perp \oplus \ldots \\
\mathrm{Bool} & = & \{\mathrm{TRUE}\}_\perp \oplus \{\mathrm{FALSE}\}_\perp \\
\mathrm{List} & = & \{\mathrm{NIL}\}_\perp \oplus \mathrm{Conscell} \\
\mathrm{Conscell} & = & (\mathrm{Val}_{\mathcal{E}} \times \mathrm{Val}_{\mathcal{E}})_\perp \\
\rho \in \mathrm{Bve}_{\mathcal{E}} & = & \mathrm{Bv} \to \mathrm{Val}_{\mathcal{E}} \\
\phi \in \mathrm{Fve}_{\mathcal{E}} & = & \mathrm{Fv} \to \mathrm{Val}_{\mathcal{E}}^{*} \to \mathrm{Val}_{\mathcal{E}}
\end{array}$$

Figure 2.3: Standard Semantic Domains

2.3 Standard Semantics

In this section the standard semantics of the language which is used throughout this thesis are defined. The standard semantic domains are shown in Figure 2.3. Expressible values in the language are atomic values or lists. Atomic values consist of integers and booleans. Integers are represented by the flat domain of integers, and booleans are represented by the values TRUE and FALSE. Empty lists are represented by the value NIL and non-empty lists are represented by pairs, where the first element of the pair represents the head of the list, and the second element of the pair represents the tail of the list.

The functionality of the standard semantic functions of the language is shown in Figure 2.4.

$$
\begin{array}{ll}
\mathcal{E}_p : & \text{Prog} \rightarrow \text{Val}_{\mathcal{E}} \\
\mathcal{E} : & \text{Exp} \rightarrow \text{Bve}_{\mathcal{E}} \rightarrow \text{Fve}_{\mathcal{E}} \rightarrow \text{Val}_{\mathcal{E}} \\
\mathcal{B} : & \text{Bas} \rightarrow \text{Val}_{\mathcal{E}}^{*} \rightarrow \text{Val}_{\mathcal{E}} \\
\mathcal{C} : & \text{Con} \rightarrow \text{Val}_{\mathcal{E}}^{*} \rightarrow \text{Val}_{\mathcal{E}} \\
\mathit{match}: & (\text{Val}_{\mathcal{E}} \times \text{Con}) \rightarrow \text{Bool}
\end{array}
$$

Figure 2.4: Standard Semantic Functions

$\mathcal{E}_p$ gives the meaning of a program, $\mathcal{E}$ gives the meaning of an expression, $\mathcal{B}$ gives the meaning of a basic function call and $\mathcal{C}$ gives the meaning of a constructor application. These functions are defined in Figure 2.5.

Empty environments are represented by $(\lambda x.\bot)$ in these functions, and non-empty environments are represented by $[x_1/v_1, \ldots, x_n/v_n]$ where the variable v_i is bound to the value x_i. The notation $\rho[x/v]$ represents an environment in which the variable v is bound to the value x, and variables other than v are bound to the value given in the environment ρ. For the sake of clarity, the domain injections and projections have been omitted from the semantics. These will be omitted from the semantics throughout the course of this thesis, unless there is an ambiguity.

The function *match* is an auxiliary function which is used to perform pattern matching within **case** expressions. This function is defined in Figure 2.6.

$$
\begin{array}{lcl}
\mathcal{E}_p[\![e & & \\
\quad \textbf{where} & & \\
\quad f_1\ v_{11} \ldots v_{1k_1} & = & e_1 \\
\quad \vdots & & \\
\quad f_n\ v_{n1} \ldots v_{nk_n} & = & e_n]\!] = \mathcal{E}[\![e]\!]\ (\lambda v.\bot)\ \phi_0
\end{array}
$$

where

$$\phi_0 = \text{fix}(\lambda\phi.[(\lambda x_1 \ldots \lambda x_{k_j}.\mathcal{E}[\![e_j]\!]\ [x_1/v_{j1}, \ldots, x_{k_j}/v_{jk_j}]\ \phi)/f_j])$$

$$
\begin{array}{lcl}
\mathcal{E}[\![k]\!]\ \rho\ \phi & = & k \\
\mathcal{E}[\![v]\!]\ \rho\ \phi & = & \rho[\![v]\!] \\
\mathcal{E}[\![b\ e_1 \ldots e_n]\!]\ \rho\ \phi & = & \mathcal{B}[\![b]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{E}[\![e_n]\!]\ \rho\ \phi) \\
\mathcal{E}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi & = & \mathcal{C}[\![c]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{E}[\![e_n]\!]\ \rho\ \phi) \\
\mathcal{E}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi & = & \phi[\![f]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{E}[\![e_n]\!]\ \rho\ \phi)
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{E}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]\ \rho\ \phi} \\
 & = & \mathcal{E}[\![e_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi \\
 & & \text{where} \\
 & & x \ = \ \mathcal{E}[\![e_0]\!]\ \rho\ \phi \\
 & & p_i \ = \ c\ v_1 \ldots v_n \text{ and } match(x, c)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{B}[\![+]\!] & = & \lambda x_1.\lambda x_2.x_1 + x_2 \\
\mathcal{B}[\![-]\!] & = & \lambda x_1.\lambda x_2.x_1 - x_2 \\
\mathcal{B}[\![<]\!] & = & \lambda x_1.\lambda x_2.x_1 < x_2 \\
\mathcal{B}[\![=]\!] & = & \lambda x_1.\lambda x_2.x_1 = x_2 \\
 & \vdots & \\
\mathcal{C}[\![\mathit{True}]\!] & = & \text{TRUE} \\
\mathcal{C}[\![\mathit{False}]\!] & = & \text{FALSE} \\
\mathcal{C}[\![\mathit{Nil}]\!] & = & \text{NIL} \\
\mathcal{C}[\![\mathit{Cons}]\!] & = & \lambda x_1.\lambda x_2.(x_1, x_2)
\end{array}
$$

Figure 2.5: Standard Semantics

$$
\begin{array}{rcl}
match(x,c) & = & (x = \text{TRUE and } c = \mathit{True}) \\
 & & \text{or } (x = \text{FALSE and } c = \mathit{False}) \\
 & & \text{or } (x = \text{NIL and } c = \mathit{Nil}) \\
 & & \text{or } (x \in \text{Conscell and } c = \mathit{Cons})
\end{array}
$$

Figure 2.6: Standard Semantics (auxiliary functions)

2.4 Store Semantics

In this section, non-standard semantics are presented which model the use of store in the language which is used throughout this thesis. These semantics are largely based on the store semantics for a higher order lazy language presented in (Hughes, 1991). They provide a reference against which store-related analyses and optimisations can be proved correct, so they model the use of store in possible implementations of the language. The store semantics may not model the use of store in possible implementations of the language particularly accurately, but they do provide a safe model[1].

The store semantic domains of the language are shown in Figure 2.7. Most of these domains are similar to the domains for the standard semantics of the language given in Figure 2.3, but some new domains have been added. Obviously, a domain of stores is required since the use of stores is being modelled. A store is represented by a function which returns the contents of a cell at a given location. Locations in the store are represented by integers. Unbound cells in the store are represented by the value UNB. Since the side-effect of updating a store is being modelled within the semantics, the current state of the store is threaded through the semantics. Values in the semantics are therefore represented by a pair, the first element of which is a location, and the second a store.

As in the standard semantics, expressible values in the language are atomic values or lists. Atomic values consist of integers and booleans. Integers are represented by the flat domain of integers, and booleans are represented by the values TRUE and FALSE. Empty lists are represented by the value NIL, and non-empty lists are represented by pairs of locations which give the head and the tail of the list respectively. Each expressible value in the semantics is allocated in the store. This is not necessary to ensure lazy evaluation, but is done to facilitate

[1]To determine whether the store semantics accurately model the use of store in implementations of the language, it would be necessary to compare them to a canonical operational semantics. This operational semantics would depend upon the evaluation mechanism of the language, which (for the sake of generality) has not been given here. It must therefore be ensured that the semantics model the use of store safely.

$$
\begin{array}{lcl}
\mathrm{Val}_{\mathcal{E}store} & = & (\mathrm{Loc} \times \mathrm{Store}_{\mathcal{E}store})_{\perp} \\
x \in \mathrm{Eval} & = & \mathrm{Atom} \oplus \mathrm{List} \\
\mathrm{Atom} & = & \mathrm{Int} \oplus \mathrm{Bool} \\
\mathrm{Int} & = & \{0\}_{\perp} \oplus \{1\}_{\perp} \oplus \{-1\}_{\perp} \oplus \ldots \\
\mathrm{Bool} & = & \{\mathrm{TRUE}\}_{\perp} \oplus \{\mathrm{FALSE}\}_{\perp} \\
\mathrm{List} & = & \{\mathrm{NIL}\}_{\perp} \oplus \mathrm{Conscell} \\
\mathrm{Conscell} & = & (\mathrm{Loc} \times \mathrm{Loc})_{\perp} \\
loc \in \mathrm{Loc} & = & \mathrm{Int} \\
\mathrm{Closure} & = & \mathrm{Store}_{\mathcal{E}store} \rightarrow \mathrm{Val}_{\mathcal{E}store} \\
\rho \in \mathrm{Bve}_{\mathcal{E}store} & = & \mathrm{Bv} \rightarrow \mathrm{Loc} \\
\phi \in \mathrm{Fve}_{\mathcal{E}store} & = & \mathrm{Fv} \rightarrow \mathrm{Loc}^{*} \rightarrow \mathrm{Store}_{\mathcal{E}store} \rightarrow \mathrm{Val}_{\mathcal{E}store} \\
\sigma \in \mathrm{Store}_{\mathcal{E}store} & = & \mathrm{Loc} \rightarrow (\mathrm{Closure} \oplus \mathrm{Loc} \oplus \mathrm{Eval} \oplus \{\mathrm{UNB}\}_{\perp})
\end{array}
$$

Figure 2.7: Store Semantic Domains

the extension of the store semantics to incorporate usage counting in the next chapter.

Within a lazy store semantics, it must be ensured that values are evaluated only when needed, and are not evaluated more than once. A new domain of *closures* is therefore introduced. These closures are used to delay the evaluation of expressions until they are actually required by the program. They are represented by functions which, when supplied with a store, will return the result of evaluating their associated expression in the given store.

Expressions are therefore evaluated only when their values are needed. The arguments of basic function applications and selectors of **case** expressions are evaluated to head normal form because they appear in a strict context. All other expressions are enclosed within closures to delay their evaluation until their values are required by the program.

To ensure that closures are not evaluated more than once, they are overwritten with the result of their evaluation immediately after they have been evaluated. Since the result of evaluating a closure is given by a location, cells in the store may contain the location

of another cell in the store, but there are no chains of indirection. Also, since it must be possible to overwrite the closures given by bound variables with the result of their evaluation, variables in the bound variable environment are bound to locations. These locations will either be bound to a closure, or to another location if the closure has been evaluated.

The functionality of the store semantic functions of the language is shown in Figure 2.8.

$$
\begin{array}{ll}
\mathcal{E}_p^{store}: & \text{Prog} \rightarrow \text{Val}_{\mathcal{E}^{store}} \\
\mathcal{E}^{store}: & \text{Exp} \rightarrow \text{Bve}_{\mathcal{E}^{store}} \rightarrow \text{Fve}_{\mathcal{E}^{store}} \rightarrow \text{Store}_{\mathcal{E}^{store}} \rightarrow \text{Val}_{\mathcal{E}^{store}} \\
\mathcal{B}_{\mathcal{E}^{store}}: & \text{Bas} \rightarrow \text{Loc}^* \rightarrow \text{Store}_{\mathcal{E}^{store}} \rightarrow \text{Val}_{\mathcal{E}^{store}} \\
\mathcal{C}_{\mathcal{E}^{store}}: & \text{Con} \rightarrow \text{Loc}^* \rightarrow \text{Store}_{\mathcal{E}^{store}} \rightarrow \text{Val}_{\mathcal{E}^{store}} \\
\mathit{alloc}: & ((\text{Closure} \oplus \text{Eval}) \times \text{Store}_{\mathcal{E}^{store}}) \rightarrow \text{Val}_{\mathcal{E}^{store}} \\
\mathit{force}: & \text{Val}_{\mathcal{E}^{store}} \rightarrow \text{Val}_{\mathcal{E}^{store}} \\
\mathit{match}: & (\text{Eval} \times \text{Con}) \rightarrow \text{Bool}
\end{array}
$$

Figure 2.8: Store Semantic Functions

$\mathcal{E}_p^{store}$ gives the meaning of a program and $\mathcal{E}^{store}$ gives the meaning of an expression. The location returned by $\mathcal{E}^{store}$ will be bound to an expressible value in the given store. $\mathcal{B}_{\mathcal{E}^{store}}$ gives the meaning of a basic function application and $\mathcal{C}_{\mathcal{E}^{store}}$ gives the meaning of a constructor application. These functions are defined in Figures 2.9 and 2.10.

The auxiliary functions of the store semantics are defined in Figure 2.11. The function *alloc* is used to allocate a given value at a location in the given store which was previously unbound. Both closures and expressible values can be allocated in this way. The function *force* is used to force the evaluation of the result of a program. It is possible that the result of a program contains closures. Any closures which are reachable from the result must therefore be evaluated. When *force* is applied to a closure, it causes the evaluation of the closure. The result of this evaluation is also forced. When it is applied to a list value it is recursively applied to the elements of the list, forcing their evaluation. All other values which can result from the evaluation of a program will have been fully evaluated already, and do not need to be forced. It is assumed that this function also serves to print out the result of the program. The function *match* is used to perform pattern matching within **case** expressions as before.

$$
\begin{array}{l}
\mathcal{E}_p^{store}[\![e \\
\qquad \textbf{where} \\
\qquad f_1\ v_{11}\ldots v_{1k_1} \quad = \quad e_1 \\
\qquad \vdots \\
\qquad f_n\ v_{n1}\ldots v_{nk_n} \quad = \quad e_n]\!] = \mathit{force}(\mathcal{E}^{store}[\![e]\!]\ (\lambda v.\bot)\ \phi_0\ (\lambda loc.\text{UNB})) \\
\text{where} \\
\phi_0 \quad = \quad \text{fix}\ (\lambda\phi.[(\lambda loc_1\ldots\lambda loc_{k_j}.\lambda\sigma.\mathcal{E}^{store}[\![e_j]\!]\ [loc_1/v_{j1},\ldots,loc_{k_j}/v_{jk_j}]\ \phi\ \sigma)/f_j])
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}^{store}[\![k]\!]\ \rho\ \phi\ \sigma & = & \mathit{alloc}(k,\sigma) \\
 & & \\
\mathcal{E}^{store}[\![v]\!]\ \rho\ \phi\ \sigma & = & (loc,\sigma'[loc/\rho[\![v]\!]]), \qquad\qquad \text{if}\ \ (\sigma\ (\rho[\![v]\!])) \in \text{Closure} \\
 & & \text{where} \\
 & & (loc,\sigma') \quad = \quad (\sigma\ (\rho[\![v]\!]))\ \sigma \\
 & & \\
 & = & ((\sigma\ (\rho[\![v]\!])),\sigma), \qquad\qquad \text{otherwise} \\
 & & \\
\mathcal{E}^{store}[\![b\ e_1\ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \mathcal{B}_{\mathcal{E}^{store}}[\![b]\!]\ loc_1\ldots loc_n\ \sigma_n \\
 & & \text{where} \\
 & & (loc_1,\sigma_1) \quad = \quad \mathcal{E}^{store}[\![e_1]\!]\ \rho\ \phi\ \sigma \\
 & & \qquad\qquad\quad \vdots \\
 & & (loc_n,\sigma_n) \quad = \quad \mathcal{E}^{store}[\![e_n]\!]\ \rho\ \phi\ \sigma_{n-1} \\
 & & \\
\mathcal{E}^{store}[\![c\ e_1\ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \mathcal{C}_{\mathcal{E}^{store}}[\![c]\!]\ loc_1\ldots loc_n\ \sigma_n \\
 & & \text{where} \\
 & & (loc_1,\sigma_1) \quad = \quad \mathit{alloc}((\mathcal{E}^{store}[\![e_1]\!]\ \rho\ \phi),\sigma) \\
 & & \qquad\qquad\quad \vdots \\
 & & (loc_n,\sigma_n) \quad = \quad \mathit{alloc}((\mathcal{E}^{store}[\![e_n]\!]\ \rho\ \phi),\sigma_{n-1}) \\
 & & \\
\mathcal{E}^{store}[\![f\ e_1\ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \phi[\![f]\!]\ loc_1\ldots loc_n\ \sigma_n \\
 & & \text{where} \\
 & & (loc_1,\sigma_1) \quad = \quad \mathit{alloc}((\mathcal{E}^{store}[\![e_1]\!]\ \rho\ \phi),\sigma) \\
 & & \qquad\qquad\quad \vdots \\
 & & (loc_n,\sigma_n) \quad = \quad \mathit{alloc}((\mathcal{E}^{store}[\![e_n]\!]\ \rho\ \phi),\sigma_{n-1})
\end{array}
$$

Figure 2.9: Store Semantics

$$
\begin{array}{lcl}
\mathcal{E}^{store}[\![\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]\ \rho\ \phi\ \sigma & & \\
 & = & \mathcal{E}^{store}[\![e_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi\ \sigma' \\
 & & \text{where} \\
 & & \begin{array}{lcl} (loc, \sigma') & = & \mathcal{E}^{store}[\![e_0]\!]\ \rho\ \phi\ \sigma \\ x & = & \sigma'\ loc \\ p_i & = & c\ v_1 \ldots v_n \text{ and } match(x, c) \end{array} \\
\\
\mathcal{B}_{\mathcal{E}^{store}}[\![+]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((x_1 + x_2), \sigma) \\
 & & \text{where} \\
 & & \begin{array}{lcl} x_1 & = & \sigma\ loc_1 \\ x_2 & = & \sigma\ loc_2 \end{array} \\
\\
\mathcal{B}_{\mathcal{E}^{store}}[\![-]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((x_1 - x_2), \sigma) \\
 & & \text{where} \\
 & & \begin{array}{lcl} x_1 & = & \sigma\ loc_1 \\ x_2 & = & \sigma\ loc_2 \end{array} \\
\\
\mathcal{B}_{\mathcal{E}^{store}}[\![<]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((x_1 < x_2), \sigma) \\
 & & \text{where} \\
 & & \begin{array}{lcl} x_1 & = & \sigma\ loc_1 \\ x_2 & = & \sigma\ loc_2 \end{array} \\
\\
\mathcal{B}_{\mathcal{E}^{store}}[\![=]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((x_1 = x_2), \sigma) \\
 & & \text{where} \\
 & & \begin{array}{lcl} x_1 & = & \sigma\ loc_1 \\ x_2 & = & \sigma\ loc_2 \end{array} \\
 & \vdots & \\
\mathcal{C}_{\mathcal{E}^{store}}[\![True]\!] & = & \lambda\sigma.alloc(\text{TRUE},\sigma) \\
\\
\mathcal{C}_{\mathcal{E}^{store}}[\![False]\!] & = & \lambda\sigma.alloc(\text{FALSE},\sigma) \\
\\
\mathcal{C}_{\mathcal{E}^{store}}[\![Nil]\!] & = & \lambda\sigma.alloc(\text{NIL},\sigma) \\
\\
\mathcal{C}_{\mathcal{E}^{store}}[\![Cons]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((loc_1, loc_2),\sigma)
\end{array}
$$

Figure 2.10: Store Semantics (continued)

$$
\begin{array}{lcl}
alloc(v,\sigma) & = & (loc, \sigma[v/loc]) \\
 & & \text{where} \\
 & & \sigma\ loc \quad = \quad \text{UNB} \\
 & & \\
force(loc,\sigma) & = & (loc',\sigma'[loc'/loc]), \qquad\qquad \text{if } (\sigma\ loc) \in \text{Closure} \\
 & & \text{where} \\
 & & (loc', \sigma') \quad = \quad force((\sigma\ loc)\ \sigma) \\
 & & \\
 & = & force((\sigma\ loc),\sigma), \qquad\qquad \text{if } (\sigma\ loc) \in \text{Loc} \\
 & & \\
 & = & (loc,\sigma_2[(loc_1,loc_2)/loc]), \quad \text{if } (\sigma\ loc) \in \text{Conscell} \\
 & & \text{where} \\
 & & (loc_1, \sigma_1) \quad = \quad force((\sigma\ loc) \downarrow 1, \sigma) \\
 & & (loc_2, \sigma_2) \quad = \quad force((\sigma\ loc) \downarrow 2, \sigma_1) \\
 & & \\
 & = & (loc,\sigma), \qquad\qquad\qquad \text{otherwise} \\
 & & \\
match(x,c) & = & (x = \text{TRUE and } c = True) \\
 & & \text{or } (x = \text{FALSE and } c = False) \\
 & & \text{or } (x = \text{NIL and } c = Nil) \\
 & & \text{or } (x \in \text{Conscell and } c = Cons)
\end{array}
$$

Figure 2.11: Store Semantics (auxiliary functions)

2.5 Congruence

Since the store semantics of the language will be used as a reference against which store-related analyses and optimisations can be proved correct, the store semantics and standard semantics of the language must be shown to be congruent. A function Φ is therefore defined which is used to extract the standard semantic component from a store value. The store semantics and standard semantics of the language can then be shown to be congruent if the result of evaluating any program in both semantics have the same standard semantic component.

Definition 2.5.1 (Standard Semantic Component of a Store Value) *The standard semantic component of a store value can be extracted using the function Φ which is defined as follows:*

$$\begin{array}{llll}
\Phi\colon \; Val_{\mathcal{E}^{store}} \rightarrow Val_{\mathcal{E}} & & & \\
\Phi(loc, \sigma) & = & \bot, & \text{if } (\sigma\ loc) = \text{UNB} \\
 & = & \Phi((\sigma\ loc)\ \sigma), & \text{if } (\sigma\ loc) \in \text{Closure} \\
 & = & \Phi((\sigma\ loc), \sigma), & \text{if } (\sigma\ loc) \in \text{Loc} \\
 & = & (\Phi((\sigma\ loc) \downarrow 1, \sigma), \Phi((\sigma\ loc) \downarrow 2, \sigma)), & \text{if } (\sigma\ loc) \in \text{Conscell} \\
 & = & \sigma\ loc, & \text{otherwise}
\end{array}$$

□

This function forces the evaluation of any closures in the store value, and extracts the standard semantic component from the resulting store value. Using this definition, the congruence of expressions in the store semantics and standard semantics of the language can be shown by proving the following lemma.

Lemma 2.5.2 (Congruence of Expressions)

for all $\rho_{\mathcal{E}^{store}} \in \text{Bve}_{\mathcal{E}^{store}}$, $\phi_{\mathcal{E}^{store}} \in \text{Fve}_{\mathcal{E}^{store}}$, $\sigma_{\mathcal{E}^{store}} \in \text{Store}_{\mathcal{E}^{store}}$, $\phi_{\mathcal{E}} \in \text{Fve}_{\mathcal{E}}$, $e \in \text{Exp}$:

if for all $f \in \text{dom}(\phi_{\mathcal{E}^{store}})$:

$\Phi(\phi_{\mathcal{E}^{store}}[\![f]\!]\ loc_1 \dots loc_n\ \sigma_{\mathcal{E}^{store}}) = \phi_{\mathcal{E}}[\![f]\!]\ (\Phi(loc_1, \sigma_{\mathcal{E}^{store}})) \dots (\Phi(loc_n, \sigma_{\mathcal{E}^{store}}))$

then for all $v \in \text{dom}(\rho_{\mathcal{E}^{store}})$:

$\Phi(\mathcal{E}^{store}[\![e]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) = \mathcal{E}[\![e]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}$

□

Proof

The proof of this lemma can be found in Appendix A.1.

□

The following lemma states that the functional variable environments in the store semantics and standard semantics of the language will always satisfy the requirement in Lemma 2.5.2.

Lemma 2.5.3 (Congruence of Functional Variable Environments)

for all $p \in$ Prog:

if $\mathcal{E}_p[\![p]\!] = \mathcal{E}[\![e]\!]\ (\lambda v.\bot)\ \phi_{\mathcal{E}}$

and $\mathcal{E}_p^{store}[\![p]\!] = force(\mathcal{E}^{store}[\![e]\!]\ (\lambda v.\bot)\ \phi_{\mathcal{E}^{store}}\ (\lambda loc.\mathrm{UNB}))$

then for all $f \in \mathrm{dom}(\phi_{\mathcal{E}^{store}})$, $\sigma_{\mathcal{E}^{store}} \in \mathrm{Store}_{\mathcal{E}^{store}}$:

$\Phi(\phi_{\mathcal{E}^{store}}[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_{\mathcal{E}^{store}}) = \phi_{\mathcal{E}}[\![f]\!]\ (\Phi(loc_1, \sigma_{\mathcal{E}^{store}})) \ldots (\Phi(loc_n, \sigma_{\mathcal{E}^{store}}))$

□

Proof

The proof of this lemma can be found in Appendix A.2.

□

The congruence of programs in the store semantics and standard semantics of the language can now be shown by proving the following theorem.

Theorem 2.5.4 (Congruence of Programs)

for all $p \in$ Prog: $\Phi(\mathcal{E}_p^{store}[\![p]\!]) = \mathcal{E}_p[\![p]\!]$

□

Proof

This theorem follows immediately from Lemmata 2.5.2 and 2.5.3.

□

2.6 Related Work

A large number of store semantics have been defined for strict languages. Examples of first order strict store semantics can be found in (Mycroft, 1981; Hudak, 1987; Jones & Le Métayer, 1989; Jensen, 1990). These semantics are similar to the store semantics presented in this

chapter since they thread the current state of the store through the semantics, but they are simpler because there is no need to deal with the closures which are required in a lazy store semantics. Examples of higher order strict store semantics can be found in (Pleban, 1990; Andersen, 1990; Deutsch, 1990; Hughes, 1991). The store semantics in (Pleban, 1990) are defined using a relatively complex continuation semantics. The store semantics in (Andersen, 1990) and (Deutsch, 1990) are defined using an operational semantics. The strict store semantics described in (Hughes, 1991) are quite similar to the semantics in (Mycroft, 1981; Jones & Le Métayer, 1989; Jensen, 1990), except that higher order values can be allocated in the store.

Examples of store semantics for lazy languages can be found in (Josephs, 1987) and (Hughes, 1991). The store semantics in (Josephs, 1987) are defined using a continuation semantics. The lazy store semantics in (Hughes, 1991) are similar to the lazy store semantics defined in this chapter, except that higher order values can be allocated in the store. Also, not all expressible values are allocated in the store in the semantics described in (Hughes, 1991), since this is not necessary to ensure lazy evaluation. This was done in the store semantics described in this chapter to facilitate their extension to incorporate usage counting in the next chapter. Of all the described store semantics, congruence with the standard semantics of the described language is considered only in (Pleban, 1990) and (Hughes, 1991).

2.7 Conclusion

In this chapter, the syntax and standard semantics of the language which will be used throughout this thesis have been defined. Non-standard store semantics which model the use of store in possible implementations of the language were then defined. These store semantics provide a reference against which store-related analyses and optimisations can be proved correct. To ensure that these store semantics model the use of store safely, they were proved to be congruent to the standard semantics of the language.

Now that the store semantics of the language have been defined, store related analyses and optimisations can be defined and proved correct with respect to them. In Chapter 3, an analysis is presented which can be used to detect which store cells will become garbage, and is proved to be correct with respect to these store semantics. In Chapter 4, it is shown how the information obtained by this analysis can be used to validate compile-time garbage collection, and in Chapter 5 it is shown how the information obtained by the analysis can be used to guide the transformation when compile-time garbage avoidance is performed.

Chapter 3

Compile-Time Garbage Detection

In this chapter, it is shown how the cells which will become garbage within a program can be detected at compile-time. A cell will become garbage during the evaluation of an expression if it is unshared when it loses a reference. To determine whether a cell is unshared, the number of times that the cell is used is determined. If the cell is used only once, then it is unshared.

To determine the number of times a cell is used, the store semantics presented in Section 2.4 are augmented to incorporate usage counting. These usage counting store semantics must be abstracted in some way to allow usage counts to be determined at compile-time. Usage counting store values are therefore abstracted to *usage patterns*. These patterns are finite objects which indicate the number of times each part of a value is used. A usage counting analysis is then defined, using these patterns, to determine at compile-time the number of times each part of a value will be used in future computations. The usage count obtained by this analysis must be safe with respect to the actual usage counting store value. This will be the case if the usage count of a value determined by the analysis is not less than the actual usage count, so it will not be assumed that a cell will become garbage when it is still required by a program. The described usage counting analysis is proved to be safe with respect to the usage counting store semantics, and some examples of its application are given.

The remainder of this chapter is structured as follows:

- **Section 3.1**: the store semantics presented in Section 2.4 are augmented to incorporate usage counting.
- **Section 3.2**: domains of usage patterns which are abstractions of usage counting store values are defined.
- **Section 3.3**: the operations which can be performed upon usage patterns are defined.
- **Section 3.4**: a usage counting analysis is defined over the domains of usage patterns.
- **Section 3.5**: the usage counting analysis is proved to be correct with respect to the usage counting store semantics.
- **Section 3.6**: some examples of the application of usage counting analysis are given.
- **Section 3.7**: related work is considered.
- **Section 3.8**: a summary of this chapter is given.

3.1 Usage Counting Store Semantics

To provide a reference against which the compile-time analysis of store usage can be proved correct, the store semantics presented in the previous chapter are augmented to incorporate usage counting. This involves counting the number of times each value is used in a program.

The usage counting store semantic domains are shown in Figure 3.1. These domains are very similar to the domains for the store semantics of the language given in Figure 2.7. As before, all expressible values are allocated in the store so that a usage count can be associated with them. A new domain is defined to associate a usage count with each expressible value. These usage counts are represented by integers. The functionality of the usage counting store semantic functions of the language is shown in Figure 3.2, and they are defined in Figures 3.3 and 3.4. They are very similar to the functions defined for the store semantics of the language given in Figures 2.9 and 2.10, except that they maintain a usage count for all values in the language. All new values which are created within a program are given an initial usage count of 0 since they have not yet been used. These usage counts are incremented only when their associated values are used. This will be the case if a value appears in a strict context. The usage count for a value is therefore incremented only if it is an argument in a basic function

$$
\begin{array}{lcl}
\text{Val}_{\mathcal{E}^{use}} & = & (\text{Loc} \times \text{Store}_{\mathcal{E}^{use}})_{\perp} \\
x \in \text{Eval} & = & \text{Atom} \oplus \text{List} \\
\text{Atom} & = & \text{Int} \oplus \text{Bool} \\
\text{Int} & = & \{0\}_{\perp} \oplus \{1\}_{\perp} \oplus \{-1\}_{\perp} \oplus \ldots \\
\text{Bool} & = & \{\text{TRUE}\}_{\perp} \oplus \{\text{FALSE}\}_{\perp} \\
\text{List} & = & \{\text{NIL}\}_{\perp} \oplus \text{Conscell} \\
\text{Conscell} & = & (\text{Loc} \times \text{Loc})_{\perp} \\
loc \in \text{Loc} & = & \text{Int} \\
\text{Uval} & = & (\text{Use} \times \text{Eval})_{\perp} \\
u \in \text{Use} & = & \text{Int} \\
\text{Closure} & = & \text{Store}_{\mathcal{E}^{use}} \rightarrow \text{Val}_{\mathcal{E}^{use}} \\
\rho \in \text{Bve}_{\mathcal{E}^{use}} & = & \text{Bv} \rightarrow \text{Loc} \\
\phi \in \text{Fve}_{\mathcal{E}^{use}} & = & \text{Fv} \rightarrow \text{Loc}^{*} \rightarrow \text{Store}_{\mathcal{E}^{use}} \rightarrow \text{Val}_{\mathcal{E}^{use}} \\
\sigma \in \text{Store}_{\mathcal{E}^{use}} & = & \text{Loc} \rightarrow (\text{Closure} \oplus \text{Loc} \oplus \text{Uval} \oplus \{\text{UNB}\}_{\perp})
\end{array}
$$

Figure 3.1: Usage Counting Store Semantic Domains

call, a selector in a **case** expression, or its value is being forced as the result of a program. Usage counts can only increase as they are never decremented.

The auxiliary functions of the usage counting store semantics are defined in Figure 3.5. These functions are very similar to the auxiliary functions of the store semantics given in Figure 2.11, except that the function *inc* has been defined to increment the usage count associated with an expressible value.

Since the usage counting store semantics of the language will be used as a reference against which store-related analyses and optimisations can be proved correct, they must also be shown to be congruent to the standard semantics. This can be done in a similar manner to that described in Section 2.5 for the store semantics.

$$
\begin{array}{ll}
\mathcal{E}_p^{use}: & \text{Prog} \rightarrow \text{Val}_{\mathcal{E}^{use}} \\
\mathcal{E}^{use}: & \text{Exp} \rightarrow \text{Bve}_{\mathcal{E}^{use}} \rightarrow \text{Fve}_{\mathcal{E}^{use}} \rightarrow \text{Store}_{\mathcal{E}^{use}} \rightarrow \text{Val}_{\mathcal{E}^{use}} \\
\mathcal{B}^{use}: & \text{Bas} \rightarrow \text{Loc}^* \rightarrow \text{Store}_{\mathcal{E}^{use}} \rightarrow \text{Val}_{\mathcal{E}^{use}} \\
\mathcal{C}^{use}: & \text{Con} \rightarrow \text{Loc}^* \rightarrow \text{Store}_{\mathcal{E}^{use}} \rightarrow \text{Val}_{\mathcal{E}^{use}} \\
alloc: & ((\text{Closure} \oplus \text{Uval}) \times \text{Store}_{\mathcal{E}^{use}}) \rightarrow \text{Val}_{\mathcal{E}^{use}} \\
inc: & \text{Val}_{\mathcal{E}^{use}} \rightarrow \text{Val}_{\mathcal{E}^{use}} \\
force: & \text{Val}_{\mathcal{E}^{use}} \rightarrow \text{Val}_{\mathcal{E}^{use}} \\
match: & (\text{Eval} \times \text{Con}) \rightarrow \text{Bool}
\end{array}
$$

Figure 3.2: Usage Counting Store Semantic Functions

3.2 Usage Patterns

The usage counting store semantics defined in the previous section must be abstracted in some way to allow usage counts to be determined at compile-time. One approach would be to use an abstract store, as is done in (Hudak, 1987; Andersen, 1990; Deutsch, 1990). Abstract stores tend to be relatively large objects, so such an analysis is likely to be inefficient. The approach which is taken here is to abstract usage counting store values to usage patterns which represent the number of times each part of a value is used in future computations. The usage pattern which gives the future usage of a value is called its *context*.

The notation D_{ABS} used in the definition of the usage counting domains represents the lifting of the domain D to add a new bottom element ABS. This lifting operation is defined as follows.

Definition 3.2.1 (The Domain Lifting Operation)

$$
\begin{array}{rcllll}
D_{ABS} & = & D \cup \{\text{ABS}\} & & & \\
 & & \text{where} & & & \\
 & & \text{ABS} & \sqsubseteq_{D_{ABS}} & d, & \forall\, d \in D_{ABS} \\
 & & d_1 & \sqsubseteq_{D_{ABS}} & d_2, & \forall\, d_1, d_2 \in D \text{ s.t. } d_1 \sqsubseteq_D d_2
\end{array}
$$

□

$$\mathcal{E}_p^{use}[\![e \\ \quad \textbf{where} \\ \quad f_1\ v_{11} \ldots v_{1k_1} \;=\; e_1 \\ \quad \vdots \\ \quad f_n\ v_{n1} \ldots v_{nk_n} \;=\; e_n]\!] = force(\mathcal{E}^{use}[\![e]\!]\ (\lambda v.\bot)\ \phi_0\ (\lambda loc.\mathrm{UNB}))$$

where

$$\phi_0 \;=\; \text{fix}\ (\lambda\phi.[(\lambda loc_1 \ldots \lambda loc_{k_j}.\lambda\sigma.\mathcal{E}^{use}[\![e_j]\!]\ [loc_1/v_{j1},\ldots,loc_{k_j}/v_{jk_j}]\ \phi\ \sigma)/f_j])$$

$$\mathcal{E}^{use}[\![k]\!]\ \rho\ \phi\ \sigma \;=\; alloc((0,k),\sigma)$$

$$\begin{aligned} \mathcal{E}^{use}[\![v]\!]\ \rho\ \phi\ \sigma \;=\;& (loc,\sigma'[loc/\rho[\![v]\!]]), \quad \text{if } (\sigma\ (\rho[\![v]\!])) \in \text{Closure} \\ & \text{where} \\ & (loc,\sigma') \;=\; (\sigma\ (\rho[\![v]\!]))\ \sigma \\ =\;& ((\sigma\ (\rho[\![v]\!])),\sigma), \quad \text{otherwise} \end{aligned}$$

$$\begin{aligned} \mathcal{E}^{use}[\![b\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma \;=\;& \mathcal{B}^{use}[\![b]\!]\ loc_1 \ldots loc_n\ \sigma_n \\ & \text{where} \\ & (loc_1,\sigma_1) \;=\; inc(\mathcal{E}^{use}[\![e_1]\!]\ \rho\ \phi\ \sigma) \\ & \qquad \vdots \\ & (loc_n,\sigma_n) \;=\; inc(\mathcal{E}^{use}[\![e_n]\!]\ \rho\ \phi\ \sigma_{n-1}) \end{aligned}$$

$$\begin{aligned} \mathcal{E}^{use}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma \;=\;& \mathcal{C}^{use}[\![c]\!]\ loc_1 \ldots loc_n\ \sigma_n \\ & \text{where} \\ & (loc_1,\sigma_1) \;=\; alloc((\mathcal{E}^{use}[\![e_1]\!]\ \rho\ \phi),\sigma) \\ & \qquad \vdots \\ & (loc_n,\sigma_n) \;=\; alloc((\mathcal{E}^{use}[\![e_n]\!]\ \rho\ \phi),\sigma_{n-1}) \end{aligned}$$

$$\begin{aligned} \mathcal{E}^{use}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma \;=\;& \phi[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_n \\ & \text{where} \\ & (loc_1,\sigma_1) \;=\; alloc((\mathcal{E}^{use}[\![e_1]\!]\ \rho\ \phi),\sigma) \\ & \qquad \vdots \\ & (loc_n,\sigma_n) \;=\; alloc((\mathcal{E}^{use}[\![e_n]\!]\ \rho\ \phi),\sigma_{n-1}) \end{aligned}$$

Figure 3.3: Usage Counting Store Semantics

$$
\begin{array}{lcl}
\mathcal{E}^{use}[\![\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k]\!] \ \rho \ \phi \ \sigma & & \\
 & = & \mathcal{E}^{use}[\![e_i]\!] \ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n] \ \phi \ \sigma' \\
 & & \text{where} \\
 & & \begin{array}{lcl} (loc, \sigma') & = & inc(\mathcal{E}^{use}[\![e_0]\!] \ \rho \ \phi \ \sigma) \\ (u, x) & = & \sigma' \ loc \\ p_i & = & c \ v_1 \ldots v_n \text{ and } match(x, c) \end{array}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{B}^{use}[\![+]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 + x_2)), \sigma) \\
 & & \text{where} \\
 & & \begin{array}{lcl} (u_1, x_1) & = & \sigma \ loc_1 \\ (u_2, x_2) & = & \sigma \ loc_2 \end{array}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{B}^{use}[\![-]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 - x_2)), \sigma) \\
 & & \text{where} \\
 & & \begin{array}{lcl} (u_1, x_1) & = & \sigma \ loc_1 \\ (u_2, x_2) & = & \sigma \ loc_2 \end{array}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{B}^{use}[\![<]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 < x_2)), \sigma) \\
 & & \text{where} \\
 & & \begin{array}{lcl} (u_1, x_1) & = & \sigma \ loc_1 \\ (u_2, x_2) & = & \sigma \ loc_2 \end{array}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{B}^{use}[\![=]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 = x_2)), \sigma) \\
 & & \text{where} \\
 & & \begin{array}{lcl} (u_1, x_1) & = & \sigma \ loc_1 \\ (u_2, x_2) & = & \sigma \ loc_2 \end{array} \\
 & \vdots &
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{C}^{use}[\![True]\!] & = & \lambda\sigma.alloc((0,\text{TRUE}),\sigma) \\
\mathcal{C}^{use}[\![False]\!] & = & \lambda\sigma.alloc((0,\text{FALSE}),\sigma) \\
\mathcal{C}^{use}[\![Nil]\!] & = & \lambda\sigma.alloc((0,\text{NIL}),\sigma) \\
\mathcal{C}^{use}[\![Cons]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0,(loc_1, loc_2)),\sigma)
\end{array}
$$

Figure 3.4: Usage Counting Store Semantics (continued)

$$
\begin{array}{lcll}
\mathit{alloc}(v,\sigma) & = & (loc, \sigma[v/loc]) & \\
 & & \text{where} & \\
 & & \sigma\ loc \ = \ \text{UNB} & \\
 & & & \\
\mathit{inc}(loc,\sigma) & = & (loc, \sigma[(u+1,x)/loc]) & \\
 & & \text{where} & \\
 & & (u,x) \ = \ \sigma\ loc & \\
 & & & \\
\mathit{force}(loc,\sigma) & = & (loc,\sigma), & \text{if } (\sigma\ loc) = \bot \\
 & & & \text{or } (\sigma\ loc) = \text{UNB} \\
 & = & (loc',\sigma'[loc'/loc]), & \text{if } (\sigma\ loc) \in \text{Closure} \\
 & & \text{where} & \\
 & & (loc',\sigma') \ = \ \mathit{force}((\sigma\ loc)\ \sigma) & \\
 & = & \mathit{force}((\sigma\ loc),\sigma), & \text{if } (\sigma\ loc) \in \text{Loc} \\
 & = & \mathit{inc}(loc,\sigma_2[(u,(loc_1,loc_2))/loc]), & \text{if } (\sigma\ loc) \in \text{Uval} \\
 & & \text{where} & \text{and } x \in \text{Conscell} \\
 & & (u,x) \ = \ \sigma\ loc & \\
 & & (loc_1,\sigma_1) \ = \ \mathit{force}(x\downarrow 1,\sigma) & \\
 & & (loc_2,\sigma_2) \ = \ \mathit{force}(x\downarrow 2,\sigma_1) & \\
 & = & \mathit{inc}(loc,\sigma), & \text{otherwise} \\
 & & & \\
\mathit{match}(x,c) & = & (x = \text{TRUE and } c = \mathit{True}) & \\
 & & \text{or } (x = \text{FALSE and } c = \mathit{False}) & \\
 & & \text{or } (x = \text{NIL and } c = \mathit{Nil}) & \\
 & & \text{or } (x \in \text{Conscell and } c = \mathit{Cons}) &
\end{array}
$$

Figure 3.5: Usage Counting Store Semantics (auxiliary functions)

A different domain of usage patterns is defined for each possible type of value in the language. The domain of usage patterns for a value of type T is given by $U(T)$. The type T_A in the definition of the domain $U(T_A)$ represents an atomic type (*int*, *bool*). These domains are defined as follows.

Definition 3.2.2 (Domains of Usage Patterns)

$$U(T_A) = (U'(T_A))_{ABS}$$

$$U'(T_A) = \{0,1,2\}$$
where
$$0 \sqsubseteq_{U'(T_A)} u, \quad \forall\, u \in U'(T_A)$$
$$u \sqsubseteq_{U'(T_A)} 2, \quad \forall\, u \in U'(T_A)$$

$$U(list\ T) = (U'(list\ T))_{ABS}$$

$$U'(list\ T) = (U'(T_A) \times U(T))$$
where
$$(u_1, u_2) \sqsubseteq_{U'(list\ T)} (u_1', u_2'), \text{ if } u_1 \sqsubseteq_{U'(T_A)} u_1' \text{ and } u_2 \sqsubseteq_{U(T)} u_2'$$

□

Each domain $U(T)$ is an abstract context domain as defined in (Hughes, 1988) with the least element ABS representing absence (indicating that an expression is not evaluated). There is no element in any of the domains $U(T)$ representing contradiction because it is assumed that all programs are well typed, and contradiction can never arise.

Elements of the domain $U(T_A)$ describe the usage of values of atomic type. The elements in this domain, other than ABS, are the usage patterns 0, 1 and 2 which indicate that a value is not used, is used at most once, or may be used any number of times respectively.

Elements of the domain $U(list\ T)$ describe the usage of list values containing elements of type T. Elements of this domain, other than ABS, are pairs, where the first element of the pair describes the usage of all the spine cells in the list, and the second element describes the usage of all the elements in the list. Since these elements describe the usage of more than one value, they give a safe approximation to the usage of all of them. The usage of the spine cells of a list are represented by the values 0, 1 and 2. The value 0 indicates that none of the spine cells are used at all. The value 1 indicates that none of the spine cells are used more than once, and the value 2 indicates that all the spine cells may be used any number of times. The usage of the elements in the list are described by the usage domain corresponding to their type. Some elements of the domain $U(list\ T)$ describe a list in which the spine cells are not

used, but the list elements are used. Although this situation cannot occur, these elements are included to simplify the definition of the domain.

Each domain $U(T)$ is a complete lattice, with the least element representing absence, and the greatest element representing a value in which all parts may be used any number of times. The usage pattern ABS indicates that an expression is not evaluated, so no parts of it are used. Usage patterns other than ABS indicating that no parts of a value are used (for example, the usage pattern (0,0) in the domain $U(list\ T_A)$) represent a context in which an expression is evaluated to normal form, but is not used in any further computations. Although this situation should not occur in a lazy functional language, it is shown in Section 5.2.1 how these usage patterns can be used to detect transient structures within expressions.

The domain $U(list\ T_A)$ can be viewed as shown in Figure 3.6.

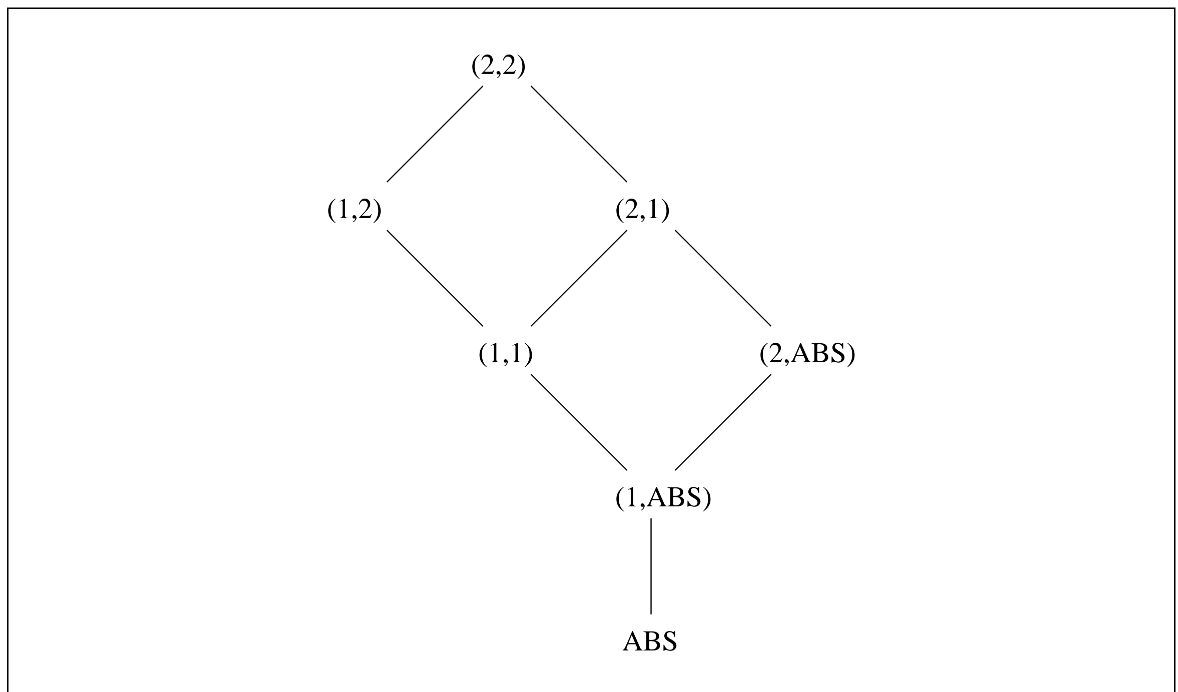

Figure 3.6: The Domain of Usage Patterns $U(list\ T_A)$

In general, if the definition of type T is parameterised by the types $T_1 \ldots T_n$, then the usage domain for a value of type T is given by $U(T)$, which is defined as follows:

Definition 3.2.3 (General Definition of Domains of Usage Patterns)

$$U(T) = (U'(T))_{ABS}$$

$$\begin{array}{rcl} U'(T) & = & (U'(T_A) \times U(T_1) \times \ldots \times U(T_n)) \\ & & \text{where} \\ & & (u_0, \ldots, u_n) \sqsubseteq_{U'(T)} (u'_0, \ldots, u'_n), \quad \begin{array}{llll} \text{if} & u_0 & \sqsubseteq_{U'(T_A)} & u'_0 \\ \text{and} & u_1 & \sqsubseteq_{U(T_1)} & u'_1 \\ \vdots & & & \\ \text{and} & u_n & \sqsubseteq_{U(T_n)} & u'_n \end{array} \end{array}$$

□

3.3 Operations on Usage Patterns

In this section, the operations which can be performed upon usage patterns are defined.

When the usage of a value in one expression is given by u_1, and the usage of the same value in another expression is given by u_2, a means of combining these two usage patterns into one describing the total usage of the value in both expressions is required. As in (Hughes, 1988), a binary operator & is defined to provide this information. This operator can be regarded as an abstract addition operator over elements in each domain $U(T)$. It is defined on the domain of usage patterns for values of atomic type as follows.

Definition 3.3.1 (The & Operator)

$$\begin{array}{lllll} u \ \& \ \text{ABS} & = & u, & \forall\, u \in U(T_A) \\ u \ \& \ 0 & = & 0, & \text{if } u = \text{ABS} \\ & = & u, & \text{otherwise} \\ u \ \& \ 1 & = & 1, & \text{if } u = \text{ABS or } u = 0 \\ & = & 2, & \text{otherwise} \\ u \ \& \ 2 & = & 2, & \forall\, u \in U(T_A) \end{array}$$

□

The definition of this operator is extended pointwise on domains of usage patterns for values of structured type.

The two usage patterns which are combined using this operator will be safe approximations to the usage of a value in two different expressions. The usage pattern which is produced as the result of this operator will therefore be a safe approximation to the total usage of the value in both expressions, since it simply acts as an abstract addition operator over domains of usage patterns.

Also following (Hughes, 1988), the binary operator $\rightarrow$ is defined to preserve absence in the context ABS. It is defined for each domain of usage patterns as follows.

Definition 3.3.2 (The $\rightarrow$ Operator)

$$\begin{array}{lllll} u_1 \rightarrow u_2 & = & \text{ABS}, & \text{if } u_1 = \text{ABS} \\ & = & u_2, & \text{otherwise} \end{array}$$

□

If an expression appears in the context ABS, then no part of the result of the expression will be used, and so no part of the sub-expressions occurring within it will be used either. It must therefore be ensured that any absence in the context of an expression is propagated to all sub-expressions.

The binary operator $\sqcup$ gives the least upper bound of two usage patterns in each domain of usage patterns.

To determine the usage of a constructor application from the usage of its arguments, abstract constructors which operate on usage patterns are defined. Corresponding to each constructor c of type $T_1 \rightarrow \ldots \rightarrow T_n$, abstract constructors $\mathcal{U}c$ which are of type $U'(T_A) \rightarrow U(T_1) \rightarrow \ldots \rightarrow U(T_n)$ are defined as follows.

Definition 3.3.3 (The Abstract Constructors $\mathcal{U}c$)

$$\begin{array}{lll} \mathcal{U}\mathit{False}(u_0) & = & u_0 \\ \mathcal{U}\mathit{True}(u_0) & = & u_0 \\ \mathcal{U}\mathit{Nil}(u_0) & = & (u_0,\text{ABS}) \\ \mathcal{U}\mathit{Cons}(u_0,u_1,u_2) & = & (u_0,u_1) \sqcup u_2 \end{array}$$

□

The additional argument for each abstract constructor is an element of the usage domain $U'(T_A)$. It represents the usage of the overall resulting structure if it is of atomic type, or the usage of the root cell of the resulting structure if it is of list type. The usage of the spine cells in a list is given by the least upper bound of the usage of the root cell of the list, and the usage of the spine cells in the tail of the list. The usage of the elements in a list is given by the least upper bound of the usage of the head of the list and the usage of the elements in the tail of the list.

In general, if a constructor c is of type $T_1 \to \ldots \to T_n$, and $U'(T_n) = (U'(T_A) \times U(T'_1) \times \ldots \times U(T'_k))$, then the abstract constructor $\mathcal{U}c$ is defined as follows:

Definition 3.3.4 (General Definition of the Abstract Constructors $\mathcal{U}c$)

$$
\begin{array}{lcl}
\mathcal{U}c & : & U'(T_A) \to U(T_1) \to \ldots \to U(T_n) \\
\mathcal{U}c(u_0, \ldots, u_n) & = & u' \sqcup (u_0, u'_1, \ldots, u'_k) \\
 & & \text{where} \\
 & & \quad u' \;=\; \bigsqcup_{i=1}^{n} \{u_i | u_i \in U(T_n)\} \\
 & & \quad u'_1 \;=\; \bigsqcup_{i=1}^{n} \{u_i | u_i \in U(T'_1)\} \\
 & & \quad \vdots \\
 & & \quad u'_k \;=\; \bigsqcup_{i=1}^{n} \{u_i | u_i \in U(T'_k)\}
\end{array}
$$

□

The usage of the head and tail of a list can be determined from the usage of the overall list using the $\mathcal{U}Cons\#1$ and $\mathcal{U}Cons\#2$ operators respectively. These operators are defined as follows:

Definition 3.3.5 (The $\mathcal{U}Cons\#1$ and $\mathcal{U}Cons\#2$ Operators)

$$
\begin{array}{lcl}
\mathcal{U}Cons\#1 \text{ ABS} & = & \text{ABS} \\
\mathcal{U}Cons\#1 \ (u_1, u_2) & = & u_2 \\
 & & \\
\mathcal{U}Cons\#2 \text{ ABS} & = & \text{ABS} \\
\mathcal{U}Cons\#2 \ (u_1, u_2) & = & (u_1, u_2)
\end{array}
$$

□

In general, if a constructor c is of type $T_1 \rightarrow \ldots \rightarrow T_n$, and $U'(T_n) = (U'(T_A) \times U(T'_1) \times \ldots \times U(T'_k))$, then the operators $\mathcal{U}c\#i$ where $1 \leq i < n$ are defined as follows:

Definition 3.3.6 (General Definition of the $\mathcal{U}c\#$i Operators)

$$
\begin{array}{lcll}
\mathcal{U}c\#\text{i} & : & U(T_n) \rightarrow U(T_i) & \\
\mathcal{U}c\#\text{i ABS} & = & \text{ABS} & \\
\mathcal{U}c\#\text{i } (u_0, \ldots, u_k) & = & (u_0, \ldots, u_k), & \text{if } T_i = T_n \\
 & = & u_j, & \text{if } T_i = T'_j,\ 1 \leq j \leq k
\end{array}
$$

□

Now that the operations on usage patterns have been defined, it remains to prove that they are monotonic and continuous. The proofs are not difficult, and are not included here.

3.4 Usage Counting Analysis

In this section, a usage counting analysis is presented which operates over the domains of usage patterns. This analysis determines the maximum number of times a value will be used in future computations within a program. The domains which are used in this analysis are shown in Figure 3.7.

$$
\begin{array}{lcl}
u \in \text{Usage} & = & U(T) \\
\phi_{\mathcal{U}} \in \text{Fve}_{\mathcal{U}} & = & (\text{Fv} \times \text{Int}) \rightarrow \text{Usage} \rightarrow \text{Usage}
\end{array}
$$

Figure 3.7: Usage Counting Analysis Domains

The future usage (or context) of a value of type T is an element of the usage domain $U(T)$ and is represented by u in this analysis. Each function in the function variable environment in the analysis gives the future usage of a given argument within a given function for a given context of function call.

The functionality of the usage counting analysis functions is shown in Figure 3.8.

$$\mathcal{U}_p\text{: } \text{Prog} \rightarrow \text{Fve}_{\mathcal{U}}$$

$$\mathcal{U}\text{: } \text{Exp} \rightarrow \text{Bv} \rightarrow \text{Usage} \rightarrow \text{Fve}_{\mathcal{U}} \rightarrow \text{Usage}$$

Figure 3.8: Usage Counting Analysis Functions

The function $\mathcal{U}_p$ gives the function variable environment resulting from the usage counting analysis of a program. The result of evaluating $\mathcal{U}[\![e]\!][\![x]\!]\ u\ \phi_{\mathcal{U}}$ gives the maximum number of times the variable x is used in future computations if the expression e appears in the context u. These functions are defined in Figure 3.9. The rules for this analysis can be explained as follows:

(U1) The result of evaluating a program is a function variable environment in which functions of the form $\mathcal{U}f\#k$ are introduced. Each function of the form $\mathcal{U}f\#k$ gives the future usage of argument number k within the function f for a given context of function call. The value of this function variable environment is determined using a least fixed point evaluation.
(U2) No part of a variable is used in a constant.
(U3) If the variable x is evaluated in a context u, then the usage of x is given by u. If any other variable is evaluated, then the variable x is absent.
(U4) Each of the arguments in a basic function application will be evaluated in a context 1, since they will be used only once. The total usage of the variable x is the total (using &) of its usage in each of these arguments.
(U5) If a constructor application is evaluated in a context u, then each of its arguments will be evaluated in a context given by the sub-component of u which corresponds to that argument. The total usage of the variable x is the total (using &) of its usage in each of these arguments.
(U6) If a function application is evaluated in a context u, then each of its arguments will be evaluated in a context given by the function variable environment for a call of the function in the context u. The total usage of the variable x is the total (using &) of its usage in each of these arguments.
(U7) If a **case** expression is evaluated in a context u, then the branches of the **case** expression will also be evaluated in the context u. The context in which the selector of the **case** expression will be evaluated depends upon which branch of the expression is selected. This context

$$
\begin{array}{lll}
\text{(U1)} & \mathcal{U}_p[\![e & \\
& \quad \textbf{where} & \\
& \quad f_1\ v_{11} \ldots v_{1k_1} \quad = \quad e_1 & \\
& \quad \vdots & \\
& \quad f_n\ v_{n1} \ldots v_{nk_n} \quad = \quad e_n]\!] & \\
& \qquad = \text{fix}\ (\lambda\phi_{\mathcal{U}}.[(\lambda u.\mathcal{U}[\![e_j]\!][\![v_{jk}]\!]\ u\ \phi_{\mathcal{U}})/\mathcal{U}f_j\#k]) &
\end{array}
$$

$$
\begin{array}{llll}
\text{(U2)} & \mathcal{U}[\![k]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} & = & \text{ABS} \\
\text{(U3)} & \mathcal{U}[\![v]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} & = & u, \quad \text{if } v = x \\
& & = & \text{ABS}, \quad \text{otherwise} \\
\text{(U4)} & \mathcal{U}[\![b\ e_1 \ldots e_n]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} & = & u \to (\mathcal{U}[\![e_1]\!][\![x]\!]\ 1\ \phi_{\mathcal{U}}\ \&\ \ldots\ \&\ \mathcal{U}[\![e_n]\!][\![x]\!]\ 1\ \phi_{\mathcal{U}})
\end{array}
$$

$$
\begin{array}{llll}
\text{(U5)} & \mathcal{U}[\![c\ e_1 \ldots e_n]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} & = & u \to (\mathcal{U}[\![e_1]\!][\![x]\!]\ u_1\ \phi_{\mathcal{U}}\ \&\ \ldots\ \&\ \mathcal{U}[\![e_n]\!][\![x]\!]\ u_n\ \phi_{\mathcal{U}}) \\
& \text{where} & & \\
& u_1 \quad = \quad \mathcal{U}c\#1\ u & & \\
& \vdots & & \\
& u_n \quad = \quad \mathcal{U}c\#n\ u & &
\end{array}
$$

$$
\begin{array}{llll}
\text{(U6)} & \mathcal{U}[\![f\ e_1 \ldots e_n]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} & = & u \to (\mathcal{U}[\![e_1]\!][\![x]\!]\ u_1\ \phi_{\mathcal{U}}\ \&\ \ldots\ \&\ \mathcal{U}[\![e_n]\!][\![x]\!]\ u_n\ \phi_{\mathcal{U}}) \\
& \text{where} & & \\
& u_1 \quad = \quad \phi_{\mathcal{U}}[\![\mathcal{U}f\#1]\!]\ u & & \\
& \vdots & & \\
& u_n \quad = \quad \phi_{\mathcal{U}}[\![\mathcal{U}f\#n]\!]\ u & &
\end{array}
$$

$$
\begin{array}{llll}
\text{(U7)} & \mathcal{U}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} & & \\
& & = & u \to (((\mathcal{U}[\![e_0]\!][\![x]\!]\ u_1\ \phi_{\mathcal{U}})\ \&\ (\mathcal{U}[\![e_1]\!][\![x]\!]\ u\ \phi_{\mathcal{U}})) \sqcup \ldots \\
& & & \sqcup\ ((\mathcal{U}[\![e_0]\!][\![x]\!]\ u_k\ \phi_{\mathcal{U}})\ \&\ (\mathcal{U}[\![e_k]\!][\![x]\!]\ u\ \phi_{\mathcal{U}}))) \\
& \text{where} & & \\
& p_1 \quad = \quad c_1\ v_{11} \ldots v_{1n_1} & & \\
& \vdots & & \\
& p_k \quad = \quad c_k\ v_{k1} \ldots v_{kn_k} & & \\
& u_1 \quad = \quad \mathcal{U}c_1(1,\mathcal{U}[\![e_1]\!][\![v_{11}]\!]\ u\ \phi_{\mathcal{U}},\ldots,\mathcal{U}[\![e_1]\!][\![v_{1n_1}]\!]\ u\ \phi_{\mathcal{U}}) & & \\
& \vdots & & \\
& u_k \quad = \quad \mathcal{U}c_k(1,\mathcal{U}[\![e_k]\!][\![v_{k1}]\!]\ u\ \phi_{\mathcal{U}},\ldots,\mathcal{U}[\![e_k]\!][\![v_{kn_k}]\!]\ u\ \phi_{\mathcal{U}}) & &
\end{array}
$$

Figure 3.9: Usage Counting Analysis

is given by the application of the abstract constructor (corresponding to the constructor in the pattern of the selected branch) to the usage patterns giving the usage of the pattern matching variables in the selected branch. The total usage of the variable x in the **case** expression is the total (using &) of its usage in the selector and its usage in the selected branch. Since it cannot be determined at compile-time which branch of the **case** expression will be evaluated, the least upper bound of the usage of the variable x when each branch is evaluated is used instead.

3.5 Proof of Correctness

Since the information obtained from usage counting analysis is going to be used to allow various optimisations to be performed, it must be shown that it is safe with respect to the usage counting store semantics. This will be the case if the future usage of a value obtained by usage counting analysis is a safe approximation to the increment in usage of the value in the usage counting store semantics due to the evaluation of the program. It will be a safe approximation if it is greater than the actual usage.

To determine the usage pattern corresponding to the increment in usage of a usage counting store value, the function δ is defined as follows.

Definition 3.5.1 (Usage Pattern Corresponding to the Increment in Usage of a Usage Counting Store Value) *The usage pattern corresponding to the increment in usage of a usage counting store value at location loc between the stores σ and σ' can be determined for each type of value using the function δ which is defined as follows:*

$$\begin{aligned}
&\delta\text{: } (\text{Loc} \times \text{Store}_{\mathcal{E}^{use}} \times \text{Store}_{\mathcal{E}^{use}}) \rightarrow U(T_A) \\
&\delta(loc, \sigma, \sigma') \quad = \quad \text{ABS}, \qquad\qquad\qquad \text{if } (\sigma\ loc) = \text{UNB or } (\sigma\ loc) = \bot \\
&\qquad\qquad\qquad\; = \quad \delta((\sigma\ loc), \sigma, \sigma'), \quad \text{if } (\sigma\ loc) \in \text{Loc} \\
&\qquad\qquad\qquad\; = \quad u, \qquad\qquad\qquad\quad\;\; \text{otherwise} \\
&\qquad\qquad\qquad\qquad\quad \text{where} \\
&\qquad\qquad\qquad\qquad\quad u \;=\; 0, \;\; \text{if } ((\sigma'\ loc) \downarrow 1) = ((\sigma\ loc) \downarrow 1) \\
&\qquad\qquad\qquad\qquad\qquad\; =\; 1, \;\; \text{if } ((\sigma'\ loc) \downarrow 1) - ((\sigma\ loc) \downarrow 1) = 1 \\
&\qquad\qquad\qquad\qquad\qquad\; =\; 2, \;\; \text{otherwise}
\end{aligned}$$

$$
\begin{array}{lll}
\delta\colon (\text{Loc} \times \text{Store}_{\mathcal{E}^{use}} \times \text{Store}_{\mathcal{E}^{use}}) \to U(\mathit{list}\ T) & & \\
\delta(\mathit{loc}, \sigma, \sigma') = \text{ABS}, & & \text{if } (\sigma\ \mathit{loc}) = \text{UNB or } (\sigma\ \mathit{loc}) = \bot \\
\quad = \delta((\sigma\ \mathit{loc}), \sigma, \sigma'), & & \text{if } (\sigma\ \mathit{loc}) \in \text{Loc} \\
\quad = \mathcal{U}\mathit{Nil}(u_0), & & \text{if } (\sigma\ \mathit{loc}) \downarrow 2 = \text{NIL} \\
\quad\quad \text{where} & & \\
\quad\quad u_0 = 0, & \text{if } ((\sigma'\ \mathit{loc}) \downarrow 1) = ((\sigma\ \mathit{loc}) \downarrow 1) & \\
\quad\quad\quad = 1, & \text{if } ((\sigma'\ \mathit{loc}) \downarrow 1) - ((\sigma\ \mathit{loc}) \downarrow 1) = 1 & \\
\quad\quad\quad = 2, & \text{otherwise} & \\
\quad = \mathcal{U}\mathit{Cons}(u_0, u_1, u_2), & & \text{if } (\sigma\ \mathit{loc}) \downarrow 2 \in \text{Conscell} \\
\quad\quad \text{where} & & \\
\quad\quad u_0 = 0, & \text{if } ((\sigma'\ \mathit{loc}) \downarrow 1) = ((\sigma\ \mathit{loc}) \downarrow 1) & \\
\quad\quad\quad = 1, & \text{if } ((\sigma'\ \mathit{loc}) \downarrow 1) - ((\sigma\ \mathit{loc}) \downarrow 1) = 1 & \\
\quad\quad\quad = 2, & \text{otherwise} & \\
\quad\quad u_1 = \delta(((\sigma\ \mathit{loc}) \downarrow 2) \downarrow 1, \sigma, \sigma') & & \\
\quad\quad u_2 = \delta(((\sigma\ \mathit{loc}) \downarrow 2) \downarrow 2, \sigma, \sigma') & &
\end{array}
$$

□

It is assumed that all closures have been evaluated before the usage pattern corresponding to the increment in usage of a usage counting store value is determined. If the value at the given location in the store is unbound or is undefined, then the corresponding usage pattern is ABS. If there is no increment in the usage of an atomic value, then the corresponding usage pattern is 0. If there is an increment of one in its usage, then the corresponding usage pattern is 1, otherwise it is 2. The usage pattern corresponding to a list value is determined recursively from the usage counting store value and gives the least upper bound of the usage of the spine cells of the list, and the least upper bound of the usage of the elements in the list.

Using this definition, the correctness of the usage counting analysis can be shown by proving the following theorem.

Theorem 3.5.2 (Correctness of Usage Counting Analysis)

for all $\rho_{\mathcal{E}^{use}} \in \text{Bve}_{\mathcal{E}^{use}}$, $\phi_{\mathcal{E}^{use}} \in \text{Fve}_{\mathcal{E}^{use}}$, $\sigma_{\mathcal{E}^{use}} \in \text{Store}_{\mathcal{E}^{use}}$, $\phi_{\mathcal{U}} \in \text{Fve}_{\mathcal{U}}$, $p \in \text{Prog}$, $e \in \text{Exp}$:

if $\mathcal{E}_p^{use}[\![p]\!] = (loc''', \sigma'''_{\mathcal{E}^{use}})$

and for all $f \in \text{dom}(\phi_{\mathcal{E}^{use}})$:

  if $\phi_{\mathcal{E}^{use}}[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$

  and $\delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$

  then **if** $\phi_{\mathcal{E}^{use}}[\![f]\!]\ loc'_1 \ldots loc'_n\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$

    and $(\phi_{\mathcal{U}}[\![\mathcal{U} f\#i]\!]\ u) \sqsubseteq \delta(loc'_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

    then $u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

and $\mathcal{E}^{use}[\![e]\!]\ \rho_{\mathcal{E}^{use}}\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$

and $\delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$

then for all $x_i \in \text{dom}(\rho_{\mathcal{E}^{use}})$:

  if $\mathcal{E}^{use}[\![e]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$

  and $(\mathcal{U}[\![e]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

  then $u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

□

Proof

The proof of this theorem can be found in Appendix B.1.

□

The following lemma states that the function variable environment in the usage counting analysis will satisfy the requirement in Theorem 3.5.2.

Lemma 3.5.3 (Correctness of Function Variable Environment)

for all $p \in$ Prog:

if $\mathcal{E}_p^{use}[\![p]\!] = force(\mathcal{E}[\![e]\!]\ (\lambda v.\bot)\ \phi_{\mathcal{E}^{use}}\ (\lambda loc.\text{UNB})) = (loc''', \sigma'''_{\mathcal{E}^{use}})$

and $\mathcal{U}_p[\![p]\!] = \phi_{\mathcal{U}}$

then for all $f \in \text{dom}(\phi_{\mathcal{E}^{use}})$, $\sigma_{\mathcal{E}^{use}} \in \text{Store}_{\mathcal{E}^{use}}$:

  if $\phi_{\mathcal{E}^{use}}[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$

  and $\delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$

  then **if** $\phi_{\mathcal{E}^{use}}[\![f]\!]\ loc'_1 \ldots loc'_n\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$

    and $(\phi_{\mathcal{U}}[\![\mathcal{U} f \# i]\!]\ u) \sqsubseteq \delta(loc'_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

    then $u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

□

Proof

The proof of this lemma can be found in Appendix B.2.

□

3.6 Examples

In this section, the results of applying usage counting analysis to the example functions given in Figure 2.2 are presented. The results of applying the analysis to the function *append* are shown in Figure 3.1.

Context	(0,ABS)	(0,0)	(0,1)	(0,2)	(1,ABS)	(1,0)
$\mathcal{U}$*append*#1	(1,ABS)	(1,0)	(1,1)	(1,2)	(1,ABS)	(1,0)
$\mathcal{U}$*append*#2	(0,ABS)	(0,0)	(0,1)	(0,2)	(1,ABS)	(1,0)
Context	(1,1)	(1,2)	(2,ABS)	(2,0)	(2,1)	(2,2)
$\mathcal{U}$*append*#1	(1,1)	(1,2)	(1,ABS)	(1,0)	(1,1)	(1,2)
$\mathcal{U}$*append*#2	(1,1)	(1,2)	(2,ABS)	(2,0)	(2,1)	(2,2)

Table 3.1: Usage Counting Analysis of the Function *append*

From this table, it can be seen that the spine cells in the first argument of *append* will never

be used more than once, and the list elements in the first argument will be used the same number of times as the list elements in the result of the function. The usage of the second argument of *append* will be exactly the same as the usage of the result of the function.

The results of applying the analysis to the function *reverse* are shown in Figure 3.2.

Context	(0,ABS)	(0,0)	(0,1)	(0,2)	(1,ABS)	(1,0)
$\mathcal{U}reverse\#1$	(1,ABS)	(1,0)	(1,1)	(1,2)	(1,ABS)	(1,0)
Context	(1,1)	(1,2)	(2,ABS)	(2,0)	(2,1)	(2,2)
$\mathcal{U}reverse\#1$	(1,1)	(1,2)	(1,ABS)	(1,0)	(1,1)	(1,2)

Table 3.2: Usage Counting Analysis of the Function *reverse*

From this table, it can be seen that the spine cells in the argument of *reverse* will never be used more than once, and the list elements in the argument will be used the same number of times as the list elements in the result of the function.

The results of applying the analysis to the function *accreverse* are shown in Figure 3.3.

Context	(0,ABS)	(0,0)	(0,1)	(0,2)	(1,ABS)	(1,0)
$\mathcal{U}accreverse\#1$	(1,ABS)	(1,0)	(1,1)	(1,2)	(1,ABS)	(1,0)
$\mathcal{U}accreverse\#2$	(0,ABS)	(0,0)	(0,1)	(0,2)	(1,ABS)	(1,0)
Context	(1,1)	(1,2)	(2,ABS)	(2,0)	(2,1)	(2,2)
$\mathcal{U}accreverse\#1$	(1,1)	(1,2)	(1,ABS)	(1,0)	(1,1)	(1,2)
$\mathcal{U}accreverse\#2$	(1,1)	(1,2)	(2,ABS)	(2,0)	(2,1)	(2,2)

Table 3.3: Usage Counting Analysis of the Function *accreverse*

From this table, it can be seen that the spine cells in the first argument of *accreverse* will never be used more than once, and the list elements in the first argument will be used the same number of times as the list elements in the result of the function. The usage of the second argument of *accreverse* will be exactly the same as the usage of the result of the function.

The results of applying the analysis to the function *flatten* are shown in Figure 3.4.

Context	(0,ABS)	(0,0)	(0,1)	(0,2)	(1,ABS)	(1,0)
$\mathcal{U}flatten\#1$	(1,(1,ABS))	(1,(1,0))	(1,(1,1))	(1,(1,2))	(1,(1,ABS))	(1,(1,0))
Context	(1,1)	(1,2)	(2,ABS)	(2,0)	(2,1)	(2,2)
$\mathcal{U}flatten\#1$	(1,(1,1))	(1,(1,2))	(1,(1,ABS))	(1,(1,0))	(1,(1,1))	(1,(1,2))

Table 3.4: Usage Counting Analysis of the Function *flatten*

From this table, it can be seen that no list cells in the argument of *flatten* will ever be used more than once, and the bottom level elements in each list in the argument will be used the same number of times as the list elements in the result of the function.

3.7 Related Work

In this section, other usage counting analyses, within the three frameworks of abstract interpretation, backward analysis and type inference, are considered.

3.7.1 Abstract Interpretation

An isolation interpretation is described in (Mycroft, 1981) which can be used to determine if data structures are used no more than once in a strict first order functional language. This extends previous work in (Schwarz, 1978) in which these isolation classes had to be supplied by the user. An approximate set of isolation patterns are determined for each value. This interpretation is relatively complex, and makes use of information obtained by two other static analyses; the E_{USES} interpretation and the E_{EXAM} interpretation. No proof of correctness is given for the isolation interpretation.

The sharing analyses described in (Jones & Le Métayer, 1989) and (Hamilton & Jones, 1990) are applicable to strict first order functional languages, and are similar to the isolation interpretation described in (Mycroft, 1981). They also make use of the information obtained by two other static analyses; *transmission* analysis and *necessity* analysis. These analyses are similar to the E_{USES} and E_{EXAM} interpretations described in (Mycroft, 1981). The domains of sharing patterns which are used in these analyses distinguish between the sharing of each of the spine cells in a list. To allow the compile-time analysis of sharing, these domains are cut off at a suitable depth. The correctness of the described sharing analyses are not considered.

In (Hudak, 1987), an abstract interpretation of reference counting in a first order strict functional language is presented. This involves counting the number of syntactic occurrences of values in a program. This differs from counting the number of times they are actually used, as is done in this chapter. A value may be referenced many times, but it might be used only once. To allow the analysis of reference counting at compile-time, 'sticky' reference counts are used. When a reference count reaches a certain maximum value, it cannot be reduced again. The analysis presented in (Hudak, 1987) uses an abstract store and is therefore likely to be inefficient. A similar analysis for a higher order strict language is described in (Andersen, 1990).

An update avoidance analysis is presented in (Marlow, 1993) which can be used to determine the number of times a value will be used in future computations. If the value is used no more than once, the cost of updating a closure with the result of its evaluation can be avoided. The analysis involves collecting a *bag* of variables which must be used when a given expression is evaluated. A bag is used because the same variable may be used more than once. The number of times a variable is used in evaluating the expression can then be determined by counting the number of occurrences of the variable in the bag. No proof of correctness is given for this analysis.

3.7.2 Backward Analysis

A simple backward analysis is described in (Hughes, 1988) which can be used to determine usage counting information. The domain used in this analysis is a simple flat domain similar to the domain defined in this chapter for values of atomic type, so it does not give very detailed usage counting information for structured data. The information obtained by this analysis can be used to optimise call-by-need to call-by-name, thus saving the cost of overwriting a closure with its value, and testing to see whether the overwrite has been performed.

A backward analysis for determining usage counting information for structured data is described in (Jensen & Mogensen, 1990) and (Jensen, 1990). This analysis is very similar to the usage counting analysis presented in this chapter. It is defined on an infinite domain of contexts, so the usual iterative method for finding fixpoints will not terminate in general. This situation is avoided by using a global environment which binds variables to their context and binds functions to the least upper bound of the contexts of the calls to them. This global environment is represented by a grammar, and it is possible to determine an approximation to this grammar at compile-time. Although the correctness of the analysis is considered in

(Jensen, 1990), no safety condition could be defined, and hence no proof of correctness could be given. A sketch is given of an extension of the analysis to higher order functions. This involves using a closure analysis like the one described in (Sestoft, 1989) to determine the set of possible abstract closures to which a function can be evaluated during the execution of a program. The least upper bound of the corresponding contexts of these abstract closures is then determined. A global environment is represented by a grammar as before, and an approximation to this grammar is determined at compile-time. Again, no proof of correctness is given for this higher order analysis.

3.7.3 Type Inference

The update avoidance analysis described in (Launchbury *et al.* , 1992) is a type scheme which can be used to determine usage counting information. This type scheme is defined on a domain similar to the usage counting domain presented in this chapter for values of an atomic type, so it does not give very detailed usage counting information for structured data. The information obtained by the analysis is used to avoid updating a closure with its value, if its value is used only once. No correctness proof is given for the analysis because no appropriate semantics could be defined as a reference for its correctness.

A type inference scheme for usage counting analysis is also presented in (Baker-Finch, 1993) and (Wright & Baker-Finch, 1993). This scheme is based on relevant logic. It involves monitoring applications of the contraction structural rule to determine the number of times a value is used. The usage count of a value is incremented each time the contraction rule is applied to it. The described work does not give an algorithm for assigning types to terms. Also, it does not deal with data structures, and recursion is considered only informally.

The type schemes described in (Wadler, 1990c; Guzmán & Hudak, 1990; Smetsers *et al.* , 1993) allow the user to indicate that a value will be used once. The linear type scheme described in (Wadler, 1990c) is based on linear logic (Girard, 1987). Values which are declared to be linear in this type scheme must be used exactly once. No distinction is made between sharing and absence. The type scheme described in (Guzmán & Hudak, 1990) is more loosely based on linear logic, and can be used to determine that values are used no more than once. This type scheme is therefore not as restrictive as the linear type scheme described in (Wadler, 1990c), but the type rules are considerably more complex. The unique type scheme described in (Smetsers *et al.* , 1993) makes use of graph reduction information to determine whether values are unique. A value is unique if there is exactly one path to it from the graph root.

3.8 Conclusion

In this chapter, it has been shown how cells which will become garbage within a program can be detected at compile-time. A cell will become garbage during the evaluation of an expression if it is unshared when it loses a reference. To determine that a cell is unshared, the store semantics presented in Section 2.4 were augmented to incorporate usage counting. Usage counting values in this semantics were then abstracted to usage patterns. These usage patterns are finite objects which indicate the number of times each part of a value is used. A usage counting analysis was then defined using these patterns to determine at compile-time the number of times each part of a value will be used in future computations. This analysis was then proved to be correct with respect to the usage counting store semantics.

Now that cells which will become garbage can be detected at compile-time, a number of optimisations can be performed to optimise the use of storage in programs. In Chapter 4, it is shown how this information can be used to validate compile-time garbage collection, and in Chapter 5, it is shown how this information can be used to guide the transformation when compile-time garbage avoidance is performed.

Chapter 4

Compile-Time Garbage Collection

Compile-time garbage collection involves annotating programs at compile-time to allow garbage cells to be collected automatically at run-time. This optimisation overcomes the problem of the programmer not being able to annotate functional programs in this way. Much work has already been done to show how compile-time garbage collection can be performed for strict languages, but not so much has been done for lazy languages. In this chapter, it is shown how information obtained by usage counting analysis can be used to annotate lazy programs for compile-time garbage collection.

Three different methods for compile-time garbage collection are presented. These are called compile-time garbage marking, explicit deallocation and destructive allocation. Compile-time garbage marking involves marking cells at their allocation to indicate that they will become garbage after their first use. These cells are returned to the memory manager immediately after their first use. Explicit deallocation involves explicitly returning cells to the memory manager at a particular point in a program. Destructive allocation involves reusing cells directly for further allocations within a program.

Store semantics are defined for programs which have been annotated for each of these methods of compile-time garbage collection, and the correctness of these store semantics are considered.

The remainder of this chapter is structured as follows:

- **Section 4.1**: existing methods for run-time garbage collection are described, and the relative advantages of each method are considered.
- **Section 4.2**: it is shown how programs can be annotated for compile-time garbage marking. A store semantics is defined for programs which have been annotated in this way, and the correctness of these store semantics is considered.
- **Section 4.3**: it is shown how programs can be annotated for explicit deallocation. A store semantics is defined for programs which have been annotated in this way, and the correctness of these store semantics is considered.
- **Section 4.4**: it is shown how programs can be annotated for destructive allocation. A store semantics is defined for programs which have been annotated in this way, and the correctness of these store semantics is considered.
- **Section 4.5**: related work is considered.
- **Section 4.6**: a summary of this chapter is given.

4.1 Run-Time Garbage Collection

In this section, existing methods for run-time garbage collection are described, and the relative advantages of each method are considered. Run-time garbage collection involves determining at run-time which store cells are no longer required by a program, and making these cells available for further use. This information is relevant in this chapter because the way in which storage is used at run-time must be considered when some of the described compile-time optimisations of store usage are performed. There are three main garbage collection strategies. These are reference counting, mark/scan and copying garbage collection. Each of these methods is now described in more detail, and the relative merits of each method are considered.

4.1.1 Reference Counting Garbage Collection

The idea of using reference counting for garbage collection was first suggested in (Collins, 1960). In this method of garbage collection, each cell in the store has an extra field which contains a number indicating how many references there are to the cell. When a cell is

allocated, its reference count is set to one. Each time a reference to the cell is created, the count is increased by one, and each time a reference to the cell is destroyed, its reference count is decreased by one. If the reference count of a cell reaches zero, then the cell is garbage since there are no references to it. Before the cell is collected, the reference counts of each cell it points to are also decremented. The reference counts of these cells may also be reduced to zero as a result, so the process is repeated. These garbage cells are added to a free list of cells which can be used for further allocations.

Some advantages of reference counting garbage collection over other methods of garbage collection are as follows:

- Garbage collection takes place continuously as part of the user program, and is not a logically separate process.
- The time spent on memory management is proportional to the number of transactions which take place, and not to the total number of active cells.
- It is suitable for use in a distributed environment, since altering a reference count is an atomic operation.

Some disadvantages of reference counting garbage collection are as follows:

- Cells in the free list will be scattered arbitrarily throughout the store. There will therefore be a low locality of reference in structures created from this free list. This may result in thrashing in a virtual memory system, and the benefits of using a cache may be lost in a real memory system.
- Cyclic structures cannot be collected easily since they always have a reference count of at least one (they point to themselves).
- Extra space is required in each store cell to hold the reference count. This must be about the same size as a pointer, since all store cells may point to the same cell.
- There is a constant overhead due to the need to update reference counts.

The problem of the space required to hold each reference count field can be alleviated by limiting their size. If the reference count for any cell reaches its maximum value, then it cannot be decremented. This approach is taken in the one-bit reference counting method described in (Wise & Friedman, 1977), which takes advantage of observations in (Clark & Green, 1977) and (Clark & Green, 1978) that most cells in LISP programs (around 97%) have

a reference count of one. Another garbage collector must still be used in this case to collect any cells in which the reference count has reached its maximum value. This garbage collector can also be used to collect any cyclic structures which cannot be collected using reference counting.

4.1.2 Mark/Scan Garbage Collection

The earliest garbage collectors were of the mark/scan type. This method of garbage collection makes use of a free list of unused cells in the store. Each time an allocation is to be performed, cells are removed from this free list to be allocated. When the free list is exhausted, the garbage collector is invoked to build a new free list from the garbage cells in the store.

To determine which cells are garbage, all the cells in the store which are accessible from any of the currently active pointers are *marked*. This marking is done by setting an extra mark bit in each cell. After this marking is complete, all the cells in the store are scanned. Any cells which are unmarked are added to the free list. As the store is scanned, the mark bit of each cell is reset ready for the next invocation of the garbage collector.

Mark/scan garbage collection is easy to implement, but it does have the following disadvantages:

- Extra space is required for the marking of cells.
- All active cells are visited twice (once during the mark phase and once during the scan phase), and all garbage cells are visited once (during the scan phase).
- As for reference counting, cells in the free list will be scattered arbitrarily throughout the store, so there will be a low locality of reference in structures created from this free list.

4.1.3 Copying Garbage Collection

Copying garbage collection involves copying all the store cells which are accessible from any of the currently active pointers to a contiguous region in memory. Any cells not in this region are therefore garbage and can be used for further allocations.

The idea of a two-space copying garbage collector was first suggested in (Fenichel & Yochelson, 1969). The method described divides the store into two semispaces. During the evaluation of a program, all new cells are allocated in one of the semispaces (the current semispace). If there is insufficient space for an allocation in the current semispace, the garbage collector

is invoked. The garbage collector copies all the cells in the current semispace which are accessible from any of the currently active pointers into the other semispace. All the cells in the current semispace are therefore garbage, and may be used for further allocations. The semispace to which all the active cells have been copied then becomes the current semispace.

A problem with two-space copying is that no more than 50% of the available storage space will be in use at any time. This problem can be alleviated by using multiple spaces. This approach is taken in the generational garbage collection methods described in (Lieberman & Hewitt, 1991; Ungar, 1984; Moon, 1984; Appel, 1989). In this approach, the store is divided into n regions of the same size, $n-1$ of which are active at any time. The remaining region is used for copying into. One region is garbage collected at a time, with the most recently allocated regions being garbage collected more frequently than older ones. This approach takes advantage of the observation that the most recently allocated store cells usually contain the most garbage (Clark, 1979).

Copying methods of garbage collection have the following advantages over other methods:

- All the free cells are compacted into a contiguous region of the store. Thus, successive cells will be allocated in successive store locations, which results in a higher locality of reference. This is advantageous in virtual memory systems and in real memory systems which make use of a cache.
- All the active cells are compacted into a contiguous region of the store. Thus, more compact storage techniques may be used for lists.
- All active cells are visited only once, and garbage cells are not visited at all.

Copying garbage collection therefore offers more advantages than other methods of garbage collection. The majority of garbage collectors which are currently used for functional languages are therefore of the copying type.

In the remainder of this chapter, it is shown how compile-time garbage collection can be performed. This compile-time garbage collection does not actually remove the need for run-time garbage collection, it merely serves to reduce the amount of garbage collection which must be performed at run-time. The compatability of the methods which are used for compile-time garbage collection and run-time garbage collection must therefore be considered. Three different methods for compile-time garbage collection are presented. These are called compile-time garbage marking, explicit deallocation and destructive allocation.

4.2 Compile-Time Garbage Marking

Compile-time garbage marking involves marking those cells which will become garbage after their first use. These cells can subsequently be freed and used for further allocations. In the next section, it is shown how the information obtained by usage counting analysis can be used to annotate programs for compile-time garbage marking. A store semantics is then defined for programs which have been annotated in this way, and the correctness of this store semantics is considered.

4.2.1 Annotating Programs for Compile-Time Garbage Marking

In this section, it is shown how programs can be annotated for compile-time garbage marking. Any cells which are used at most once can be marked at their allocation to indicate that they will become garbage after their first use. The safety condition for the annotation of cells in this way can be formally defined as follows.

Definition 4.2.1 (Safety of Annotation for Compile-Time Garbage Marking) *A location loc can be safely marked for compile-time garbage marking within a program p if the following condition holds:*

$$(\sigma' \ loc) \downarrow 1 \leq 1$$

where $\mathcal{E}_p^{use}[\![p]\!] = (loc', \sigma')$

□

Thus, any *Cons* cells or closures which are used at most once after their allocation can be annotated for compile-time garbage marking. It is shown how *Cons* cells can be annotated in this way in this section, but the same techniques can be used for the annotation of closures.

In order to annotate a program for compile-time garbage marking, the context of each expression is determined from an initial context for the program indicating that its result will be used exactly once. Any *Cons* applications which appear in a context in which the root cell of the resulting structure will be used at most once are annotated with the superscript m. These $Cons^m$ applications indicate that the root cell of the resulting structure will be marked when it is allocated. Any cells which are marked in this way can be returned to the memory manager immediately after they are used.

For example, consider the following expression:

$$accreverse\ (append\ xs\ ys)\ zs$$

None of the spine cells created in the result of the function call $(append\ xs\ ys)$ will be used more than once (see Table 3.3). Thus any cells created within the spine of this structure can be marked to indicate that they will become garbage after their first use. The annotation of this expression for compile-time garbage marking is shown in Figure 4.1.

$$
\begin{array}{lcll}
\multicolumn{4}{l}{accreverse\ (append'\ xs\ ys)\ zs} \\
\multicolumn{4}{l}{\textbf{where}} \\
accreverse\ xs\ ys & = & \multicolumn{2}{l}{\textbf{case}\ xs\ \textbf{of}} \\
 & & \quad Nil & :\ ys \\
 & & \quad Cons\ x\ xs & :\ accreverse\ xs\ (Cons\ x\ ys) \\
 & & & \\
append'\ xs\ ys & = & \multicolumn{2}{l}{\textbf{case}\ xs\ \textbf{of}} \\
 & & \quad Nil & :\ ys \\
 & & \quad Cons\ x\ xs & :\ Cons^m\ x\ (append'\ xs\ ys)
\end{array}
$$

Figure 4.1: Annotation of $accreverse\ (append\ xs\ ys)\ zs$ for Compile-Time Garbage Marking

4.2.2 Compile-Time Garbage Marking Store Semantics

In this section, it is shown how the store semantics of the language defined in Section 2.4 must be changed to handle programs which have been annotated for compile-time garbage marking. A usage counting store semantics is defined for programs which have been annotated in this way so that it can be shown that they are equivalent to the usage counting store semantics for unannotated programs presented in Section 3.1.

The semantic domains of the usage counting store semantics for compile-time garbage marking are shown in Figure 4.2. These domains are similar to those for the usage counting store semantics for unannotated programs given in Figure 3.1, except that an extra boolean flag is associated with each list cell. This flag is used to indicate whether or not the list cell will become garbage after its first use.

The functionality of the store semantic functions for performing compile-time garbage marking is shown in Figure 4.3. These functions are defined in Figures 4.4 and 4.5. They are very similar to the functions defined for the usage counting store semantics for unannotated programs given in Figures 3.3 and 3.4. List cells are marked at their allocation to indicate

$$
\begin{array}{lcl}
\mathrm{Val}_{\mathcal{E}ctgm} & = & (\mathrm{Loc} \times \mathrm{Store}_{\mathcal{E}ctgm})_\perp \\
x \in \mathrm{Eval} & = & \mathrm{Atom} \oplus \mathrm{List} \\
\mathrm{Atom} & = & \mathrm{Int} \oplus \mathrm{Bool} \\
\mathrm{Int} & = & \{0\}_\perp \oplus \{1\}_\perp \oplus \{-1\}_\perp \oplus \ldots \\
\mathrm{Bool} & = & \{\mathrm{TRUE}\}_\perp \oplus \{\mathrm{FALSE}\}_\perp \\
\mathrm{List} & = & \{\mathrm{NIL}\}_\perp \oplus \mathrm{Conscell} \\
\mathrm{Conscell} & = & (\mathrm{Bool} \times \mathrm{Loc} \times \mathrm{Loc})_\perp \\
loc \in \mathrm{Loc} & = & \mathrm{Int} \\
\mathrm{Uval} & = & (\mathrm{Use} \times \mathrm{Eval})_\perp \\
u \in \mathrm{Use} & = & \mathrm{Int} \\
\mathrm{Closure} & = & \mathrm{Store}_{\mathcal{E}ctgm} \to \mathrm{Val}_{\mathcal{E}ctgm} \\
\rho \in \mathrm{Bve}_{\mathcal{E}ctgm} & = & \mathrm{Bv} \to \mathrm{Loc} \\
\phi \in \mathrm{Fve}_{\mathcal{E}ctgm} & = & \mathrm{Fv} \to \mathrm{Loc}^* \to \mathrm{Store}_{\mathcal{E}ctgm} \to \mathrm{Val}_{\mathcal{E}ctgm} \\
\sigma \in \mathrm{Store}_{\mathcal{E}ctgm} & = & \mathrm{Loc} \to (\mathrm{Closure} \oplus \mathrm{Loc} \oplus \mathrm{Uval} \oplus \{\mathrm{UNB}\}_\perp)
\end{array}
$$

Figure 4.2: Compile-Time Garbage Marking Store Semantic Domains

whether or not they will become garbage after their first use. Any cells created by $Cons^m$ applications are marked in this way, but cells created by *Cons* applications are not. List cells are used during the evaluation of an expression only if they are the root cells of a selector in a **case** expression. When a cell is used in this way, a check is made to see if it is marked. If this is the case, then the cell is freed so that it can be used for further allocations.

The auxiliary functions of the usage counting store semantics for performing compile-time garbage marking are defined in Figure 4.6. These functions are very similar to the auxiliary functions of the usage counting store semantics for unannotated programs given in Figure 3.5, except that the function *dealloc* has been defined to deallocate a given location in the given store.

$$\begin{array}{ll}
\mathcal{E}_p^{ctgm}: & \text{Prog} \rightarrow \text{Val}_{\mathcal{E}^{ctgm}} \\
\mathcal{E}^{ctgm}: & \text{Exp} \rightarrow \text{Bve}_{\mathcal{E}^{ctgm}} \rightarrow \text{Fve}_{\mathcal{E}^{ctgm}} \rightarrow \text{Store}_{\mathcal{E}^{ctgm}} \rightarrow \text{Val}_{\mathcal{E}^{ctgm}} \\
\mathcal{B}^{ctgm}: & \text{Bas} \rightarrow \text{Loc}^* \rightarrow \text{Store}_{\mathcal{E}^{ctgm}} \rightarrow \text{Val}_{\mathcal{E}^{ctgm}} \\
\mathcal{C}^{ctgm}: & \text{Con} \rightarrow \text{Loc}^* \rightarrow \text{Store}_{\mathcal{E}^{ctgm}} \rightarrow \text{Val}_{\mathcal{E}^{ctgm}} \\
alloc: & ((\text{Closure} \oplus \text{Uval}) \times \text{Store}_{\mathcal{E}^{ctgm}}) \rightarrow \text{Val}_{\mathcal{E}^{ctgm}} \\
dealloc: & \text{Val}_{\mathcal{E}^{ctgm}} \rightarrow \text{Store}_{\mathcal{E}^{ctgm}} \\
inc: & \text{Val}_{\mathcal{E}^{ctgm}} \rightarrow \text{Val}_{\mathcal{E}^{ctgm}} \\
force: & \text{Val}_{\mathcal{E}^{ctgm}} \rightarrow \text{Val}_{\mathcal{E}^{ctgm}} \\
match: & (\text{Eval} \times \text{Con}) \rightarrow \text{Bool}
\end{array}$$

Figure 4.3: Compile-Time Garbage Marking Store Semantic Functions

4.2.3 Correctness

To prove that the store semantics for programs which have been annotated for compile-time garbage marking are correct, the following conjecture must be proved:

Conjecture 4.2.2 *The usage counting store semantics for programs which have been safely annotated for compile-time garbage marking as defined in Definition 4.2.1 are equivalent to the usage counting store semantics for unannotated programs.*

□

Sketch Proof

To prove this conjecture, it must be shown that the usage counts of values in both store semantics are the same, and that the standard semantic components of values in both store semantics are the same. If the usage counts of values in both store semantics can be shown to be the same, the annotations of programs for compile-time garbage marking will be correct with respect to the compile-time garbage marking store semantics. This will be the case since list cells are deallocated only if they will not be used again. If these deallocated cells are subsequently allocated again, their usage counts cannot be affected by any uses due to their

$$
\begin{array}{l}
\mathcal{E}_p^{ctgm}[\![e \\
\qquad \textbf{where} \\
\qquad f_1\ v_{11}\ldots v_{1k_1} \quad = \quad e_1 \\
\qquad \vdots \\
\qquad f_n\ v_{n1}\ldots v_{nk_n} \quad = \quad e_n]\!] = \mathit{force}(\mathcal{E}^{ctgm}[\![e]\!]\ (\lambda v.\bot)\ \phi_0\ (\lambda loc.\mathrm{UNB})) \\
\text{where} \\
\phi_0 \quad = \quad \text{fix}\ (\lambda\phi.[(\lambda loc_1\ldots\lambda loc_{k_j}.\lambda\sigma.\mathcal{E}^{ctgm}[\![e_j]\!]\ [loc_1/v_{j1},\ldots,loc_{k_j}/v_{jk_j}]\ \phi\ \sigma)/f_j])
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{E}^{ctgm}[\![k]\!]\ \rho\ \phi\ \sigma & = & \mathit{alloc}((0,k),\sigma) & \\
 & & & \\
\mathcal{E}^{ctgm}[\![v]\!]\ \rho\ \phi\ \sigma & = & (loc,\sigma'[loc/\rho[\![v]\!]]), & \text{if } (\sigma\ (\rho[\![v]\!])) \in \text{Closure} \\
 & & \text{where} & \\
 & & (loc,\sigma') \ = \ (\sigma\ (\rho[\![v]\!]))\ \sigma & \\
 & & & \\
 & = & ((\sigma\ (\rho[\![v]\!])),\sigma), & \text{otherwise} \\
 & & & \\
\mathcal{E}^{ctgm}[\![b\ e_1\ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \mathcal{B}^{ctgm}[\![b]\!]\ loc_1\ldots loc_n\ \sigma_n & \\
 & & \text{where} & \\
 & & (loc_1,\sigma_1) \ = \ \mathit{inc}(\mathcal{E}^{ctgm}[\![e_1]\!]\ \rho\ \phi\ \sigma) & \\
 & & \vdots & \\
 & & (loc_n,\sigma_n) \ = \ \mathit{inc}(\mathcal{E}^{ctgm}[\![e_n]\!]\ \rho\ \phi\ \sigma_{n-1}) & \\
 & & & \\
\mathcal{E}^{ctgm}[\![c\ e_1\ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \mathcal{C}^{ctgm}[\![c]\!]\ loc_1\ldots loc_n\ \sigma_n & \\
 & & \text{where} & \\
 & & (loc_1,\sigma_1) \ = \ \mathit{alloc}((\mathcal{E}^{ctgm}[\![e_1]\!]\ \rho\ \phi),\sigma) & \\
 & & \vdots & \\
 & & (loc_n,\sigma_n) \ = \ \mathit{alloc}((\mathcal{E}^{ctgm}[\![e_n]\!]\ \rho\ \phi),\sigma_{n-1}) & \\
 & & & \\
\mathcal{E}^{ctgm}[\![f\ e_1\ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \phi[\![f]\!]\ loc_1\ldots loc_n\ \sigma_n & \\
 & & \text{where} & \\
 & & (loc_1,\sigma_1) \ = \ \mathit{alloc}((\mathcal{E}^{ctgm}[\![e_1]\!]\ \rho\ \phi),\sigma) & \\
 & & \vdots & \\
 & & (loc_n,\sigma_n) \ = \ \mathit{alloc}((\mathcal{E}^{ctgm}[\![e_n]\!]\ \rho\ \phi),\sigma_{n-1}) &
\end{array}
$$

Figure 4.4: Compile-Time Garbage Marking Store Semantics

$$
\begin{array}{lll}
\mathcal{E}^{ctgm}[\![\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]\ \rho\ \phi\ \sigma & & \\
\quad = \mathcal{E}^{ctgm}[\![e_i]\!]\ \rho[x \downarrow 2/v_1, \ldots, x \downarrow (n+1)/v_n]\ \phi\ \sigma'' & & \\
\quad\quad \text{where} & & \\
\quad\quad (loc, \sigma') & = & inc(\mathcal{E}^{ctgm}[\![e_0]\!]\ \rho\ \phi\ \sigma) \\
\quad\quad (u, x) & = & \sigma'\ loc \\
\quad\quad p_i & = & c\ v_1 \ldots v_n \text{ and } match(x, c) \\
\quad\quad \sigma'' & = & dealloc(loc, \sigma'), \quad \text{if } x \in \text{Conscell and } x \downarrow 1 = \text{TRUE} \\
 & = & \sigma', \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B}^{ctgm}[\![+]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 + x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma\ loc_1 \\
 & & (u_2, x_2) = \sigma\ loc_2
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B}^{ctgm}[\![-]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 - x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma\ loc_1 \\
 & & (u_2, x_2) = \sigma\ loc_2
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B}^{ctgm}[\![<]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 < x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma\ loc_1 \\
 & & (u_2, x_2) = \sigma\ loc_2
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B}^{ctgm}[\![=]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 = x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma\ loc_1 \\
 & & (u_2, x_2) = \sigma\ loc_2 \\
 & \vdots &
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{C}^{ctgm}[\![True]\!] & = & \lambda\sigma.alloc((0,\text{TRUE}),\sigma) \\
\mathcal{C}^{ctgm}[\![False]\!] & = & \lambda\sigma.alloc((0,\text{FALSE}),\sigma) \\
\mathcal{C}^{ctgm}[\![Nil]\!] & = & \lambda\sigma.alloc((0,\text{NIL}),\sigma) \\
\mathcal{C}^{ctgm}[\![Cons]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0,(\text{FALSE},loc_1,loc_2)),\sigma) \\
\mathcal{C}^{ctgm}[\![Cons^m]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0,(\text{TRUE},loc_1,loc_2)),\sigma)
\end{array}
$$

Figure 4.5: Compile-Time Garbage Marking Store Semantics (continued)

$$
\begin{array}{lcll}
alloc(v,\sigma) & = & (loc, \sigma[v/loc]) & \\
 & & \text{where} & \\
 & & \sigma\ loc \quad = \quad \text{UNB} & \\[1em]
dealloc(loc,\sigma) & = & \sigma[\text{UNB}/loc] & \\[1em]
inc(loc,\sigma) & = & (loc, \sigma[(u+1, x)/loc]) & \\
 & & \text{where} & \\
 & & (u, x) \quad = \quad \sigma\ loc & \\[1em]
force(loc,\sigma) & = & (loc,\sigma), & \text{if } (\sigma\ loc) = \bot \\
 & & & \text{or } (\sigma\ loc) = \text{UNB} \\[1em]
 & = & (loc',\sigma'[loc'/loc]), & \text{if } (\sigma\ loc) \in \text{Closure} \\
 & & \text{where} & \\
 & & (loc', \sigma') \quad = \quad force((\sigma\ loc)\ \sigma) & \\[1em]
 & = & force((\sigma\ loc),\sigma), & \text{if } (\sigma\ loc) \in \text{Loc} \\[1em]
 & = & inc(loc,\sigma_2[(u,(x \downarrow 1, loc_1, loc_2))/loc]), & \text{if } (\sigma\ loc) \in \text{Uval} \\
 & & \text{where} & \text{and } x \in \text{Conscell} \\
 & & (u, x) \quad = \quad \sigma\ loc & \\
 & & (loc_1, \sigma_1) \quad = \quad force(x \downarrow 2, \sigma) & \\
 & & (loc_2, \sigma_2) \quad = \quad force(x \downarrow 3, \sigma_1) & \\[1em]
 & = & inc(loc,\sigma), & \text{otherwise} \\[1em]
match(x,c) & = & (x = \text{TRUE and } c = True) & \\
 & & \text{or } (x = \text{FALSE and } c = False) & \\
 & & \text{or } (x = \text{NIL and } c = Nil) & \\
 & & \text{or } (x \in \text{Conscell and } c = Cons) &
\end{array}
$$

Figure 4.6: Compile-Time Garbage Marking Store Semantics (auxiliary functions)

previous allocation. If the standard semantic components of values in both store semantics can be shown to be the same, then the result of evaluating programs which have been annotated for compile-time garbage marking will be equivalent to the result of evaluating the same programs before they were annotated. This will be the case since list cells are deallocated only if they will not be used again. Only values which are used again in a program can affect its result. The proof of this conjecture remains as further work.

4.3 Explicit Deallocation

The compile-time garbage marking method described in the previous section requires an extra bit per cell to indicate whether or not the cell is marked. The extra space required for this may be more than the space which is saved by using this method. Also, extra time is required to check each cell to see if it is marked. It is therefore unlikely that compile-time garbage marking is suitable for practical use.

If it could be determined that a cell will always become garbage at a particular point in a program, then there would be no need to mark it and check it, since it could always be deallocated at this point. The program could therefore be annotated to indicate that the cell can always be deallocated at this point. This form of compile-time garbage collection is called *explicit deallocation*. In the next section, it is shown how programs can be annotated for explicit deallocation. A store semantics is then defined for programs which have been annotated in this way, and the correctness of this store semantics is considered.

4.3.1 Annotating Programs for Explicit Deallocation

In this section, it is shown how programs can be annotated for explicit deallocation. It is shown only how *Cons* cells can be explicitly deallocated, but the same techniques can be used for the explicit deallocation of closures. If it can be determined that the root cell of a list which is the selector in a **case** expression is always unshared, then it can be explicitly deallocated after it has been used within the **case** expression. The usage counting analysis described in Section 3.4 does not provide this sharing information; it can be used only to determine whether a value is used at most once in *future* computations, not in *all* computations. It is shown in (Hamilton, 1992b) how usage counting analysis can be combined with an abstract interpretation to determine whether a value is used at most once in all computations. The safety condition for the explicit deallocation of cells can be formally defined as follows.

Definition 4.3.1 (Safety of Explicit Deallocation) *A location loc can be safely deallocated within a program p when the current store is σ if the following condition holds:*

$$(\sigma\ loc) \downarrow 1 = (\sigma'\ loc) \downarrow 1$$

where $\mathcal{E}_p^{use}[\![p]\!] = (loc', \sigma')$

□

The usage analysis described in (Hamilton, 1992b) can be used to determine whether cells satisfy this safety condition. To indicate that the root cell of a **case** selector can be explicitly deallocated, any *Cons* applications in the patterns of the **case** expression are annotated with the superscript d. These $Cons^d$ applications indicate that the root cell of the resulting structure is garbage. For example, consider the following expression:

$$accreverse\ (flatten\ xss)\ ys$$

All the spine cells in the result of the function call (*flatten xss*) are unshared. They can therefore be explicitly deallocated within the **case** expression in the *accreverse* function. The annotation of this expression for explicit deallocation is shown in Figure 4.7.

$$
\begin{array}{lll}
\multicolumn{3}{l}{accreverse'\ (flatten\ xss)\ ys} \\
\multicolumn{3}{l}{\mathbf{where}} \\
accreverse'\ xs\ ys & = & \mathbf{case}\ xs\ \mathbf{of} \\
 & & \quad Nil \quad : \quad ys \\
 & & \quad Cons^d\ x\ xs \quad : \quad accreverse'\ xs\ (Cons\ x\ ys) \\
 & & \\
flatten\ xss & = & \mathbf{case}\ xss\ \mathbf{of} \\
 & & \quad Nil \quad : \quad Nil \\
 & & \quad Cons\ xs\ xss \quad : \quad append\ xs\ (flatten\ xss) \\
 & & \\
append\ xs\ ys & = & \mathbf{case}\ xs\ \mathbf{of} \\
 & & \quad Nil \quad : \quad ys \\
 & & \quad Cons\ x\ xs \quad : \quad Cons\ x\ (append\ xs\ ys)
\end{array}
$$

Figure 4.7: Annotation of *accreverse* (*flatten xss*) *ys* for Explicit Deallocation

4.3.2 Explicit Deallocation Store Semantics

In this section, it is shown how the store semantics of the language defined in Section 2.4 must be changed to handle programs which have been annotated for explicit deallocation as described in the previous section. A usage counting store semantics is defined for programs which have been annotated in this way so that they can be shown to be equivalent to the usage counting store semantics for unannotated programs presented in Section 3.1.

The semantic domains of the usage counting store semantics for explicit deallocation are shown in Figure 4.8.

$$
\begin{array}{lcl}
\mathrm{Val}_{\mathcal{E}ed} & = & (\mathrm{Loc} \times \mathrm{Store}_{\mathcal{E}ed})_\perp \\
x \in \mathrm{Eval} & = & \mathrm{Atom} \oplus \mathrm{List} \\
\mathrm{Atom} & = & \mathrm{Int} \oplus \mathrm{Bool} \\
\mathrm{Int} & = & \{0\}_\perp \oplus \{1\}_\perp \oplus \{-1\}_\perp \oplus \ldots \\
\mathrm{Bool} & = & \{\mathrm{TRUE}\}_\perp \oplus \{\mathrm{FALSE}\}_\perp \\
\mathrm{List} & = & \{\mathrm{NIL}\}_\perp \oplus \mathrm{Conscell} \\
\mathrm{Conscell} & = & (\mathrm{Loc} \times \mathrm{Loc})_\perp \\
loc \in \mathrm{Loc} & = & \mathrm{Int} \\
\mathrm{Uval} & = & (\mathrm{Use} \times \mathrm{Eval})_\perp \\
u \in \mathrm{Use} & = & \mathrm{Int} \\
\mathrm{Closure} & = & \mathrm{Store}_{\mathcal{E}ed} \rightarrow \mathrm{Val}_{\mathcal{E}ed} \\
\rho \in \mathrm{Bve}_{\mathcal{E}ed} & = & \mathrm{Bv} \rightarrow \mathrm{Loc} \\
\phi \in \mathrm{Fve}_{\mathcal{E}ed} & = & \mathrm{Fv} \rightarrow \mathrm{Loc}^* \rightarrow \mathrm{Store}_{\mathcal{E}ed} \rightarrow \mathrm{Val}_{\mathcal{E}ed} \\
\sigma \in \mathrm{Store}_{\mathcal{E}ed} & = & \mathrm{Loc} \rightarrow (\mathrm{Closure} \oplus \mathrm{Loc} \oplus \mathrm{Uval} \oplus \{\mathrm{UNB}\}_\perp)
\end{array}
$$

Figure 4.8: Explicit Deallocation Store Semantic Domains

These domains are similar to those for the usage counting store semantics for unannotated programs given in Figure 3.1.

The functionality of the store semantic functions for performing explicit deallocation is shown in Figure 4.9. These functions are defined in Figures 4.10 and 4.11. They are very similar to the functions defined for the usage counting store semantics for unannotated programs given in Figures 3.3 and 3.4. If the selector in a **case** expression is a non-empty list and the pattern in one of the branches contains an application of a $Cons^d$ constructor, then the root cell of the selector is freed for further use.

The auxiliary functions of the store semantics for performing explicit deallocation are defined in Figure 4.12. These functions are very similar to the auxiliary functions of the

$\mathcal{E}_p^{ed}$:	Prog $\rightarrow$ $\text{Val}_{\mathcal{E}^{ed}}$
$\mathcal{E}^{ed}$:	Exp $\rightarrow$ $\text{Bve}_{\mathcal{E}^{ed}}$ $\rightarrow$ $\text{Fve}_{\mathcal{E}^{ed}}$ $\rightarrow$ $\text{Store}_{\mathcal{E}^{ed}}$ $\rightarrow$ $\text{Val}_{\mathcal{E}^{ed}}$
$\mathcal{B}^{ed}$:	Bas $\rightarrow$ Loc^* $\rightarrow$ $\text{Store}_{\mathcal{E}^{ed}}$ $\rightarrow$ $\text{Val}_{\mathcal{E}^{ed}}$
$\mathcal{C}^{ed}$:	Con $\rightarrow$ Loc^* $\rightarrow$ $\text{Store}_{\mathcal{E}^{ed}}$ $\rightarrow$ $\text{Val}_{\mathcal{E}^{ed}}$
alloc:	$((\text{Closure} \oplus \text{Uval}) \times \text{Store}_{\mathcal{E}^{ed}}) \rightarrow \text{Val}_{\mathcal{E}^{ed}}$
dealloc:	$\text{Val}_{\mathcal{E}^{ed}} \rightarrow \text{Store}_{\mathcal{E}^{ed}}$
inc:	$\text{Val}_{\mathcal{E}^{ed}} \rightarrow \text{Val}_{\mathcal{E}^{ed}}$
force:	$\text{Val}_{\mathcal{E}^{ed}} \rightarrow \text{Val}_{\mathcal{E}^{ed}}$
match:	$(\text{Eval} \times \text{Con}) \rightarrow \text{Bool}$

Figure 4.9: Explicit Deallocation Store Semantic Functions

usage counting store semantics for unannotated programs given in Figure 3.5, except that the function *dealloc* has been defined to deallocate a given location in the given store.

4.3.3 Correctness

To prove that the store semantics for programs which have been annotated for explicit deallocation are correct, the following conjecture must be proved:

Conjecture 4.3.2 *The usage counting store semantics for programs which have been safely annotated for explicit deallocation as defined in Definition 4.3.1 are equivalent to the usage counting store semantics for unannotated programs.*

□

Sketch Proof

The proof of this conjecture would be similar to the proof of Conjecture 4.2.2 for compile-time garbage marking. It also remains as further work.

$$
\begin{array}{l}
\mathcal{E}_p^{ed}[\![e \\
\quad \textbf{where} \\
\quad f_1\ v_{11} \ldots v_{1k_1} \quad = \quad e_1 \\
\quad \vdots \\
\quad f_n\ v_{n1} \ldots v_{nk_n} \quad = \quad e_n]\!] = \mathit{force}(\mathcal{E}^{ed}[\![e]\!]\ (\lambda v.\bot)\ \phi_0\ (\lambda loc.\mathrm{UNB})) \\
\text{where} \\
\phi_0 \quad = \quad \text{fix}\ (\lambda\phi.[(\lambda loc_1 \ldots \lambda loc_{k_j}.\lambda\sigma.\mathcal{E}^{ed}[\![e_j]\!]\ [loc_1/v_{j1}, \ldots, loc_{k_j}/v_{jk_j}]\ \phi\ \sigma)/f_j])
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{E}^{ed}[\![k]\!]\ \rho\ \phi\ \sigma & = & \mathit{alloc}((0,k),\sigma) & \\
 & & & \\
\mathcal{E}^{ed}[\![v]\!]\ \rho\ \phi\ \sigma & = & (loc, \sigma'[loc/\rho[\![v]\!]]), & \text{if } (\sigma\ (\rho[\![v]\!])) \in \text{Closure} \\
 & & \text{where} & \\
 & & (loc, \sigma') \quad = \quad (\sigma\ (\rho[\![v]\!]))\ \sigma & \\
 & & & \\
 & = & ((\sigma\ (\rho[\![v]\!])), \sigma), & \text{otherwise} \\
 & & & \\
\mathcal{E}^{ed}[\![b\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \mathcal{B}^{ed}[\![b]\!]\ loc_1 \ldots loc_n\ \sigma_n & \\
 & & \text{where} & \\
 & & (loc_1, \sigma_1) \quad = \quad \mathit{inc}(\mathcal{E}^{ed}[\![e_1]\!]\ \rho\ \phi\ \sigma) & \\
 & & \qquad\qquad\quad \vdots & \\
 & & (loc_n, \sigma_n) \quad = \quad \mathit{inc}(\mathcal{E}^{ed}[\![e_n]\!]\ \rho\ \phi\ \sigma_{n-1}) & \\
 & & & \\
\mathcal{E}^{ed}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \mathcal{C}^{ed}[\![c]\!]\ loc_1 \ldots loc_n\ \sigma_n & \\
 & & \text{where} & \\
 & & (loc_1, \sigma_1) \quad = \quad \mathit{alloc}((\mathcal{E}^{ed}[\![e_1]\!]\ \rho\ \phi), \sigma) & \\
 & & \qquad\qquad\quad \vdots & \\
 & & (loc_n, \sigma_n) \quad = \quad \mathit{alloc}((\mathcal{E}^{ed}[\![e_n]\!]\ \rho\ \phi), \sigma_{n-1}) & \\
 & & & \\
\mathcal{E}^{ed}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \phi[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_n & \\
 & & \text{where} & \\
 & & (loc_1, \sigma_1) \quad = \quad \mathit{alloc}((\mathcal{E}^{ed}[\![e_1]\!]\ \rho\ \phi), \sigma) & \\
 & & \qquad\qquad\quad \vdots & \\
 & & (loc_n, \sigma_n) \quad = \quad \mathit{alloc}((\mathcal{E}^{ed}[\![e_n]\!]\ \rho\ \phi), \sigma_{n-1}) &
\end{array}
$$

Figure 4.10: Explicit Deallocation Store Semantics

$$
\begin{array}{lll}
\mathcal{E}^{ed}[\![\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k]\!] \, \rho \, \phi \, \sigma & & \\
 & = & \mathcal{E}^{ed}[\![e_i]\!] \, \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n] \, \phi \, \sigma'' \\
 & & \text{where} \\
 & & \begin{array}{lll}
 (loc, \sigma') & = & inc(\mathcal{E}^{ed}[\![e_0]\!] \, \rho \, \phi \, \sigma) \\
 (u, x) & = & \sigma' \, loc \\
 p_i & = & c \; v_1 \ldots v_n \text{ and } match(x, c) \\
 \sigma'' & = & dealloc(loc, \sigma'), \quad \text{if } c = Cons^d \\
 & = & \sigma', \quad \text{otherwise}
 \end{array}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B}^{ed}[\![+]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 + x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma \; loc_1 \\
 & & (u_2, x_2) = \sigma \; loc_2
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B}^{ed}[\![-]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 - x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma \; loc_1 \\
 & & (u_2, x_2) = \sigma \; loc_2
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B}^{ed}[\![<]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 < x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma \; loc_1 \\
 & & (u_2, x_2) = \sigma \; loc_2
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B}^{ed}[\![=]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 = x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma \; loc_1 \\
 & & (u_2, x_2) = \sigma \; loc_2 \\
 & \vdots &
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{C}^{ed}[\![True]\!] & = & \lambda\sigma.alloc((0,\text{TRUE}),\sigma) \\
\mathcal{C}^{ed}[\![False]\!] & = & \lambda\sigma.alloc((0,\text{FALSE}),\sigma) \\
\mathcal{C}^{ed}[\![Nil]\!] & = & \lambda\sigma.alloc((0,\text{NIL}),\sigma) \\
\mathcal{C}^{ed}[\![Cons]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0,(loc_1, loc_2)),\sigma)
\end{array}
$$

Figure 4.11: Explicit Deallocation Store Semantics (continued)

$$
\begin{array}{lcll}
alloc(v,\sigma) & = & (loc, \sigma[v/loc]) & \\
 & & \text{where} & \\
 & & \sigma\ loc \;\; = \;\; \text{UNB} & \\
 & & & \\
dealloc(loc,\sigma) & = & \sigma[\text{UNB}/loc] & \\
 & & & \\
inc(loc,\sigma) & = & (loc, \sigma[(u+1,x)/loc]) & \\
 & & \text{where} & \\
 & & (u,x) \;\; = \;\; \sigma\ loc & \\
 & & & \\
force(loc,\sigma) & = & (loc,\sigma), & \text{if } (\sigma\ loc) = \bot \\
 & & & \text{or } (\sigma\ loc) = \text{UNB} \\
 & & & \\
 & = & (loc',\sigma'[loc'/loc]), & \text{if } (\sigma\ loc) \in \text{Closure} \\
 & & \text{where} & \\
 & & (loc',\sigma') \;\; = \;\; force((\sigma\ loc)\ \sigma) & \\
 & & & \\
 & = & force((\sigma\ loc),\sigma), & \text{if } (\sigma\ loc) \in \text{Loc} \\
 & & & \\
 & = & inc(loc,\sigma_2[(u,(loc_1,loc_2))/loc]), & \text{if } (\sigma\ loc) \in \text{Uval} \\
 & & \text{where} & \text{and } x \in \text{Conscell} \\
 & & (u,x) \;\; = \;\; \sigma\ loc & \\
 & & (loc_1,\sigma_1) \;\; = \;\; force(x \downarrow 1, \sigma) & \\
 & & (loc_2,\sigma_2) \;\; = \;\; force(x \downarrow 2, \sigma_1) & \\
 & & & \\
 & = & inc(loc,\sigma), & \text{otherwise} \\
 & & & \\
match(x,c) & = & (x = \text{TRUE and } c = True) & \\
 & & \text{or } (x = \text{FALSE and } c = False) & \\
 & & \text{or } (x = \text{NIL and } c = Nil) & \\
 & & \text{or } (x \in \text{Conscell and } c = Cons) & \\
 & & \text{or } (x \in \text{Conscell and } c = Cons^d) &
\end{array}
$$

Figure 4.12: Explicit Deallocation Store Semantics (auxiliary functions)

4.4 Destructive Allocation

Explicit deallocation requires that any cells which are explicitly deallocated are added to a free list. This method of compile-time garbage collection can therefore be used only if the run-time garbage collector also makes use of a free list. As was explained in Section 4.1, the most efficient methods for performing run-time garbage collection do not make use of a free list. It is therefore concluded that explicit deallocation is of limited use.

To avoid the need for a run-time free list, garbage cells could be reused directly within a program. This form of compile-time garbage collection is called *destructive allocation*. In the next section, it is shown how programs can be annotated for destructive allocation. A store semantics is then defined for programs which have been annotated in this way, and the correctness of this store semantics is considered.

4.4.1 Annotating Programs for Destructive Allocation

In this section, it is shown how programs can be annotated for destructive allocation. It is shown only how *Cons* cells can be destructively allocated, but the same techniques can be used for the destructive allocation of closures.

As for explicit deallocation, if it can be determined that the root cell of a list which is the selector in a **case** expression is always unshared, then it can be destructively allocated after it has been used within the **case** expression. The root cell of the selector can be reused within the selected branch of the **case** expression if it contains a *Cons* application in its pattern. The safety condition for the destructive allocation of cells can be formally defined as follows.

Definition 4.4.1 (Safety of Destructive Allocation) *A location loc can be safely destructively allocated within a program p when the current store is σ if the following condition holds:*

$$(\sigma\ loc) \downarrow 1 = (\sigma'\ loc) \downarrow 1$$

where $\mathcal{E}_p^{use}[\![p]\!] = (loc', \sigma')$

□

To indicate that the root cell of a **case** selector can be destructively allocated, any *Cons* applications in the patterns of the **case** expression are superscripted with a variable which represents the root cell of the selector. The variable name which is used in this annotation should not clash with any of the variables in the branch of the **case** expression. In a branch in which the pattern has been changed in this way, one *Cons* application can also be superscripted with the same variable to indicate that the root cell of the selector can be used to hold the result of the application.

For example, consider the following expression in the definition of the function *reverse* (see Figure 2.2):

$$append\ (reverse\ xs)\ (Cons\ x\ Nil)$$

All the spine cells in the result of the function call (*reverse xs*) are unshared. They can therefore be destructively allocated within the **case** expression in the *append* function. The annotation of this expression for destructive allocation is shown in Figure 4.13.

$$
\begin{array}{lll}
\multicolumn{3}{l}{append'\ (reverse'\ xs)\ (Cons\ x\ Nil)} \\
\multicolumn{3}{l}{\textbf{where}} \\
append'\ xs\ ys & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Nil \qquad\quad :\ ys \\
 & & \quad Cons^v\ x\ xs\ :\ Cons^v\ x\ (append'\ xs\ ys) \\
 & & \\
reverse'\ xs & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad Nil \qquad\quad :\ Nil \\
 & & \quad Cons\ x\ xs\ :\ append'\ (reverse'\ xs)\ (Cons\ x\ Nil)
\end{array}
$$

Figure 4.13: Annotation of *append* (*reverse xs*) (*Cons x Nil*) for Destructive Allocation

4.4.2 Destructive Allocation Store Semantics

In this section, it is shown how the store semantics of the language defined in Section 2.4 must be changed to handle programs which have been annotated for destructive allocation as described in the previous section. A usage counting store semantics is defined for programs which have been annotated in this way so that it can be shown that they are equivalent to the usage counting store semantics for unannotated programs presented in Section 3.1.

The semantic domains of the usage counting store semantics for destructive allocation are shown in Figure 4.14. These domains are similar to those for the usage counting store semantics for unannotated programs given in Figure 3.1.

The functionality of the store semantic functions for performing destructive allocation is shown in Figure 4.15. These functions are defined in Figures 4.16 and 4.17. They are very similar to the functions defined for the store semantics for unannotated programs given in Figures 2.9 and 2.10. If the selector in a **case** expression matches with a pattern of the form $Cons^v\ v_1 \ldots v_n$, then the variable v is bound to the root cell of the selector. If any $Cons^v$ applications are subsequently encountered during the evaluation of an expression, then the root cell of the variable v is used to hold the result of the application.

$$
\begin{array}{lcl}
\text{Val}_{\mathcal{E}da} & = & (\text{Loc} \times \text{Store}_{\mathcal{E}da})_\perp \\
x \in \text{Eval} & = & \text{Atom} \oplus \text{List} \\
\text{Atom} & = & \text{Int} \oplus \text{Bool} \\
\text{Int} & = & \{0\}_\perp \oplus \{1\}_\perp \oplus \{-1\}_\perp \oplus \ldots \\
\text{Bool} & = & \{\text{TRUE}\}_\perp \oplus \{\text{FALSE}\}_\perp \\
\text{List} & = & \{\text{NIL}\}_\perp \oplus \text{Conscell} \\
\text{Conscell} & = & (\text{Loc} \times \text{Loc})_\perp \\
loc \in \text{Loc} & = & \text{Int} \\
\text{Uval} & = & (\text{Use} \times \text{Eval})_\perp \\
u \in \text{Use} & = & \text{Int} \\
\text{Closure} & = & \text{Store}_{\mathcal{E}da} \rightarrow \text{Val}_{\mathcal{E}da} \\
\rho \in \text{Bve}_{\mathcal{E}da} & = & \text{Bv} \rightarrow \text{Loc} \\
\phi \in \text{Fve}_{\mathcal{E}da} & = & \text{Fv} \rightarrow \text{Loc}^* \rightarrow \text{Store}_{\mathcal{E}da} \rightarrow \text{Val}_{\mathcal{E}da} \\
\sigma \in \text{Store}_{\mathcal{E}da} & = & \text{Loc} \rightarrow (\text{Closure} \oplus \text{Loc} \oplus \text{Uval} \oplus \{\text{UNB}\}_\perp)
\end{array}
$$

Figure 4.14: Destructive Allocation Store Semantic Domains

The auxiliary functions of the store semantics for performing destructive allocation are defined in Figure 4.18. These functions are very similar to the auxiliary functions of the usage counting store semantics for unannotated programs given in Figure 3.5. No deallocation function is required since garbage cells are reused within programs rather than being added to a free list.

$$\begin{aligned}
\mathcal{E}_p^{da}:&\quad \text{Prog} \rightarrow \text{Val}_{\mathcal{E}^{da}}\\
\mathcal{E}^{da}:&\quad \text{Exp} \rightarrow \text{Bve}_{\mathcal{E}^{da}} \rightarrow \text{Fve}_{\mathcal{E}^{da}} \rightarrow \text{Store}_{\mathcal{E}^{da}} \rightarrow \text{Val}_{\mathcal{E}^{da}}\\
\mathcal{B}^{da}:&\quad \text{Bas} \rightarrow \text{Loc}^* \rightarrow \text{Store}_{\mathcal{E}^{da}} \rightarrow \text{Val}_{\mathcal{E}^{da}}\\
\mathcal{C}^{da}:&\quad \text{Con} \rightarrow \text{Loc}^* \rightarrow \text{Store}_{\mathcal{E}^{da}} \rightarrow \text{Val}_{\mathcal{E}^{da}}\\
alloc:&\quad ((\text{Closure} \oplus \text{Uval}) \times \text{Store}_{\mathcal{E}^{da}}) \rightarrow \text{Val}_{\mathcal{E}^{da}}\\
inc:&\quad \text{Val}_{\mathcal{E}^{da}} \rightarrow \text{Val}_{\mathcal{E}^{da}}\\
force:&\quad \text{Val}_{\mathcal{E}^{da}} \rightarrow \text{Val}_{\mathcal{E}^{da}}\\
match:&\quad (\text{Eval} \times \text{Con}) \rightarrow \text{Bool}
\end{aligned}$$

Figure 4.15: Destructive Allocation Store Semantic Functions

4.4.3 Correctness

To prove that the store semantics for programs which have been annotated for destructive allocation are correct, the following conjecture must be proved:

Conjecture 4.4.2 *The usage counting store semantics for programs which have been safely annotated for destructive allocation as defined in Definition 4.4.1 are equivalent to the usage counting store semantics for unannotated programs.*

□

Sketch Proof

The proof of this conjecture would be similar to the proof of Conjecture 4.2.2 for compile-time garbage marking. It also remains as further work.

$$
\begin{array}{l}
\mathcal{E}_p^{da}[\![e \\
\quad \textbf{where} \\
\quad f_1\ v_{11} \ldots v_{1k_1} \quad = \quad e_1 \\
\quad \vdots \\
\quad f_n\ v_{n1} \ldots v_{nk_n} \quad = \quad e_n]\!] = \mathit{force}(\mathcal{E}^{da}[\![e]\!]\ (\lambda v.\bot)\ \phi_0\ (\lambda loc.\text{UNB})) \\
\text{where} \\
\phi_0 \quad = \quad \text{fix}\ (\lambda \phi.[(\lambda loc_1 \ldots \lambda loc_{k_j}.\lambda\sigma.\mathcal{E}^{da}[\![e_j]\!]\ [loc_1/v_{j1}, \ldots, loc_{k_j}/v_{jk_j}]\ \phi\ \sigma)/f_j])
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{E}^{da}[\![k]\!]\ \rho\ \phi\ \sigma & = & \mathit{alloc}((0,k),\sigma) & \\[2ex]
\mathcal{E}^{da}[\![v]\!]\ \rho\ \phi\ \sigma & = & (loc, \sigma'[loc/\rho[\![v]\!]]), & \text{if } (\sigma\ (\rho[\![v]\!])) \in \text{Closure} \\
 & & \text{where} & \\
 & & (loc, \sigma') \ = \ (\sigma\ (\rho[\![v]\!]))\ \sigma & \\[2ex]
 & = & ((\sigma\ (\rho[\![v]\!])), \sigma), & \text{otherwise} \\[2ex]
\mathcal{E}^{da}[\![b\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \mathcal{B}^{da}[\![b]\!]\ loc_1 \ldots loc_n\ \sigma_n & \\
 & & \text{where} & \\
 & & (loc_1, \sigma_1) \ = \ \mathit{inc}(\mathcal{E}^{da}[\![e_1]\!]\ \rho\ \phi\ \sigma) & \\
 & & \vdots & \\
 & & (loc_n, \sigma_n) \ = \ \mathit{inc}(\mathcal{E}^{da}[\![e_n]\!]\ \rho\ \phi\ \sigma_{n-1}) & \\[2ex]
\mathcal{E}^{da}[\![\mathit{Cons}^v\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \mathcal{C}^{da}[\![\mathit{Cons}^v]\!]\ \rho[\![v]\!]\ loc_1 \ldots loc_n\ \sigma_n & \\
 & & \text{where} & \\
 & & (loc_1, \sigma_1) \ = \ \mathit{alloc}((\mathcal{E}^{da}[\![e_1]\!]\ \rho\ \phi), \sigma) & \\
 & & \vdots & \\
 & & (loc_n, \sigma_n) \ = \ \mathit{alloc}((\mathcal{E}^{da}[\![e_n]\!]\ \rho\ \phi), \sigma_{n-1}) & \\[2ex]
\mathcal{E}^{da}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \mathcal{C}^{da}[\![c]\!]\ loc_1 \ldots loc_n\ \sigma_n & \\
 & & \text{where} & \\
 & & (loc_1, \sigma_1) \ = \ \mathit{alloc}((\mathcal{E}^{da}[\![e_1]\!]\ \rho\ \phi), \sigma) & \\
 & & \vdots & \\
 & & (loc_n, \sigma_n) \ = \ \mathit{alloc}((\mathcal{E}^{da}[\![e_n]\!]\ \rho\ \phi), \sigma_{n-1}) & \\[2ex]
\mathcal{E}^{da}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \phi[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_n & \\
 & & \text{where} & \\
 & & (loc_1, \sigma_1) \ = \ \mathit{alloc}((\mathcal{E}^{da}[\![e_1]\!]\ \rho\ \phi), \sigma) & \\
 & & \vdots & \\
 & & (loc_n, \sigma_n) \ = \ \mathit{alloc}((\mathcal{E}^{da}[\![e_n]\!]\ \rho\ \phi), \sigma_{n-1}) &
\end{array}
$$

Figure 4.16: Destructive Allocation Store Semantics

$$
\begin{array}{lll}
\mathcal{E}^{da}[\![\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]\ \rho\ \phi\ \sigma & & \\
\quad = \mathcal{E}^{da}[\![e_i]\!]\ \rho'\ \phi\ \sigma' & & \\
\quad\quad \text{where} & & \\
\quad\quad (loc, \sigma') & = & inc(\mathcal{E}^{da}[\![e_0]\!]\ \rho\ \phi\ \sigma) \\
\quad\quad (u, x) & = & \sigma'\ loc \\
\quad\quad p_i & = & c\ v_1 \ldots v_n \text{ and } match(x, c) \\
\quad\quad \rho' & = & \rho[loc/v, x \downarrow 1/v_1, \ldots, x \downarrow n/v_n], \quad \text{if } c = Cons^v \\
 & = & \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n], \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{B}^{da}[\![+]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 + x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma\ loc_1 \\
 & & (u_2, x_2) = \sigma\ loc_2 \\
\\
\mathcal{B}^{da}[\![-]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 - x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma\ loc_1 \\
 & & (u_2, x_2) = \sigma\ loc_2 \\
\\
\mathcal{B}^{da}[\![<]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 < x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma\ loc_1 \\
 & & (u_2, x_2) = \sigma\ loc_2 \\
\\
\mathcal{B}^{da}[\![=]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0, (x_1 = x_2)), \sigma) \\
 & & \text{where} \\
 & & (u_1, x_1) = \sigma\ loc_1 \\
 & & (u_2, x_2) = \sigma\ loc_2 \\
 & \vdots & \\
\\
\mathcal{C}^{da}[\![True]\!] & = & \lambda\sigma.alloc((0,\text{TRUE}),\sigma) \\
\\
\mathcal{C}^{da}[\![False]\!] & = & \lambda\sigma.alloc((0,\text{FALSE}),\sigma) \\
\\
\mathcal{C}^{da}[\![Nil]\!] & = & \lambda\sigma.alloc((0,\text{NIL}),\sigma) \\
\\
\mathcal{C}^{da}[\![Cons]\!] & = & \lambda loc_1.\lambda loc_2.\lambda\sigma.alloc((0,(loc_1, loc_2)),\sigma) \\
\\
\mathcal{C}^{da}[\![Cons^v]\!] & = & \lambda loc_0.\lambda loc_1.\lambda loc_2.\lambda\sigma.(loc_0,\sigma[(0,(loc_1, loc_2))/loc_0])
\end{array}
$$

Figure 4.17: Destructive Allocation Store Semantics (continued)

$$
\begin{array}{lcll}
\mathit{alloc}(v,\sigma) & = & (\mathit{loc}, \sigma[v/\mathit{loc}]) & \\
 & & \text{where} & \\
 & & \sigma\ \mathit{loc} \ = \ \text{UNB} & \\
 & & & \\
\mathit{inc}(\mathit{loc},\sigma) & = & (\mathit{loc}, \sigma[(u+1,x)/\mathit{loc}]) & \\
 & & \text{where} & \\
 & & (u,x) \ = \ \sigma\ \mathit{loc} & \\
 & & & \\
\mathit{force}(\mathit{loc},\sigma) & = & (\mathit{loc},\sigma), & \text{if } (\sigma\ \mathit{loc}) = \bot \\
 & & & \text{or } (\sigma\ \mathit{loc}) = \text{UNB} \\
 & & & \\
 & = & (\mathit{loc}',\sigma'[\mathit{loc}'/\mathit{loc}]), & \text{if } (\sigma\ \mathit{loc}) \in \text{Closure} \\
 & & \text{where} & \\
 & & (\mathit{loc}',\sigma') \ = \ \mathit{force}((\sigma\ \mathit{loc})\ \sigma) & \\
 & & & \\
 & = & \mathit{force}((\sigma\ \mathit{loc}),\sigma), & \text{if } (\sigma\ \mathit{loc}) \in \text{Loc} \\
 & & & \\
 & = & \mathit{inc}(\mathit{loc},\sigma_2[(u,(\mathit{loc}_1,\mathit{loc}_2))/\mathit{loc}]), & \text{if } (\sigma\ \mathit{loc}) \in \text{Uval} \\
 & & \text{where} & \text{and } x \in \text{Conscell} \\
 & & (u,x) \ = \ \sigma\ \mathit{loc} & \\
 & & (\mathit{loc}_1,\sigma_1) \ = \ \mathit{force}(x \downarrow 1, \sigma) & \\
 & & (\mathit{loc}_2,\sigma_2) \ = \ \mathit{force}(x \downarrow 2, \sigma_1) & \\
 & & & \\
 & = & \mathit{inc}(\mathit{loc},\sigma), & \text{otherwise} \\
 & & & \\
\mathit{match}(x,c) & = & (x = \text{TRUE and } c = \mathit{True}) & \\
 & & \text{or } (x = \text{FALSE and } c = \mathit{False}) & \\
 & & \text{or } (x = \text{NIL and } c = \mathit{Nil}) & \\
 & & \text{or } (x \in \text{Conscell and } c = \mathit{Cons}) & \\
 & & \text{or } (x \in \text{Conscell and } c = \mathit{Cons}^v) &
\end{array}
$$

Figure 4.18: Destructive Allocation Store Semantics (auxiliary functions)

4.5 Related Work

4.5.1 Compile-Time Garbage Marking

Compile-time garbage marking is quite similar to the use of a one-bit reference count, as described in (Wise & Friedman, 1977). Any cells which have a reference count of one can be collected using this method, but any cells with a greater reference count cannot be collected. This is similar to the one-bit usage count which is used for marking cells in this chapter.

The method for validating compile-time garbage marking described in this thesis is similar

to that described in (Jensen & Mogensen, 1990) and (Jensen, 1990). The method described in (Jensen & Mogensen, 1990) and (Jensen, 1990) also involves marking cells at their allocation which will be used at most once. A usage counting analysis, similar to the one presented in Section 3.4, is used to determine the number of times that cells will be used. Any *Cons* applications in which the root cells of the resulting structures will be used no more than once are tagged to indicate that their root cells will become garbage after they have been used. No store semantics are defined for programs which have been annotated in this way, and the correctness of programs which have been annotated in this way is not considered.

4.5.2 Explicit Deallocation

The methods for validating explicit deallocation in a strict language described in (Inoue *et al.* , 1988) and (Hughes, 1991) both make use of information obtained by an *inheritance analysis* and a *generation* analysis. The inheritance analysis is used to determine which cells will appear directly in the result of a function, and the generation analysis is used to determine which cells are created within a function argument. Any cells generated within a function argument which are unshared and do not appear in the result of the function can be collected after evaluation of the function call. To determine whether generated cells are unshared, an *overlapping* analysis is presented in (Inoue *et al.* , 1988). In (Hughes, 1991), it is noted that cells are always shared at the same level in a list in a well-typed language. A complete level of a list which is generated can therefore be explicitly deallocated *en-masse* if it is not inherited. This method cannot be used to validate explicit deallocation in lazy languages, since some arguments which do not appear in the result of a function may not have been evaluated during the evaluation of the function. Attempting to explicitly deallocate these arguments may therefore force their evaluation, which is unsafe when using a lazy evaluation strategy. Another problem with this method of explicit deallocation is that there may be a substantial delay between a cell becoming garbage and its explicit deallocation. This is because cells are explicitly deallocated only after the evaluation of a function call. The need for run-time garbage collection will therefore not be delayed as long as possible. In the method of explicit deallocation described in this chapter, cells are explicitly deallocated immediately after becoming garbage.

An implementation of explicit deallocation in a lazy language is described in (Wakeling & Runciman, 1991). This optimisation is validated by making use of the linear type system described in (Wadler, 1990c). Values which are determined to be linear in the type system

will be used exactly once. They can therefore be explicitly deallocated immediately after they have been used. In the work described in (Wakeling & Runciman, 1991), explicit deallocation is performed in a similar manner to the way in which it is performed in this chapter. If the selector in a **case** expression is of linear type, then its root cell is explicitly deallocated immediately after it has been used. Unfortunately, it was found that very little benefit was obtained from performing explicit deallocation in this way. This was partly due to the need to maintain a free list for values which were explicitly deallocated.

An optimisation which is quite similar to explicit deallocation is the use of stack allocation. This involves determining which values appear directly in the result of a function call. Any values which do not appear in the result can be allocated on a stack, and automatically collected after evaluation of the function call. Examples of validating this kind of optimisation in a strict language are described in (Chase, 1987; Ruggieri & Murtagh, 1988; Hughes, 1988; Goldberg & Gil Park, 1990). It would be quite difficult to implement in lazy languages because values which do not appear in the result of a function call may not be evaluated until a considerable time after the evaluation of the function call. Also, it is argued in (Appel, 1987) that garbage collection can be faster than stack allocation when reasonably large stores are used.

4.5.3 Destructive Allocation

One of the earliest examples of validating destructive allocation is the method described in (Barth, 1977). This method involves performing a global flow analysis of a program which uses the run-time garbage collection method described in (Deutsch & Bobrow, 1976). Information obtained by the global flow analysis is used to avoid redundant operations for run-time garbage collection. For example, a deallocation followed by an allocation can be coalesced to give a destructive allocation instead.

An analysis for determining when destructive operators can be used without altering the meaning of strict first order programs is described in (Schwarz, 1978). These destructive operators are introduced according to the sharing properties of a program, which are given by isolation classes supplied by the user. The isolation classes given by the user are checked by ensuring that the meaning of programs are not changed by introducing destructive operators based on this information. In (Mycroft, 1981), it is shown how the isolation classes in (Schwarz, 1978) can be determined automatically. Destructive operators are then introduced based on this sharing information.

The methods for validating destructive allocation in a strict first order language which are described in (Jones & Le Métayer, 1989) and (Hamilton & Jones, 1990) both involve performing a sharing analysis to determine when cells can be deallocated. An interpreter is defined in which these unshared cells are added to a free list. The output from this interpreter is analysed to determine when destructive allocation can be performed. This will be the case when a deallocation is followed by an allocation.

A method for performing destructive allocation in a first order strict language is described in (Peterossi, 1978). This method involves reusing the arguments of basic function applications to hold the result of the application. It is not concerned with the destructive allocation of structured data.

The methods for validating destructive allocation described in (Mason, 1988) and (Hughes, 1991) involve adding destructive operators to a program, and then checking their validity. In the method for validating destructive allocation described in this chapter, programs are analysed first to indicate where destructive operators can be used, and then these destructive operators are added to the programs.

An optimisation which is quite similar to destructive allocation is the in-place update of arrays. In conventional functional implementations of arrays, the modification of an array involves making a copy of it, in case the original array is ever needed again. This problem is described in (Hudak & Bloss, 1985). A usage counting analysis could be used to determine when the in-place update of an array can be performed. Examples of analyses which can be used to determine when in-place updates of arrays can be performed are described in (Hudak, 1987; Bloss, 1989; Gopinath & Hennessy, 1989; Draghicescu & Purushothaman, 1990; Sastry *et al.* , 1993).

Another optimisation which is quite similar to destructive allocation is the globalisation of variables. This involves determining whether a value is single threaded. If this is the case, the value can be implemented globally and updated in-place each time it is modified. Examples of analyses which can be used to determine when this optimisation can be performed are described in (Schmidt, 1985; Sestoft, 1989; Gomard & Sestoft, 1991; Fradet, 1991).

The type schemes described in (Wadler, 1990c; Guzmán & Hudak, 1990; Smetsers *et al.* , 1993) and the use of monads (Wadler, 1990a) allow the user to indicate that values can always be destructively updated.

4.6 Conclusion

In this chapter, it has been shown how information obtained by usage counting analysis can be used to validate compile-time garbage collection. Three different optimisations were presented which can be viewed as different forms of compile-time garbage collection; compile-time garbage marking, explicit deallocation and destructive allocation.

Compile-time garbage marking involves marking cells at their allocation to indicate that they will become garbage after their first use. These cells can be returned to the memory manager immediately after their first use. This method has the disadvantages of requiring a run-time free list, extra space to allow for the marking of cells, and extra time to allow for the checking of cells to see if they are marked at run-time. It is therefore concluded that this form of compile-time garbage collection is probably not suitable for practical use.

Explicit deallocation involves explicitly returning cells to the memory manager at a particular point in a program. This compile-time garbage collection technique also requires the use of a free list at run-time, so the method of run-time garbage collection which must be used will not be very efficient. It is therefore concluded that this form of compile-time garbage collection is of limited use.

Destructive allocation involves reusing cells directly for further allocations within a program, thus avoiding the need for a run-time free list, so a more efficient method for run-time garbage collection can be used. It is therefore concluded that this is the only method for compile-time garbage collection which merits further consideration.

Store semantics were defined for programs which have been annotated for each of the three methods of compile-time garbage collection, and the correctness of these store semantics was considered.

It has been shown in this chapter how information obtained from usage counting analysis can be used to validate compile-time garbage collection. In Chapter 5, it is shown how information obtained from usage counting analysis can also be used to guide the transformation when compile-time garbage avoidance is performed.

Chapter 5

Compile-Time Garbage Avoidance

Compile-time garbage avoidance involves transforming programs at compile-time to reduce the amount of garbage they will produce at run-time. This optimisation attempts to overcome the problem of more readable programs being less than optimal in their use of storage. Programs which use intermediate structures are usually much easier to understand, but they are less efficient in their use of storage at run-time. To reduce the run-time costs associated with intermediate structures, a transformation algorithm called *deforestation* was proposed in (Wadler, 1990b) to eliminate them. A *treeless form* of expression is characterised in (Wadler, 1990b) which does not create any intermediate structures, and the *deforestation theorem* is given. This theorem states that the deforestation algorithm will always terminate for expressions in which all functions have definitions which are in treeless form. The sketch proof of this theorem given in (Wadler, 1990b) is fleshed out in this chapter.

The deforestation algorithm will also terminate for some expressions in which functions have definitions which are not in treeless form. The notion of an intermediate structure as described in (Wadler, 1990b) is therefore extended to that of a *transient* structure by making use of information obtained by usage counting analysis. It is shown how treeless form can be extended by making use of this definition, and that the deforestation algorithm will always terminate for expressions in which all functions have definitions which are in this extended treeless form.

Some intermediate structures can still be eliminated from expressions in which some functions have definitions which are not in extended treeless form. It is therefore shown how any function definition can be generalised in such a way that it will be in extended treeless form, and the deforestation algorithm is extended to be able to cope with these generalisations.

The remainder of this chapter is structured as follows:

- **Section 5.1**: the deforestation transformation algorithm presented in (Wadler, 1990b) is described. The treeless form of expressions defined in (Wadler, 1990b) is described, and the sketch proof given in (Wadler, 1990b) that the deforestation algorithm will always terminate for expressions in which all functions have definitions which are in this treeless form is fleshed out.

- **Section 5.2**: it is shown how treeless form can be extended by making use of information obtained by usage counting analysis. It is then proved that the deforestation algorithm will always terminate for expressions in which all functions have definitions which are in this extended treeless form.

- **Section 5.3**: it is shown how any function definition can be generalised in such a way that it is in extended treeless form. The deforestation algorithm is extended to be able to deal with these generalisations, and it is proved that this generalised deforestation algorithm will always terminate for expressions in which all functions have definitions which have been generalised in the described manner.

- **Section 5.4**: related work is considered.

- **Section 5.5**: a summary of this chapter is given.

5.1 Deforestation

In this section, the deforestation algorithm presented in (Wadler, 1990b) is described. This algorithm can be used to transform programs to eliminate intermediate structures. A form of expression, called *treeless form*, which does not create any intermediate structures is defined. The transformation rules of the deforestation algorithm are then given. In (Wadler, 1990b), a sketch proof is given that the deforestation algorithm is guaranteed to terminate for expressions in which all functions have definitions which are in treeless form. This sketch proof is fleshed out in this section. The remainder of the work this section is merely an exposition of the work given in (Wadler, 1990b).

5.1.1 Treeless Form

In (Wadler, 1990b), a treeless form of expression is characterised which creates no intermediate structures. This form of expression is defined as follows.

Definition 5.1.1 (Treeless Form) *An expression is in treeless form if it is linear*[1] *in all variables, it contains no basic function applications, every argument in a function application and every selector in a* **case** *expression is a variable, and all functions within it have treeless definitions.*

□

Expressions in treeless form must therefore satisfy the following grammar:

$$
\begin{array}{rcl}
tf & ::= & k \\
 & | & v \\
 & | & c\ tf_1 \ldots tf_n \\
 & | & f\ v_1 \ldots v_n \\
 & | & \textbf{case}\ v_0\ \textbf{of}\ p_1 : tf_1 \mid \ldots \mid p_k : tf_k
\end{array}
$$

where tf is linear in all variables, and the definition of each function f is in treeless form.

Basic function applications are not allowed in treeless expressions because they cannot be unfolded. The restriction that every argument of a function and every selector of a **case** expression must be a variable guarantees that no intermediate structures are created. The restriction that treeless expressions must be linear in all variables guarantees that certain transformations will not duplicate expressions which are expensive to compute. For example, consider a function call *square e* where *square* is a non-linear function defined as follows:

$$square\ x \quad = \quad x * x$$

If e is an expression which is expensive to compute, then the unfolded expression $e * e$ will be less efficient than the original function call *square e*. This situation will be avoided if expressions are linear in all variables.

The definition of *append* given in Figure 2.2 is in treeless form, but the definitions of *flatten*, *reverse* and *accreverse* are not because they contain function arguments which are not variables.

[1] An expression other than a **case** expression is said to be linear if no variable appears in it more than once. A **case** expression is said to be linear if no variables appear in both the selector and a branch.

5.1.2 The Deforestation Algorithm

The transformation rules of the deforestation algorithm described in (Wadler, 1990b) are shown in Figure 5.1.

$$
\begin{array}{lll}
(1) & \mathcal{T}[\![k]\!] & = \; k \\[2ex]
(2) & \mathcal{T}[\![v]\!] & = \; v \\[2ex]
(3) & \mathcal{T}[\![c\ e_1 \ldots e_n]\!] & = \; c\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!] \\[2ex]
(4) & \mathcal{T}[\![f\ e_1 \ldots e_n]\!] & = \; \mathcal{T}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!] \\
 & & \quad \text{where } f \text{ is defined by } f\ v_1 \ldots v_n = e \\[2ex]
(5) & \multicolumn{2}{l}{\mathcal{T}[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]} \\
 & & = \; \textbf{case}\ v\ \textbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!] \\[2ex]
(6) & \multicolumn{2}{l}{\mathcal{T}[\![\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]} \\
 & & = \; \mathcal{T}[\![e'_i[e_1/v_1, \ldots, e_n/v_n]]\!] \\
 & & \quad \text{where } p'_i = c\ v_1 \ldots v_n \\[2ex]
(7) & \multicolumn{2}{l}{\mathcal{T}[\![\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]} \\
 & & = \; \mathcal{T}[\![\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
 & & \quad \text{where } f \text{ is defined by } f\ v_1 \ldots v_n = e \\[2ex]
(8) & \multicolumn{2}{l}{\mathcal{T}[\![\textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_n : e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]} \\
 & & = \; \mathcal{T}[\![\textbf{case}\ e_0\ \textbf{of} \\
 & & \qquad p_1 \;:\; \textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\
 & & \qquad \vdots \\
 & & \qquad p_n \;:\; \textbf{case}\ e_n\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]
\end{array}
$$

Figure 5.1: Transformation Rules for Deforestation

A valid input to the deforestation algorithm is a linear expression in which there are no

basic function applications, and all functions have treeless definitions. The output from the algorithm will be an equivalent expression which is in treeless form.

The transformation rules given in Figure 5.1 cover all possible expressions which can be encountered by the deforestation algorithm. Basic function applications will not be encountered by the algorithm since they cannot be present in its input, or in any treeless function definition. All four possibilities for the selector of a **case** expression are considered in rules (5) to (8). The selector of a **case** expression cannot be a constant since pattern matching is not performed on integers.

Rule (8) is valid only if there is no name clash between the variables in the patterns $p_1 \ldots p_n$, and the free variables in the branches $p'_1 : e'_1 \ldots p'_k : e'_k$. It is always possible to rename the variables in the patterns $p_1 \ldots p_n$ so that this condition applies.

As they stand, these transformation rules will not necessarily terminate. For example, consider the deforestation of the expression *append* (*append xs ys*) *zs* shown in Figure 5.2.

```
T[[append (append xs ys) zs]]

  =  T[[case (append xs ys) of                                          (By 4)
            Nil       :  zs
            Cons x xs :  Cons x (append xs zs)]]
  =  T[[case (case xs of                                                (By 7)
                  Nil       :  ys
                  Cons x xs :  Cons x (append xs ys)) of
            Nil       :  zs
            Cons x xs :  Cons x (append xs zs)]]
  =  case xs of                                                   (By 8,5,5,2,6)
       Nil        :  case ys of
                          Nil       :  zs
                          Cons x xs :  T[[Cons x (append xs zs)]]
       Cons x xs  :  T[[Cons x (append (append xs ys) zs)]]
  =  case xs of                                                   (By 3,2,4,3,2)
       Nil        :  case ys of
                          Nil       :  zs
                          Cons x xs :
                               Cons x (T[[case xs of
                                              Nil       :  zs
                                              Cons x xs :  Cons x (append xs zs)]])
       Cons x xs  :  Cons x (T[[append (append xs ys) zs]])
```

Figure 5.2: Deforestation of *append* (*append xs ys*) *zs*

The transformation rules are applied until an expression is obtained which is a renaming of a previously encountered expression. If the transformation were to continue, the rules would be applied without end. This non-termination can be avoided by introducing appropriate new function definitions. For the given example, the following function definitions need to be introduced:

$$f\ xs\ ys\ zs \quad = \quad \mathcal{T}[\![append\ (append\ xs\ ys)\ zs]\!].$$

$$\begin{array}{lcl} f'\ xs\ ys & = & \mathcal{T}[\![\textbf{case}\ xs\ \textbf{of} \\ & & \quad Nil \quad : \quad ys \\ & & \quad Cons\ x\ xs \quad : \quad Cons\ x\ (append\ xs\ ys)]\!] \end{array}$$

Expressions which match the right hand side of one of these definitions (modulo renaming of variables) are replaced by an appropriate call of the corresponding function, resulting in the expression shown in Figure 5.3.

$$\begin{array}{lcl} f\ xs\ ys\ zs & & \\ \textbf{where} & & \\ f\ xs\ ys\ zs & = & \textbf{case}\ xs\ \textbf{of} \\ & & \quad Nil \quad : \quad f'\ ys\ zs \\ & & \quad Cons\ x\ xs \quad : \quad Cons\ x\ (f\ xs\ ys\ zs) \\ & & \\ f'\ xs\ ys & = & \textbf{case}\ xs\ \textbf{of} \\ & & \quad Nil \quad : \quad ys \\ & & \quad Cons\ x\ xs \quad : \quad Cons\ x\ (f'\ xs\ ys) \end{array}$$

Figure 5.3: Result of Deforestation of *append* (*append xs ys*) *zs*

It remains to be shown when these new function definitions should be introduced. Any infinite sequence of transformation steps must involve applications of rules (4) or (7) in which function calls are unfolded. A new function definition is therefore introduced before the application of each of these rules. The right hand sides of these function definitions are the expressions which were about to be transformed by rules (4) and (7). When an expression is encountered later in the transformation which matches the right hand side of one of these function definitions (modulo renaming of variables), it is replaced by an appropriate call of the corresponding function. Transformation rules (4) and (7) must therefore be changed to make this more explicit.

These modified rules are shown in Figure 5.4.

$$
\begin{array}{lll}
(4) & \mathcal{T}[\![f\ e_1 \ldots e_n]\!]\ \phi & \\
 & = f'\ v_1 \ldots v_j, & \text{if } (f'\ v'_1 \ldots v'_j = e) \in \phi \\
 & & \text{and } f\ e_1 \ldots e_n = e[v_1/v'_1, \ldots, v_j/v'_j] \\
 & \quad \text{where} & \\
 & \quad v_1 \ldots v_j \text{ are the free variables in } (f\ e_1 \ldots e_n) & \\
 & = f''\ v_1 \ldots v_j, & \text{otherwise} \\
 & \quad \text{where} & \\
 & \quad f''\ v_1 \ldots v_j = \mathcal{T}[\![e[e_1/v'_1, \ldots, e_n/v'_n]]\!]\ \phi' & \\
 & \qquad\qquad \text{where } f \text{ is defined by } f\ v'_1 \ldots v'_n = e & \\
 & \quad \phi' = \phi\ \cup\ \{f''\ v_1 \ldots v_j = f\ e_1 \ldots e_n\} & \\
 & \quad v_1 \ldots v_j \text{ are the free variables in } (f\ e_1 \ldots e_n) &
\end{array}
$$

$$
\begin{array}{lll}
(7) & \mathcal{T}[\![\mathbf{case}\ (f\ e_1 \ldots e_n)\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \phi & \\
 & = f'\ v_1 \ldots v_j, & \text{if } (f'\ v'_1 \ldots v'_j = e) \in \phi \\
 & & \text{and } \mathbf{case}\ (f\ e_1 \ldots e_n)\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k = e[v_1/v'_1, \ldots, v_j/v'_j] \\
 & \quad \text{where} & \\
 & \quad v_1 \ldots v_j \text{ are the free variables in } (\mathbf{case}\ (f\ e_1 \ldots e_n)\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) & \\
 & = f''\ v_1 \ldots v_j, & \text{otherwise} \\
 & \quad \text{where} & \\
 & \quad f''\ v_1 \ldots v_j = \mathcal{T}[\![\mathbf{case}\ (e[e_1/v'_1, \ldots, e_n/v'_n])\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \phi' & \\
 & \qquad\qquad \text{where } f \text{ is defined by } f\ v'_1 \ldots v'_n = e & \\
 & \quad \phi' = \phi\ \cup\ \{f''\ v_1 \ldots v_j = \mathbf{case}\ (f\ e_1 \ldots e_n)\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k\} & \\
 & \quad v_1 \ldots v_j \text{ are the free variables in } (\mathbf{case}\ (f\ e_1 \ldots e_n)\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) &
\end{array}
$$

Figure 5.4: Modified Transformation Rules for Deforestation

In these rules, the additional parameter ϕ contains the set of function definitions which have been created during the transformation so far.

5.1.3 The Deforestation Theorem

The main result of the deforestation transformation presented in (Wadler, 1990b) is the *deforestation theorem.*

Theorem 5.1.2 (Deforestation Theorem) *Every expression which is linear in all variables, contains no basic function applications, and in which all functions have treeless definitions, will be transformed by the deforestation algorithm to an equivalent treeless expression, without loss of efficiency.*

□

Proof

The deforestation theorem can be proved by showing the following four lemmata, which together demonstrate the validity of the theorem.

□

Lemma 5.1.3 *Every expression which contains no basic function applications will be transformed to an equivalent expression if the deforestation algorithm terminates.*

□

Lemma 5.1.4 *Every expression which contains no basic function applications will be transformed to a treeless expression if the deforestation algorithm terminates.*

□

Lemma 5.1.5 *Every expression which is linear in all variables, contains no basic function applications, and in which all functions have treeless definitions, will be transformed without loss of efficiency if the deforestation algorithm terminates.*

□

Lemma 5.1.6 *The deforestation algorithm will always terminate for every expression which contains no basic function applications and in which all functions have treeless definitions.*

□

Proof of Lemma 5.1.3

The proof of this lemma can be found in Appendix C.1.1.

□

Proof of Lemma 5.1.4

The proof of this lemma can be found in Appendix C.1.2.

□

Proof of Lemma 5.1.5

The proof of this lemma can be found in Appendix C.1.3.

□

Proof of Lemma 5.1.6

As described in (Wadler, 1990b), to prove that the deforestation algorithm always terminates, it is sufficient to show that there is a bound on the size of the expressions encountered during transformation. If there is such a bound, then there will be a finite number of expressions encountered (modulo renaming of variables), and a renaming of a previous expression must eventually be encountered. The algorithm will therefore be guaranteed to terminate. A sketch proof of this is given in (Wadler, 1990b). This proof is fleshed out here. First of all, it is shown that any expression encountered by the deforestation algorithm must always satisfy a particular grammatical form. It is then shown that there is a bound on the size of expressions described by this grammar.

Definition 5.1.7 (Size of Expressions) *The size of an expression is given by $\mathcal{S}$, as defined in Figure 5.5.*

□

$$
\begin{array}{lcl}
\mathcal{S}[\![k]\!] & = & 0 \\
\mathcal{S}[\![v]\!] & = & 0 \\
\mathcal{S}[\![b\ e_1 \ldots e_n]\!] & = & 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
\mathcal{S}[\![c\ e_1 \ldots e_n]\!] & = & 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
\mathcal{S}[\![f\ e_1 \ldots e_n]\!] & = & 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
\mathcal{S}[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!] & = & 1 + max(\mathcal{S}[\![e_0]\!], \ldots, \mathcal{S}[\![e_k]\!])
\end{array}
$$

Figure 5.5: Definition of the Size of Expressions

This definition corresponds to the definition of the *depth* of an expression given in (Wadler, 1990b).

Definition 5.1.8 (Maximum Size of Function Definitions in a Program)
For a given program in which the right hand sides of function definitions are $e_1 \ldots e_n$, the maximum size of the function definitions is defined as follows:

$$s = max(1, \mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!])$$

□

Definition 5.1.9 (Grammar of Expressions Encountered During Deforestation)
The grammar of expressions encountered during deforestation is given by $dg^s(s,n)$, as described in Figure 5.6 for a suitable value of n where s is the maximum size of any function definitions accessible within the expression.

□

In the definition of this grammar, fv represents any free variable in the expression which is described by the grammar. All treeless function definitions are described by the grammar $dg^s(s,1)$ since the size of all function definitions is bounded by s. The expression to be transformed must be described by the grammar $dg^s(s,n)$ for a suitable value of n. The value of s may need to be changed to satisfy this criterion, but no loss of generality results. The value of n corresponds to the *order* of an expression, as described in (Wadler, 1990b).

$$
\begin{array}{rcll}
dg^s(x,y) & ::= & k & \text{if } x \geq 0 \text{ and } y > 0 \\
 & | & v & \text{if } x \geq 0 \text{ and } y > 0 \\
 & | & c\ dg_1^s(x-1,y)\ldots dg_n^s(x-1,y) & \text{if } x > 0 \text{ and } y > 0 \\
 & | & f\ dg_1^s(0,y)\ldots dg_n^s(0,y) & \text{if } x > 0 \text{ and } y > 0 \\
 & & \multicolumn{2}{l}{\text{where } f \text{ is defined by } f\ v_1\ldots v_n = e \text{ and } e \in dg^s(s,1)} \\
 & | & \multicolumn{2}{l}{\textbf{case}\ dg_0^s(0,y)\ \textbf{of}\ p_1 : dg_1^s(x-y,y) \mid \ldots \mid p_k : dg_k^s(x-y,y)} \\
 & & & \text{if } x > 0 \text{ and } y > 0 \\
 & | & dg^s(x-1,y) & \text{if } x > 0 \text{ and } y > 0 \\
 & | & fg^s(s,y-1) & \text{if } x \geq 0 \text{ and } y > 1 \\
 & & & \\
fg^s(x,y) & ::= & k & \text{if } x \geq 0 \text{ and } y > 0 \\
 & | & fv & \text{if } x \geq 0 \text{ and } y > 0 \\
 & | & c\ fg_1^s(x-1,y)\ldots fg_n^s(x-1,y) & \text{if } x > 0 \text{ and } y > 0 \\
 & | & f\ fg_1^s(0,y)\ldots fg_n^s(0,y) & \text{if } x > 0 \text{ and } y > 0 \\
 & & \multicolumn{2}{l}{\text{where } f \text{ is defined by } f\ v_1\ldots v_n = e \text{ and } e \in dg^s(s,1)} \\
 & | & \multicolumn{2}{l}{\textbf{case}\ fg_0^s(0,y)\ \textbf{of}\ p_1 : fg_1^s(x-y,y) \mid \ldots \mid p_k : fg_k^s(x-y,y)} \\
 & & & \text{if } x > 0 \text{ and } y > 0 \\
 & | & fg^s(x-1,y) & \text{if } x > 0 \text{ and } y > 0 \\
 & | & fg^s(s,y-1) & \text{if } x \geq 0 \text{ and } y > 1
\end{array}
$$

Figure 5.6: Grammar of Expressions Encountered During Deforestation

If an expression is described by the grammar $dg^s(x,y)$ where $x \leq s$ and $y \leq n$, then the expression is also described by the grammar $dg^s(s,n)$.

Lemma 5.1.6 can now be proved by showing the following two lemmata.

Lemma 5.1.10 *All expressions encountered by the deforestation algorithm are described by the grammar $dg^s(s,n)$ if the original expression to be transformed is also described by the grammar $dg^s(s,n)$ for a suitable value of n, where s is as defined in Definition 5.1.8.*

□

Lemma 5.1.11 *The size of all expressions described by the grammar $dg^s(s,n)$ is bounded by $s \times n$.*

□

Proof of Lemma 5.1.10

The proof of this lemma can be found in Appendix C.1.4.

□

Proof of Lemma 5.1.11

The proof of this lemma can be found in Appendix C.1.5.

□

5.2 Extended Deforestation

The deforestation algorithm is guaranteed to terminate for expressions in which all functions have definitions which are in treeless form. It may, however, also terminate for expressions in which some functions have definitions which are not in treeless form. For example, the definition of the function *flatten* given in Figure 2.2 is not in treeless form, but expressions involving calls of this function can be successfully transformed by the deforestation algorithm.

In this section, it is shown how the definition of treeless form (Definition 5.1.1) can be extended by making use of information obtained by usage counting analysis. It is then proved that the deforestation algorithm is guaranteed to terminate for expressions in which all functions have definitions which are in this extended treeless form.

5.2.1 Transient Structures

In the definition of treeless form (Definition 5.1.1), an intermediate structure is assumed to be a function argument or **case** selector, and these are restricted to being variables. However, some function arguments may appear directly in the result of the function. These function arguments can be treated in the same way as the arguments in constructor applications. The notion of an intermediate structure is therefore extended to that of a *transient* structure, which is defined as follows.

Definition 5.2.1 (Transient Structure) *A structure is transient within an expression if it is used as the selector in a* **case** *expression during the evaluation of the expression to normal form.*

□

```
T[[append (flatten xss) ys]]

  = T[[case (flatten xss) of                                               (By 4)
         Nil        :  ys
         Cons x xs  :  Cons x (append xs ys)]]
  = T[[case (case xss of                                                   (By 7)
                 Nil          :  Nil
                 Cons xs xss  :  append xs (flatten xss)) of
         Nil        :  ys
         Cons x xs  :  Cons x (append xs ys)]]
  = case xss of                                                     (By 8,5,6,2)
      Nil          :  ys
      Cons xs xss  :  T[[case (append xs (flatten xss)) of
                             Nil        :  ys
                             Cons x xs  :  Cons x (append xs ys)]]
  = case xss of                                                            (By 7)
      Nil          :  ys
      Cons xs xss  :
            T[[case (case xs of
                            Nil        :  flatten xss
                            Cons x xs  :  Cons x (append xs (flatten xss))) of
                   Nil        :  ys
                   Cons x xs  :  Cons x (append xs ys)]]
  = case xss of                                                 (By 8,5,6,3,2,4)
      Nil          :  ys
      Cons xs xss  :
            case xs of
              Nil        :  T[[case (flatten xss) of
                                   Nil        :  ys
                                   Cons x xs  :  Cons x (append xs ys)]]

              Cons x xs  :  Cons x (T[[case (append xs (flatten xss)) of
                                             Nil        :  ys
                                             Cons x xs  :  Cons x (append xs ys)]])
```

Figure 5.7: Deforestation of *append* (*flatten xss*)

This information can be determined by usage counting analysis. If some parts of a structure are used during the evaluation of an expression, then the structure is transient within the expression. A variable v of atomic type is a transient structure within the expression e if the following condition holds:

$$1 \sqsubseteq_{T_A} \mathcal{U}[\![e]\!][\![v]\!]\ (0,\ldots,0)\ \phi_{\mathcal{U}}$$

A variable v of structured type T is a transient structure within an expression e if the following condition holds:

$$(1, ABS) \sqsubseteq_{T} \mathcal{U}[\![e]\!][\![v]\!]\ (0,\ldots,0)\ \phi_{\mathcal{U}}$$

As described in Section 3.2, the usage pattern $(0,\ldots,0)$ represents a context in which an expression is evaluated to normal form, but is not used in any further computations. Thus the only usage of a structure within the expression must be as the selector in a **case** expression.

In order to determine the transient structures within a function definition, the context of each expression is determined from an initial top-level context for the function indicating that its result will not be used. For example, consider the definition of the function *flatten* given in Figure 2.2. All transient structures within this definition are variables. The second argument in the call of the function *append* within this definition is not a variable, but it is not a transient structure (see Table 3.1). The deforestation algorithm can be successfully applied to expressions containing calls of the function *flatten*. For example, the deforestation of the expression *append* (*flatten xss*) *ys* is shown in Figure 5.7. The result of this transformation is the expression shown in Figure 5.8.

```
f xss ys
where
f xss ys      = case xss of
                  Nil           : ys
                  Cons xs xss   : f' xs xss ys

f' xs xss ys  = case xs of
                  Nil         : f xss ys
                  Cons x xs   : Cons x (f' xs xss ys)
```

Figure 5.8: Result of Deforestation of *append* (*flatten xss*) *ys*

5.2.2 Accumulating Parameters

The deforestation algorithm will not terminate for some expressions in which all transient structures are variables. For example, all transient structures in the definition of the function *accreverse* given in Figure 2.2 are variables. The deforestation of the expression *accreverse xs ys* is shown in Figure 5.9.

$$
\begin{array}{ll}
\mathcal{T}[\![accreverse\ xs\ ys]\!] & \\
\\
= \mathcal{T}[\![\textbf{case}\ xs\ \textbf{of} & \text{(By 4)} \\
\quad\quad Nil \quad : \ ys & \\
\quad\quad Cons\ x\ xs \ :\ accreverse\ xs\ (Cons\ x\ ys)]\!] & \\
= \textbf{case}\ xs\ \textbf{of} & \text{(By 5,2)} \\
\quad Nil \quad : \ ys & \\
\quad Cons\ x\ xs \ :\ \mathcal{T}[\![accreverse\ xs\ (Cons\ x\ ys)]\!] & \\
= \textbf{case}\ xs\ \textbf{of} & \text{(By 4)} \\
\quad Nil \quad : \ ys & \\
\quad Cons\ x\ xs \ :\ \mathcal{T}[\![\textbf{case}\ xs\ \textbf{of} & \\
\quad\quad\quad\quad Nil \quad : \ Cons\ x\ ys & \\
\quad\quad\quad\quad Cons\ x'\ xs' \ :\ accreverse\ xs'\ (Cons\ x'\ (Cons\ x\ ys))]\!] & \\
= \textbf{case}\ xs\ \textbf{of} & \text{(By 5,3,2,2)} \\
\quad Nil \quad : \ ys & \\
\quad Cons\ x\ xs \ :\ \textbf{case}\ xs\ \textbf{of} & \\
\quad\quad\quad\quad Nil \quad : \ Cons\ x\ ys & \\
\quad\quad\quad\quad Cons\ x'\ xs' \ :\ \mathcal{T}[\![accreverse\ xs'\ (Cons\ x'\ (Cons\ x\ ys))]\!] & \\
\\
\vdots &
\end{array}
$$

Figure 5.9: Deforestation of *accreverse xs ys*

The size of the second parameter in the recursive call of *accreverse* continually increases during the transformation, so the transformation fails to terminate. This situation occurs when a recursive function accumulates information in its parameters. A recursive function call is defined as follows[2].

Definition 5.2.2 (Recursive Function Call) *A function call is recursive if it occurs within the definition of a function which the recursive function calls (either directly or indirectly).*

□

[2]This definition also defines mutually recursive function calls.

Accumulating parameters can now be defined as follows.

Definition 5.2.3 (Accumulating Parameter) *An argument in a recursive function call is an accumulating parameter if it is not a variable.*

□

5.2.3 Shared Values

In the deforestation theorem, the expressions to be transformed are required to be linear in all variables. This is to avoid the duplication of expressions which may be expensive to compute, so that they will not need to be evaluated more than once. It may be the case that a duplicated expression will not be evaluated more than once. For example, consider the following function definitions:

$$f\,x \quad = \quad \mathbf{K}\ x\ x$$

$$\mathbf{K}\ x\ y \quad = \quad x$$

The definition of the function f is not in treeless form because it is not linear in the variable x. However, the expression represented by the variable x will be used only once, so there is no reason why it should not be involved in transformations using the deforestation algorithm. Values will be duplicated by the deforestation algorithm only if they are shared. Shared values are defined as follows.

Definition 5.2.4 (Shared Value) *A value is shared if it is used more than once.*

□

This information can be determined by usage counting analysis. A variable v of atomic type is a shared value within the expression e if the following condition holds:

$$2 \sqsubseteq_{T_A} \mathcal{U}[\![e]\!][\![v]\!]\ (1, \ldots, 1)\ \phi_{\mathcal{U}}$$

A value of structured type is a shared value within the expression e if the following condition holds:

$$(2,\text{ABS}) \sqsubseteq_T \mathcal{U}[\![e]\!][\![v]\!]\ (1,\ldots,1)\ \phi_{\mathcal{U}}$$

In order to determine the shared values within a function definition, the context of each expression is determined from an initial top-level context for the function indicating that its result will be used exactly once.

5.2.4 Extended Treeless Form

Extended treeless form can now be defined as follows.

Definition 5.2.5 (Extended Treeless Form) *An expression is in extended treeless form if it contains no basic function applications, accumulating parameters or shared values, all transient structures within it are variables, and all functions within it have extended treeless definitions.*

□

As for treeless form, basic function applications are not allowed in extended treeless expressions because they cannot be unfolded. The definitions of the functions *append* and *flatten* given in Figure 2.2 are in extended treeless form, but the definitions of the functions *reverse* and *accreverse* are not.

A valid input to the deforestation algorithm is an expression in which there are no shared values or basic function applications, and all functions have extended treeless definitions. The output from the algorithm will be an equivalent treeless expression and a collection of treeless function definitions.

5.2.5 The Extended Deforestation Theorem

The deforestation theorem can now be extended to the *extended deforestation theorem.*

Theorem 5.2.6 (Extended Deforestation Theorem) *Every expression which contains no shared values or basic function applications, and in which all functions have extended treeless definitions, will be transformed to an equivalent treeless expression by the deforestation algorithm, without loss of efficiency.*

□

Proof

The proof of the extended deforestation theorem is very similar to the proof of the deforestation theorem. It can be proved by showing the following four lemmata, which together demonstrate the validity of the theorem.

□

Lemma 5.2.7 *Every expression which contains no basic function applications will be transformed to an equivalent expression if the deforestation algorithm terminates.*

□

Lemma 5.2.8 *Every expression which contains no basic function applications will be transformed to a treeless expression if the deforestation algorithm terminates.*

□

Lemma 5.2.9 *Every expression which contains no shared values or basic function applications, and in which all functions have extended treeless definitions, will be transformed without loss of efficiency if the deforestation algorithm terminates.*

□

Lemma 5.2.10 *The deforestation algorithm will always terminate for every expression which contains no basic function applications and in which all functions have extended treeless definitions.*

□

Proof of Lemma 5.2.7

This lemma is identical to Lemma 5.1.3, the proof of which is given in Appendix C.1.1.

□

Proof of Lemma 5.2.8

This lemma is identical to Lemma 5.1.4, the proof of which is given in Appendix C.1.2.

□

Proof of Lemma 5.2.9

The proof of this lemma is very similar to the proof of Lemma 5.1.5, which is given in Appendix C.1.3.

□

Proof of Lemma 5.2.10

As for the proof of Lemma 5.1.6, to prove that the deforestation algorithm always terminates, it is sufficient to show that there is a bound on the size of expressions encountered during deforestation. It is therefore shown that expressions which are encountered by the deforestation algorithm are always described by the grammar $edg^{s,n}(s, f, n)$ for a suitable value of n, where f is the number of function definitions in the overall program, and s is the maximum size of the right hand side of any function definition (Definition 5.1.8). It is then shown that there is a bound on the size of expressions described by the grammar $edg^{s,n}(s, f, n)$.

□

Definition 5.2.11 (Grammar of Expressions Encountered During Extended Deforestation) *The grammar of expressions encountered during extended deforestation is described by* $edg^{s,n}(s, f, n)$*, as defined in Figure 5.10 for a suitable value of* n *where* s *is the maximum size of any function definitions accessible from within the expression, and* f *is the maximum number of functions accessible from within the expression..*

□

$$
\begin{array}{lll}
\multicolumn{3}{l}{edg^{s,n}(x,y,z) ::=} \\
 & k & \text{if } x \geq 0,\ y \geq 0 \text{ and } z > 0 \\
| & v & \text{if } x \geq 0,\ y \geq 0 \text{ and } z > 0 \\
| & c\ edg_1^{s,n}(x-1,y,z) \ldots edg_n^{s,n}(x-1,y,z) & \\
 & & \text{if } x > 0,\ y \geq 0 \text{ and } z > 0 \\
| & f\ e_1 \ldots e_n & \text{if } x > 0,\ 0 \leq y < f \text{ and } z > 0 \\
 & \multicolumn{2}{l}{\text{where } f \text{ is defined by } f\ v_1 \ldots v_n = e \text{ and } e \in edg^{s,n}(s,0,1)} \\
 & \multicolumn{2}{l}{\text{and } e_i \ \in edg^{s,n}(0,0,z), \quad \text{if } e_i \text{ is a transient structure}} \\
 & \multicolumn{2}{l}{\qquad\quad \in edg^{s,n}(x-1,y,z), \quad \text{otherwise}} \\
| & f\ v_1 \ldots v_n & \text{if } x > 0,\ y = f \text{ and } z > 0 \\
 & \multicolumn{2}{l}{\text{where } f \text{ is defined by } f\ v_1' \ldots v_n' = e \text{ and } e \in edg^{s,n}(s,0,1)} \\
| & \multicolumn{2}{l}{\textbf{case } edg_0^{s,n}(0,0,z) \textbf{ of } p_1 : edg_1^{s,n}(x-z,y,z) \mid \ldots \mid p_k : edg_k^{s,n}(x-z,y,z)} \\
 & & \text{if } x > 0,\ y \geq 0 \text{ and } z > 0 \\
| & edg^{s,n}(x-1,y,z) & \text{if } x > 0,\ y \geq 0 \text{ and } z > 0 \\
| & edg^{s,n}(s,y-1,z) & \text{if } x \geq 0,\ y > 0 \text{ and } z > 0 \\
| & efg^{s,n}(s,f,z-1) & \text{if } x \geq 0,\ y \geq 0 \text{ and } z > 1
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{efg^{s,n}(x,y,z) ::=} \\
 & k & \text{if } x \geq 0,\ y \geq 0 \text{ and } z > 0 \\
| & fv & \text{if } x \geq 0,\ y \geq 0 \text{ and } z > 0 \\
| & c\ efg_1^{s,n}(x-1,y,z) \ldots efg_n^{s,n}(x-1,y,z) & \\
 & & \text{if } x > 0,\ y \geq 0 \text{ and } z > 0 \\
| & f\ e_1 \ldots e_n & \text{if } x > 0,\ 0 \leq y < f \text{ and } z > 0 \\
 & \multicolumn{2}{l}{\text{where } f \text{ is defined by } f\ v_1 \ldots v_n = e \text{ and } e \in efg^{s,n}(s,0,1)} \\
 & \multicolumn{2}{l}{\text{and } e_i \ \in efg^{s,n}(0,0,z), \quad \text{if } e_i \text{ is a transient structure}} \\
 & \multicolumn{2}{l}{\qquad\quad \in efg^{s,n}(x-1,y,z), \quad \text{otherwise}} \\
| & f\ v_1 \ldots v_n & \text{if } x > 0,\ y = f \text{ and } z > 0 \\
 & \multicolumn{2}{l}{\text{where } f \text{ is defined by } f\ v_1' \ldots v_n' = e \text{ and } e \in efg^{s,n}(s,0,1)} \\
| & \multicolumn{2}{l}{\textbf{case } efg_0^{s,n}(0,0,z) \textbf{ of } p_1 : efg_1^{s,n}(x-z,y,z) \mid \ldots \mid p_k : efg_k^{s,n}(x-z,y,z)} \\
 & & \text{if } x > 0,\ y \geq 0 \text{ and } z > 0 \\
| & efg^{s,n}(x-1,y,z) & \text{if } x > 0,\ y \geq 0 \text{ and } z > 0 \\
| & efg^{s,n}(s,y-1,z) & \text{if } x \geq 0,\ y > 0 \text{ and } z > 0 \\
| & efg^{s,n}(s,f,z-1) & \text{if } x \geq 0,\ y \geq 0 \text{ and } z > 1
\end{array}
$$

Figure 5.10: Grammar of Expressions Encountered During Extended Deforestation

In the definition of the grammar $efg^{s,n}(x,y,z)$, the value of y represents the number of different functions which have been unfolded to produce the current expression. If the value of y is equal to f, all function calls within the current expression must be recursive (Definition 5.2.2), and can have only variables as arguments.

All extended treeless function definitions are described by the grammar $efg^{s,n}(s,0,1)$, since the size of all function definitions is bounded by s. The expression to be transformed

must be described by the grammar $efg^{s,n}(s, f, n)$ for a suitable value of n. The value of s may need to be changed to satisfy this criterion, but no loss of generality results. If an expression is described by the grammar $edg^{s,n}(x, y, z)$, where $x \leq s$, $y \leq f$ and $z \leq n$, then the expression is also described by the grammar $edg^{s,n}(s, f, n)$.

Lemma 5.2.10 can now be proved by showing the following two lemmata.

Lemma 5.2.12 *All expressions encountered by the deforestation algorithm are described by the grammar $edg^{s,n}(s, f, n)$ if the original expression to be transformed is also described by the grammar $edg^{s,n}(s, f, n)$.*

□

Lemma 5.2.13 *The size of all expressions described by the grammar $edg^{s,n}(s, f, n)$ is bounded by $s \times (f + 1) \times n$.*

□

Proof of Lemma 5.2.12

The proof of this lemma can be found in Appendix C.2.1.

□

Proof of Lemma 5.2.13

The proof of this lemma can be found in Appendix C.2.2.

□

5.3 Generalised Deforestation

The deforestation algorithm is guaranteed to terminate for expressions in which all functions have definitions which are in extended treeless form. It may, however, be possible to eliminate intermediate structures from an expression in which some functions have definitions which are not in extended treeless form. For example, in the expression *accreverse* (*flatten xss*) *ys*, it is

possible to eliminate the intermediate list created as the result of the function call (*flatten xss*), even though the definition of the function *accreverse* given in Figure 2.2 is not in extended treeless form.

In this section, it is shown how expressions can be generalised to extended treeless form. The deforestation algorithm is then extended to be able to cope with these generalisations. It is then proved that this generalised deforestation algorithm is guaranteed to terminate for expressions in which all functions have definitions which are in this generalised treeless form.

5.3.1 Generalised Treeless Form

If all function definitions could be generalised in such a way that they are in extended treeless form, then the deforestation algorithm would be guaranteed to terminate for all expressions in the language. An expression is not in extended treeless form if it contains accumulating parameters, shared values or transient structures which are not variables. Accumulating parameters, shared values and transient structures which are not variables are therefore extracted so that they can be transformed independently, as is done in the blazed deforestation algorithm described in (Wadler, 1990b) for values of atomic type. To represent the result of these extractions, **let** expressions of the following form are introduced:

$$\textbf{let } v = e_0 \textbf{ in } e_1$$

Generalised treeless form can now be defined as follows.

Definition 5.3.1 (Generalised Treeless Form) *An expression is in generalised treeless form if all accumulating parameters, shared values and transient structures which are not variables have been extracted from it using* **let** *expressions, and all functions within it have generalised treeless definitions.*

□

For example, the *accreverse* function defined in Figure 2.2 is not in extended treeless form because there is an accumulating parameter in its recursive call. This accumulating parameter can be extracted to give the following generalised treeless definition:

$$\begin{array}{lcl} \mathit{accreverse}\ \mathit{xs}\ \mathit{ys} & = & \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & & \quad \begin{array}{lcl} \mathit{Nil} & : & \mathit{ys} \\ \mathit{Cons}\ x\ \mathit{xs} & : & \textbf{let}\ v = \mathit{Cons}\ x\ \mathit{ys} \\ & & \textbf{in}\ \ \mathit{accreverse}\ \mathit{xs}\ v \end{array} \end{array}$$

5.3.2 The Generalised Deforestation Algorithm

The four additional transformation rules shown in Figure 5.11 must be added to the deforestation algorithm to cope with the described generalisations.

$$
\begin{array}{lll}
(9) & \mathcal{T}[\![b\ e_1 \ldots e_n]\!] & = \ b\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!] \\
\\
(10) & \mathcal{T}[\![\mathbf{case}\ (b\ e_1 \ldots e_n)\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] & \\
 & & = \ \mathbf{case}\ (b\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!])\ \mathbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!] \\
\\
(11) & \mathcal{T}[\![\mathbf{let}\ v = e_0\ \mathbf{in}\ e_1]\!] & \\
 & & = \ \mathbf{let}\ v = \mathcal{T}[\![e_0]\!]\ \mathbf{in}\ \mathcal{T}[\![e_1]\!] \\
\\
(12) & \mathcal{T}[\![\mathbf{case}\ (\mathbf{let}\ v = e_0\ \mathbf{in}\ e_1)\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] & \\
 & & = \ \mathbf{let}\ v = \mathcal{T}[\![e_0]\!]\ \mathbf{in}\ \mathcal{T}[\![\mathbf{case}\ e_1\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]
\end{array}
$$

Figure 5.11: Additional Transformation Rules for the Generalised Deforestation Algorithm

Rules (9) and (10) cover the application of basic functions. Basic function applications are not allowed in the input to the deforestation algorithm since they cannot be unfolded. They are handled by the generalised deforestation algorithm by recursively transforming their arguments. Rules (11) and (12) deal with the transformation of **let** expressions. Rule (12) is valid only if the variable v does not occur free in any of the branches of the **case** expression. It is always possible to rename this variable so that this condition applies.

A valid input to the generalised deforestation algorithm is an expression in which all shared values have been extracted, and all functions have generalised treeless definitions. The output from the generalised deforestation algorithm will be an equivalent expression from which intermediate structures have been removed. After the transformation is complete, all expressions of the form **let** $v = e_0$ **in** e_1 may be removed in the same manner as described in (Wadler, 1990b). If the variable v is used at most once in the expression e_1, then the **let** expression may be replaced by $e_1[e_0/v]$. Otherwise, a new function f defined by $f\ v = e_1$ is introduced, and the **let** expression can be replaced by $f\ e_0$. Alternatively, the language could be extended to include **let** expressions. The standard semantics for such a **let** expression would be defined as follows:

$$\mathcal{T}[\![\mathit{accreverse}\ (\mathit{flatten}\ \mathit{xss})\ \mathit{ys}]\!]$$

$$
\begin{array}{lll}
= & \mathcal{T}[\![\mathbf{case}\ (\mathit{flatten}\ \mathit{xss})\ \mathbf{of} & \text{(By 4)}\\
& \quad \begin{array}{lll} \mathit{Nil} & : & \mathit{ys}\\ \mathit{Cons}\ x\ \mathit{xs} & : & \mathbf{let}\ v = \mathit{Cons}\ x\ \mathit{ys}\\ & & \mathbf{in}\ \ \mathit{accreverse}\ \mathit{xs}\ v]\!] \end{array} & \\
= & \mathcal{T}[\![\mathbf{case}\ (\mathbf{case}\ \mathit{xss}\ \mathbf{of} & \text{(By 7)}\\
& \qquad \begin{array}{lll} \mathit{Nil} & : & \mathit{Nil}\\ \mathit{Cons}\ \mathit{xs}\ \mathit{xss} & : & \mathit{append}\ \mathit{xs}\ (\mathit{flatten}\ \mathit{xss}))\ \mathbf{of} \end{array} & \\
& \quad \begin{array}{lll} \mathit{Nil} & : & \mathit{ys}\\ \mathit{Cons}\ x\ \mathit{xs} & : & \mathbf{let}\ v = \mathit{Cons}\ x\ \mathit{ys}\\ & & \mathbf{in}\ \ \mathit{accreverse}\ \mathit{xs}\ v]\!] \end{array} & \\
= & \mathbf{case}\ \mathit{xss}\ \mathbf{of} & \text{(By 8,5,6,2)}\\
& \begin{array}{lll} \mathit{Nil} & : & \mathit{ys}\\ \mathit{Cons}\ \mathit{xs}\ \mathit{xss} & : & \mathcal{T}[\![\mathbf{case}\ (\mathit{append}\ \mathit{xs}\ (\mathit{flatten}\ \mathit{xss}))\ \mathbf{of}\\ & & \quad \begin{array}{lll} \mathit{Nil} & : & \mathit{ys}\\ \mathit{Cons}\ x\ \mathit{xs} & : & \mathbf{let}\ v = \mathit{Cons}\ x\ \mathit{ys}\\ & & \mathbf{in}\ \ \mathit{accreverse}\ \mathit{xs}\ v]\!] \end{array} \end{array} & \\
= & \mathbf{case}\ \mathit{xss}\ \mathbf{of} & \text{(By 7)}\\
& \begin{array}{lll} \mathit{Nil} & : & \mathit{ys}\\ \mathit{Cons}\ \mathit{xs}\ \mathit{xss} & : & \end{array} & \\
& \qquad \mathcal{T}[\![\mathbf{case}\ (\mathbf{case}\ \mathit{xs}\ \mathbf{of} & \\
& \qquad\qquad \begin{array}{lll} \mathit{Nil} & : & \mathit{flatten}\ \mathit{xss}\\ \mathit{Cons}\ x\ \mathit{xs} & : & \mathit{Cons}\ x\ (\mathit{append}\ \mathit{xs}\ (\mathit{flatten}\ \mathit{xss})))\ \mathbf{of} \end{array} & \\
& \qquad\quad \begin{array}{lll} \mathit{Nil} & : & \mathit{ys}\\ \mathit{Cons}\ x\ \mathit{xs} & : & \mathbf{let}\ v = \mathit{Cons}\ x\ \mathit{ys}\\ & & \mathbf{in}\ \ \mathit{accreverse}\ \mathit{xs}\ v]\!] \end{array} & \\
= & \mathbf{case}\ \mathit{xss}\ \mathbf{of} & \text{(By 8,5,6,11,3,2,2,4)}\\
& \begin{array}{lll} \mathit{Nil} & : & \mathit{ys}\\ \mathit{Cons}\ \mathit{xs}\ \mathit{xss} & : & \end{array} & \\
& \qquad \mathbf{case}\ \mathit{xs}\ \mathbf{of} & \\
& \qquad\quad \begin{array}{lll} \mathit{Nil} & : & \mathcal{T}[\![\mathbf{case}\ (\mathit{flatten}\ \mathit{xss})\ \mathbf{of}\\ & & \quad \begin{array}{lll} \mathit{Nil} & : & \mathit{ys}\\ \mathit{Cons}\ x\ \mathit{xs} & : & \mathbf{let}\ v = \mathit{Cons}\ x\ \mathit{ys}\\ & & \mathbf{in}\ \ \mathit{accreverse}\ \mathit{xs}\ v]\!] \end{array}\\ \mathit{Cons}\ x\ \mathit{xs} & : & \mathbf{let}\ v = \mathit{Cons}\ x\ \mathit{ys}\\ & & \mathbf{in}\ \ \mathcal{T}[\![\mathbf{case}\ (\mathit{append}\ \mathit{xs}\ (\mathit{flatten}\ \mathit{xss}))\ \mathbf{of}\\ & & \qquad \begin{array}{lll} \mathit{Nil} & : & v\\ \mathit{Cons}\ x\ \mathit{xs} & : & \mathbf{let}\ v' = \mathit{Cons}\ x\ v\\ & & \mathbf{in}\ \ \mathit{accreverse}\ \mathit{xs}\ v']\!] \end{array} \end{array} &
\end{array}
$$

Figure 5.12: Generalised Deforestation of *accreverse* (*flatten xss*) *ys*

$$\mathcal{E}[\![\textbf{let } v = e_0 \textbf{ in } e_1]\!]\ \rho\ \phi = \mathcal{E}[\![e_1]\!]\ \rho[(\mathcal{E}[\![e_0]\!]\ \rho\ \phi)/v]\ \phi$$

The generalised deforestation of the expression *accreverse* (*flatten xss*) *ys* is shown in Figure 5.12. The result of this transformation is shown in Figure 5.13. Non-termination will result if this expression is transformed by the original unextended deforestation algorithm.

```
f xss ys
where
f xss ys      =  case xss of
                   Nil           :  ys
                   Cons xs xss   :  f' xs xss ys

f' xs xss ys  =  case xs of
                   Nil         :  f xss ys
                   Cons x xs   :  f' xs xss (Cons x ys)
```

Figure 5.13: Result of Generalised Deforestation of *accreverse* (*flatten xss*) *ys*

5.3.3 The Generalised Deforestation Theorem

The *generalised deforestation theorem* can now be stated as follows.

Theorem 5.3.2 (Generalised Deforestation Theorem) *Every expression from which shared values have been extracted, and in which all functions have generalised treeless definitions, will be transformed by the generalised deforestation algorithm to an equivalent expression without loss of efficiency.*

□

Proof

The proof of the generalised deforestation theorem is very similar to the proof of the extended deforestation theorem. It can be proved by showing the following three lemmata, which together demonstrate the validity of the theorem.

□

Lemma 5.3.3 *Every expression will be transformed to an equivalent expression if the generalised deforestation algorithm terminates.*

□

Lemma 5.3.4 *Every expression from which shared values have been extracted, and in which all functions have generalised treeless definitions, will be transformed without loss of efficiency if the generalised deforestation algorithm terminates.*

□

Lemma 5.3.5 *The generalised deforestation algorithm will always terminate for every expression in which all functions have generalised treeless definitions.*

□

Proof of Lemma 5.3.3

The proof of this lemma can be found in Appendix C.3.1

□

Proof of Lemma 5.3.4

The proof of this lemma can be found in Appendix C.3.2

□

Proof of Lemma 5.3.5

As for the proof of Lemma 5.2.10, to prove that the generalised deforestation algorithm always terminates, it is sufficient to show that there is a bound on the size of expressions encountered during generalised deforestation.

Definition 5.3.6 (Size of Generalised Expressions) *The definition of the size of expressions (Definition 5.1.7) is extended in the following way to define the size of generalised expressions.*

$$\mathcal{S}[\![\textbf{let } v = e_0 \textbf{ in } e_1]\!] \quad = \quad 1 + max(\mathcal{S}[\![e_0]\!],\mathcal{S}[\![e_1]\!])$$

□

Definition 5.3.7 (Grammar of Expressions Encountered During Generalised Deforestation) *The following terms must be added to the grammar $edg^{s,n}(s,f,n)$ to describe the grammar of expressions which are encountered by the generalised deforestation algorithm.*

$$\begin{array}{rcl}
edg^{s,n}(x,y,z) & ::= & b\ edg_1^{s,n}(x-1,y,z)\ldots edg_n^{s,n}(x-1,y,z) \\
& & \qquad \text{if } x>0,\ y\geq 0 \text{ and } z>0 \\
& | & \textbf{let } v = edg_0^{s,n}(x-1,y,z) \textbf{ in } edg_1^{s,n}(x-1,y,z) \\
& & \qquad \text{if } x>0,\ y\geq 0 \text{ and } z>0 \\
efg^{s,n}(x,y,z) & ::= & b\ efg_1^{s,n}(x-1,y,z)\ldots efg_n^{s,n}(x-1,y,z) \\
& & \qquad \text{if } x>0,\ y\geq 0 \text{ and } z>0 \\
& | & \textbf{let } v = efg_0^{s,n}(x-1,y,z) \textbf{ in } efg_1^{s,n}(x-1,y,z) \\
& & \qquad \text{if } x>0,\ y\geq 0 \text{ and } z>0
\end{array}$$

□

Lemma 5.3.5 can now be proved by showing the following two lemmata.

Lemma 5.3.8 *All expressions encountered by the generalised deforestation algorithm are described by the grammar $edg^{s,n}(s,f,n)$, if the original expression to be transformed is also described by the grammar $edg^{s,n}(s,f,n)$.*

□

Lemma 5.3.9 *The size of all expressions described by the grammar $edg^{s,n}(s,f,n)$ is bounded by $s \times (f+1) \times n$.*

□

Proof of Lemma 5.3.8

The proof of this lemma can be found in Appendix C.3.3.

□

Proof of Lemma 5.3.9

The proof of this lemma is very similar to the proof of Lemma 5.2.13, which is given in Appendix C.2.2.

□

5.4 Related Work

5.4.1 Deforestation

Deforestation grew out of earlier work by Wadler on *listlessness* (Wadler, 1984). The listless transformer is a semi-decision procedure which can convert recursive programs with a bounded evaluation property (programs needing bounded internal storage to perform computation) to equivalent listless programs. The work described in (Wadler, 1985) shows how two listless programs can be combined into a single listless program. The programs to be combined are required to be *preorder*. This means that the inputs of each program are traversed once, and the outputs are produced in a left-to-right manner. The transformations in the listless transformer are not source-to-source, and give a non-functional result. Also, the definition of listless form is not as simple as the treeless form defined for deforestation, so it is harder to determine when an expression is in listless form.

An area of work related to the listless transformer is the transformation technique proposed in (Waters, 1991) for eliminating unnecessary intermediate *series*, where a series is a sequence of items such as vectors or lists which may be unbounded. The class of expressions which can be transformed by this technique are those which are *preorder*, *statically analysable* and *on-line cyclic*. The preorder restriction is the same as that which is used in (Wadler, 1985). The on-line cyclic restriction allows the transformation of functions which take multiple inputs originating from common variables (thus forming cycles) with the on-line

characteristic (lockstep production of one output for every input consumed). The class of expressions which can be transformed by this technique are therefore not as simple as those which can be transformed by deforestation.

A closely related work to the deforestation algorithm is the *supercompiler* described in (Turchin, 1986). This involves *driving* (unfolding) programs to obtain a history of computational states (*configurations*) from the symbolic evaluation of programs. The graphs of configurations obtained can then be used to compile more efficient programs. Folding is applied when a configuration matches one which has been encountered previously, as is done in the deforestation algorithm. The graphs of configurations which are obtained during supercompilation are potentially infinite. A complicated generalisation algorithm is therefore used to obtain a finite set of configurations. These generalised configurations must then be supercompiled again. This is a much more complicated procedure to ensure termination of the transformation process than is required for the deforestation algorithm.

Another related area to deforestation is partial evaluation (Bjorner *et al.* , 1988). Partial evaluation involves the specialisation of function calls in which the arguments are known (or partially known). These calls can be transformed into more efficient equivalent functions which make use of the known properties of their arguments. Deforestation allows the transformation of symbolic data in which the values of arguments may not be known. The result of the partial evaluation process is a residual program which contains evaluated and unevaluated expressions. Transformations in the deforestation algorithm are source-to-source.

5.4.2 Extended Deforestation

Other work has already been done on trying to extend deforestation for first order expressions in (Chin, 1991) and (Chin, 1992). This work is explained using a *producer-consumer* model of functions. A function argument is a good consumer if it is linear and non-accumulating, where linear and accumulating are defined in the same way as in this chapter. An extended treeless form of expression is defined in which all good consumers are variables. An expression is a good producer if it satisfies this extended treeless form. Good producers are fused with good consumers during transformation, whilst expressions which are not good consumers or good producers are extracted and transformed separately. This extended treeless form is more restrictive than the extended treeless form defined in this chapter. All good consumers are restricted to being variables, even if they are not transient structures. Thus, for example, the *flatten* function defined in Figure 2.2 is not in the extended treeless form defined in (Chin,

1991) and (Chin, 1992), but it is in the extended treeless form defined in this chapter. More intermediate structures can therefore be eliminated by the method presented in this thesis.

Previous work has been done in (Hamilton & Jones, 1991b; Hamilton & Jones, 1991a; Hamilton, 1992a) to try to extend deforestation for first order functions. This work also makes use of the information obtained by static analysis. In (Hamilton & Jones, 1991b), a *transmission* analysis is used to determine whether structures are transient. This analysis determines whether a structure will appear directly in the result of an expression. Structures which do not satisfy this criterion, and are not variables, are extracted and transformed independently. However, accumulating parameters and shared values are not extracted in this work, so non-termination or loss of efficiency may occur as a result of applying the deforestation algorithm. In (Hamilton & Jones, 1991a), a *creation* analysis is performed in addition to the transmission analysis. This analysis is used to determine whether an expression will produce a list result in preorder. Transient structures which are created in this way can be eliminated by the deforestation algorithm. This analysis is still not sufficient to ensure the termination of the deforestation algorithm, because accumulating parameters are not extracted. In the work described in (Hamilton, 1992a), accumulating parameters are extracted, thus ensuring the termination of the deforestation algorithm. The work described in (Hamilton, 1992a) is similar to the work described in this chapter.

5.4.3 Generalised Deforestation

The blazed deforestation algorithm described in (Wadler, 1990b) is a generalisation of the original deforestation algorithm. This generalisation is performed on the basis of the types of expressions. Expressions of atomic type are blazed $\ominus$, and expressions of structured type are blazed $\oplus$. Expressions blazed $\ominus$ are extracted using **let** expressions and transformed independently, since they cannot be intermediate structures. More expressions can be transformed as a result of this generalisation, but there are still many function definitions which are not in the described blazed treeless form. More intermediate structures can therefore be eliminated using the generalised deforestation algorithm described in this chapter.

The universal deforestation algorithm described in (Chin, 1991) and (Chin, 1992) is similar to the generalised deforestation algorithm described in this chapter. Any sub-expressions which prevent an expression from being in the described extended treeless form are extracted using **let** expressions and are transformed separately. Thus, any function arguments which are not variables are extracted, even if they are not transient structures. More intermediate

structures can therefore be eliminated by the generalised deforestation algorithm described in this chapter.

In the work described in (Turchin, 1986), the graphs of configurations which are obtained during supercompilation are potentially infinite. Generalisation is therefore performed to obtain a finite set of configurations. This generalisation determines a more general configuration for a configuration which does not precisely match a previous one. The algorithm for this generalisation which is presented in (Turchin, 1988) ensures termination of the supercompilation process. This generalisation algorithm is a sophisticated *on-line* technique (Jones, 1988), which looks back at the history of configurations at transformation-time in order to perform on-the-fly generalisation. The generalised deforestation algorithm presented here uses a simple *off-line* generalisation to determine which intermediate structures can be eliminated. This off-line technique is used to determine in advance where generalisations must be introduced. The supercompiler requires a complex algorithm to determine at transformation time when generalisations must be performed, and re-supercompilation of the generalised configurations when they are introduced.

5.5 Conclusion

In this chapter, it has been shown how information obtained by usage counting analysis can be used to guide the transformation when compile-time garbage avoidance is performed. The method of compile-time garbage avoidance which was used is the deforestation algorithm described in (Wadler, 1990b). A treeless form of expression was characterised in (Wadler, 1990b) which does not create any intermediate structures. It has been proved in this chapter that the deforestation algorithm will always terminate for expressions in which all functions have definitions which are in treeless form.

The deforestation algorithm will also terminate for some expressions in which functions have definitions which are not in treeless form. It was therefore shown how treeless form can be extended by making use of the information obtained by usage counting analysis. It was then proved that the deforestation algorithm will always terminate for expressions in which all functions have definitions which are in this extended treeless form.

Some intermediate structures can also be eliminated from expressions in which some functions have definitions which are not in extended treeless form. It was therefore shown how any function definition can be generalised in such a way that it will be in extended treeless form. The deforestation algorithm was extended to be able to deal with these generalisations.

Chapter 6

Conclusion

In this thesis, it has been shown how the use of storage in lazy functional programs can be optimised at compile-time by utilising the information obtained by usage counting analysis. Two different approaches to performing this optimisation were taken; compile-time garbage collection and compile-time garbage avoidance. The information obtained by usage counting analysis can be used to annotate programs for compile-time garbage collection, and to guide the transformation when compile-time garbage avoidance is performed. In this chapter, a summary is given of the work in this thesis, directions for further work are discussed, and general conclusions are drawn.

The remainder of this chapter is structured as follows:

- **Section 6.1**: a summary is given of the work in this thesis.
- **Section 6.2**: directions for further work arising from this thesis are discussed.
- **Section 6.3**: the general conclusions of the thesis are given.

6.1 Summary of Thesis

6.1.1 Language

In Chapter 2, the syntax and semantics of the language used throughout this thesis were defined. The standard semantics of the language do not model the use of store, and so could not be used as a reference against which store-related analyses and optimisations could be proved correct. Non-standard store semantics were therefore defined for the language. To ensure that these store semantics model the use of store safely, they were proved to be congruent to the standard semantics of the language.

6.1.2 Compile-Time Garbage Detection

In Chapter 3, it was shown how the cells which will become garbage within a program can be detected at compile-time. A cell will become garbage during the evaluation of an expression if it is unshared when it loses a reference. To determine that a cell is unshared (used once), the store semantics presented in Chapter 2 were augmented to incorporate usage counting. These usage counting store semantics had to be abstracted in some way to allow usage counts to be determined at compile-time. Usage counting store values were therefore abstracted to usage patterns. These patterns are finite objects which indicate the number of times each part of a value is used. A usage counting analysis was then defined, using these patterns, to determine at compile-time the number of times each part of a value will be used in future computations. This usage counting analysis was then proved to be safe with respect to the usage counting store semantics by showing that the usage count of a value determined by the analysis is not less than its actual usage count. Thus, it is not assumed that a cell will become garbage when it is still required by a program.

6.1.3 Compile-Time Garbage Collection

In Chapter 4, it was shown how information obtained from usage counting analysis can be used to annotate programs for compile-time garbage collection. Three different methods for compile-time garbage collection were presented; compile-time garbage marking, explicit deallocation and destructive allocation. Compile-time garbage marking involves marking cells at their allocation to indicate that they will become garbage after their first use. This method requires an extra bit per cell to indicate whether or not a cell is marked, so the extra space required may be more than the space which is saved by using this method. Explicit deallocation involves returning cells to the memory manager at a particular point in a program. This avoids the need to mark cells since it is known that cells will always become garbage at this point. This method requires that the run-time garbage collector makes use of a free list, which is not the most efficient way to perform garbage collection at run-time. Destructive allocation involves reusing cells directly within a program for further allocations. This avoids the need to use a free list, so a more efficient method for performing run-time garbage collection can be used. Store semantics were defined for programs which have been annotated for each of these methods of compile-time garbage collection, and the correctness of these store semantics was considered.

6.1.4 Compile-Time Garbage Avoidance

In Chapter 5, it was shown how information obtained by usage counting analysis can be used to guide the transformation when compile-time garbage avoidance is performed. The method of compile-time garbage avoidance which was used is the deforestation algorithm described in (Wadler, 1990b). A treeless form of expression was characterised in (Wadler, 1990b) which does not create any intermediate structures. A proof was given in Chapter 5 that the deforestation algorithm will always terminate for expressions in which functions have definitions which are in treeless form. The deforestation algorithm will also terminate for some expressions in which functions have definitions which are not in treeless form. It was therefore shown how treeless form can be extended by making use of the information obtained by usage counting analysis. It was then proved that the deforestation algorithm will always terminate for expressions in which all functions have definitions which are in this extended treeless form. Some intermediate structures can also be eliminated from expressions in which some functions have definitions which are not in extended treeless form. It was therefore shown how any function definition can be generalised in such a way that it will

be in extended treeless form. The deforestation algorithm was extended to be able to deal with these generalisations, and it was proved that this generalised deforestation algorithm will always terminate.

6.2 Further Work

There are many directions for further work arising from this thesis. These are summarised below.

6.2.1 Compile-Time Garbage Detection

The usage counting analysis presented in this thesis is for a first order monomorphic language. This analysis could be extended to deal with higher order expressions and polymorphism.

In order to deal with higher order expressions, the analysis could be combined with an abstract interpretation in which all higher order values are analysed in a forward direction, in the manner described in (Hughes, 1988). Alternatively, a closure analysis, such as the one performed in (Sestoft, 1989), could be performed to determine the set of possible abstract closures to which a function can be evaluated during the execution of a program. The least upper bound of the corresponding contexts of these abstract closures could then be determined to give a safe approximation to the context of each function.

In (Abramsky, 1985), it is shown that it is necessary only to analyse a polymorphic function at its simplest instance when abstract interpretation is used to perform strictness analysis. This result for the simplest instance of the function is then applicable to every instance of the function. In order to extend usage counting analysis to deal with polymorphism, it would have to be shown that this is also the case for usage counting analysis.

6.2.2 Compile-Time Garbage Collection

A full proof of correctness is required for the three methods of compile-time garbage collection which have been presented in this thesis. This would involve defining an equivalence relation between the usage counting store semantics for programs which have been annotated for compile-time garbage collection and the usage counting store semantics for unannotated programs.

If usage counting analysis could be extended to handle higher order expressions, then the described methods for compile-time garbage collection could also be extended. This would

allow an implementation of the methods for compile-time garbage collection to be incorporated into the optimisation phase of a compiler, and a thorough assessment could be made of the benefits which can be obtained by these optimisations.

6.2.3 Compile-Time Garbage Avoidance

More intermediate structures could be removed from expressions by making use of laws (for example, the commutativity or associativity of functions). In (Wadler, 1987a), it is shown how some intermediate structures which are unshared can be removed from function definitions by making use of the associativity of the *append* function. However, the function definitions which result from this transformation contain accumulating parameters, so they are still not suitable for transformation by the deforestation algorithm.

The generalised deforestation algorithm could also be extended to deal with higher order expressions. It has already been shown in (Marlow & Wadler, 1992) and (Hamilton, 1993) how the deforestation transformation rules can be re-formulated in order to be able to deal with higher order expressions. If usage counting analysis could be extended to handle higher order expressions, then the generalised deforestation algorithm presented in this thesis could also be extended. This generalised algorithm would allow a much wider range of expressions to be transformed. For example, in many of the more widely used higher order functions (for example *map*, *filter*, *fold*), the function type argument is used more than once. These function type arguments would therefore have to be extracted before the functions could be involved in higher order deforestation transformations. Also, in higher order languages, applications are of the form $e_1\ e_2$, where the function e_1 is applied to the argument e_2. Without an analysis similar to usage counting analysis to determine which expressions are intermediate structures, it would have to be assumed that the expression e_2 is intermediate, and it would have to be restricted to being a variable. Thus, not many useful higher order expressions could be transformed.

Finally, an implementation of a higher order generalised deforestation algorithm could be incorporated into the optimisation phase of a compiler so that a thorough assessment could be made of the benefits obtained by this optimisation.

6.3 General Conclusions

In this thesis, it has been shown that usage counting analysis provides useful information for the compile-time optimisation of store usage in lazy functional programs. The three desirable criteria for compile-time optimisations given in Section 1.1.3 (termination, automatability and correctness) have been of paramount importance in the optimisations described in this thesis.

It has been shown how usage counting information can be used to annotate lazy programs for compile-time garbage collection. Most of the previous work in the area of compile-time garbage collection has been for strict languages. Three different methods of compile-time garbage collection were presented; compile-time garbage marking, explicit deallocation and destructive allocation. The correctness of each of these methods was considered. In most of the previous work in the area of compile-time garbage collection, correctness has not been considered. Of the three described methods for compile-time garbage collection, it has been found that destructive allocation is the only method which is of practical use.

It has also been shown how usage counting information can be used to guide the transformation when compile-time garbage avoidance is performed. The method of compile-time garbage avoidance which was used is the deforestation algorithm described in (Wadler, 1990b). A proof of the deforestation theorem stated in (Wadler, 1990b) has been given in this thesis. It has also been shown how the class of expressions for which the deforestation algorithm is guaranteed to terminate can be extended by utilising the information obtained by usage counting analysis.

Compile-time garbage avoidance produces greater increases in efficiency than compile-time garbage collection. Time which is required to allocate, traverse and subsequently deallocate intermediate structures is saved through the use of compile-time avoidance, but not through the use of compile-time garbage collection. Compile-time garbage collection merely serves to reduce the amount of time required for garbage collection at run-time. However, much of the garbage which can be collected by compile-time garbage collection cannot be avoided at compile-time. The two approaches are therefore complementary, and the expressions resulting from compile-time garbage avoidance transformations could be annotated for compile-time garbage collection to further optimise the use of storage.

References

Abramsky, S. 1985. Strictness Analysis and Polymorphic Invariance. *Lecture Notes in Computer Science*, **217**, 1–23.

Abramsky, S., & Hankin, C. (eds). 1987. *Abstract Interpretation of Declarative Languages.* Ellis Horwood.

Andersen, J. 1990 (Aug.). *Abstract Interpretation Using Operational Semantics.* Ph.D. thesis, University of London.

Appel, A.W. 1987. Garbage Collection can be Faster Than Stack Allocation. *Information Processing Letters*, **25**(4), 275–279.

Appel, A.W. 1989. Simple Generational Garbage Collection and Fast Allocation. *Software - Practice and Experience*, **19**(2), 171–183.

Augustsson, L. 1984. A Compiler for Lazy ML. *Pages 218–227 of: Proceedings of the ACM Conference on LISP and Functional Programming.*

Augustsson, L. 1985. Compiling Pattern Matching. *Pages 368–381 of: Functional Programming Languages and Computer Architecture.* Lecture Notes in Computer Science, vol. 201. Springer-Verlag.

Baker-Finch, C.A. 1993. *Relevance and Contraction: A Logical Basis for Strictness and Sharing Analysis.* Tech. rept. ISE RR 34/94. University of Canberra.

Barth, J.M. 1977. Shifting Garbage Collection Overhead to Compile Time. *Communications of the ACM*, **20**(7), 513–518.

Bellegarde, F. 1986. Rewriting Systems on FP Expressions That Reduce the Number of Sequences Yielded. *Science of Computer Programming*, **6**, 11–34.

Bjorner, D., Ershov, A.P., & Jones, N.D. (eds). 1988. *Workshop on Partial Evaluation and Mixed Computation.* North-Holland.

Bloss, A. 1989. Update Analysis and the Efficient Implementation of Functional Aggregates. *Pages 26–38 of: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture.*

Burstall, R.M., & Darlington, J. 1977. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, **24**(1), 44–67.

Chase, D.R. 1987 (Aug.). *Garbage Collection and Other Optimizations.* Ph.D. thesis, Rice University, Houston, Texas.

Chin, Wei-Ngan. 1991 (Oct.). Generalising Deforestation for All First-Order Functional Programs. *Pages 173–181 of: Journées de Travail sur L'Analyse Statique en Programmation Equationnelle, Fonctionnelle et Logique.*

Chin, Wei-Ngan. 1992. Safe Fusion of Functional Expressions. *Pages 11–20 of: Proceedings of the ACM Conference on LISP and Functional Programming.*

Clark, D.W. 1979. Measurements of Dynamic List Structure Use in Lisp. *IEEE Transactions on Software Engineering*, **5**(1), 51–59.

Clark, D.W., & Green, C.C. 1977. An Empirical Study of List Structure in Lisp. *Communications of the ACM*, **20**(2), 78–86.

Clark, D.W., & Green, C.C. 1978. A Note on Shared List Structure in Lisp. *Information Processing Letters*, **7**(6), 312–314.

Collins, G.E. 1960. A Method For Overlapping and Erasure of Lists. *Communications of the ACM*, **3**(12), 655–657.

Cousot, P., & Cousot, R. 1977 (Jan.). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Pages 238–252 of: Proceedings of the Fourth ACM Symposium on Principles of Programming Languages.*

Damas, L., & Milner, R. 1982. Principal Type Schemes for Functional Programs. *Pages 207–212 of: Proceedings of the Ninth ACM Symposium on Principles of Programming Languages.*

Deutsch, A. 1990 (Jan.). On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-Order Functional Specifications. *Pages 157–168 of: Proceedings of the ACM Symposium on Principles of Programming Languages.*

Deutsch, L.P., & Bobrow, D.G. 1976. An Efficient, Incremental, Automatic Garbage Collector. *Communications of the ACM*, **19**(9), 522–526.

Draghicescu, M., & Purushothaman, S. 1990. A Compositional Analysis of Evaluation Order and its Application. *Pages 242–250 of: Proceedings of the 1990 ACM Conference on Lisp and Functional Programming.*

Fenichel, R.R., & Yochelson, J.C. 1969. A LISP Garbage Collector for Virtual-Memory Computer Systems. *Communications of the ACM*, **12**(11), 611–612.

Fradet, P. 1991. Syntactic Detection of Single-Threading Using Continuations. *Lecture Notes in Computer Science*, **523**, 241–258.

Friedman, D.P., & Wise, D.S. 1976. CONS Should Not Evaluate Its arguments. *Automata Languages and Programming*, 257–284.

Gill, A., Launchbury, J., & Peyton Jones, S.L. 1993. A Short Cut to Deforestation. *In: Proceedings of the Sixth International Conference on Functional Programming Languages and Computer Architecture.*

Girard, J.-Y. 1987. Linear Logic. *Theoretical Computer Science*, **50**(1), 1–101.

Goldberg, B., & Gil Park, Y. 1990. Higher Order Escape Analysis: Optimizing Stack Allocation in Functional Program Implementations. *Lecture Notes in Computer Science*, **432**, 152–160.

Gomard, C.K., & Sestoft, P. 1991. Globalization and Live Variables. *Pages 166–177 of: Symposium on Partial Evaluation and Semantics-Based Program Manipulation.*

Gopinath, K., & Hennessy, J.L. 1989. Copy Elimination in Functional Languages. *Pages 303–314 of: Proceedings of the Sixteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.*

Guzmán, J.C., & Hudak, P. 1990 (June). Single Threaded Polymorphic Lambda Calculus. *In: Fifth IEEE Symposium on Logic in Computer Science.*

Hamilton, G.W. 1992a. *Compile-Time Garbage Avoidance*. Technical Report TR 74. Dept. of Computing Science and Mathematics, University of Stirling.

Hamilton, G.W. 1992b (Sept.). Sharing Analysis of Lazy First Order Functional Programs. *Pages 68–78 of: Proceedings of the Workshop on Static Analysis (WSA '92)*. BIGRE, vol. 81–82.

Hamilton, G.W. 1993. *Higher Order Deforestation*. Unpublished.

Hamilton, G.W., & Jones, S.B. 1990. *Compile-Time Garbage Collection by Necessity Analysis*. Technical Report TR 67. Dept. of Computing Science and Mathematics, University of Stirling.

Hamilton, G.W., & Jones, S.B. 1991a. Extending Deforestation for First Order Functional Programs. *Pages 134–145 of: Proceedings of the 1991 Glasgow Workshop on Functional Programming (GWFP '91)*. Workshops in Computing. Springer-Verlag.

Hamilton, G.W., & Jones, S.B. 1991b (Oct.). Transforming Programs to Eliminate Intermediate Structures. *Pages 182–188 of: Journées de Travail sur L'Analyse Statique en Programmation Equationnelle Fonctionnelle et Logique (WSA '91)*. "BIGRE", vol. 74.

Henderson, P., & Morris, J. 1976. A Lazy Evaluator. *Pages 95–103 of: Proceedings of the Third Symposium on Principles of Programming Languages*.

Hindley, R. 1979. The Principal Type Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematics Society*, **146**, 29–60.

Hudak, P. 1987. A Semantic Model of Reference Counting and its Abstraction. *Pages 45–62 of:* Abramsky, S., & Hankin, C. (eds), *Abstract Interpretation of Declarative Languages*. Ellis Horwood.

Hudak, P., & Bloss, A. 1985. The Aggregate Update Problem in Functional Programming Systems. *Pages 300–314 of: Proceedings of the Twelfth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*.

Hudak, P., & Wadler, P. 1990. *Report on the Programming Language Haskell*. Technical Report. Yale University and Glasgow University.

Hughes, R.J.M. 1988. Backwards Analysis of Functional Programs. *Pages 187–208 of:* Bjørner, D., Ersov, A.P., & Jones, N.D. (eds), *Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation.* Elsevier Science Publishers B.V. North-Holland.

Hughes, R.J.M. 1989. Why Functional Programming Matters. *The Computer Journal*, **32**(2), 98–107.

Hughes, S. 1991 (Oct.). *Static Analysis of Store Use in Functional Programs.* Ph.D. thesis, Imperial College, University of London.

Inoue, K., Seki, H., & Yagi, H. 1988. Analysis of Functional Programs to Detect Run-Time Garbage Cells. *ACM Transactions on Programming Languages and Systems*, **10**(4), 555–578.

Jensen, T.P. 1990. *Context Analysis of Functional Programs.* M.Phil. thesis, University of Copenhagen.

Jensen, T.P., & Mogensen, T.Æ. 1990. A Backwards Analysis for Compile-Time Garbage Collection. *Lecture Notes in Computer Science*, **432**, 227–239.

Johnsson, T. 1985 (Feb.). Lambda Lifting: Transforming Programs to Recursive Equations. *Pages 165–180 of: Proceedings of the Workshop on Implementation of Functional Languages.*

Jones, N.D. 1988. Challenging Problems in Partial Evaluation and Mixed Computations. *Pages 1–14 of: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation.*

Jones, S.B., & Le Métayer, D. 1989. Compile-Time Garbage Collection by Sharing Analysis. *Pages 54–74 of: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture.*

Josephs, M.B. 1987. *Functional Programming With Side-Effects.* Technical Monograph 55. PRG, Oxford University.

Kuo, T-M., & Mishra, P. 1989. Strictness Analysis: A New Perspective Based on Type Inference. *Pages 260–272 of: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture.*

Launchbury, J., Gill, A., Hughes, J., Marlow, S., Peyton Jones, S.L., & Wadler, P. 1992 (July). Avoiding Unnecessary Updates. *Pages 144–153 of: Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming.*

Lieberman, H., & Hewitt, C. 1991. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, **11**(3–4), 217–242.

Marlow, S. 1993 (July). Update Avoidance Analysis by Abstract Interpretation. *In: Proceedings of the Sixth Annual Glasgow Workshop on Functional Programming.*

Marlow, S., & Wadler, P. 1992 (July). Deforestation for Higher-Order Functions. *Pages 154–165 of: Proceedings of the Fifth Annual Glasgow Workshop on Functional Programming.*

Mason, I.A. 1988. Verification of Programs That Destructively Update Data. *Science of Computer Programming*, **10**, 177–210.

Milner, R. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, **17**, 348–375.

Moon, D.A. 1984. Garbage Collection in a Large Lisp System. *Pages 235–246 of: Proceedings of the ACM Conference on LISP and Functional Programming.*

Mycroft, A. 1981. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. thesis, University of Edinburgh.

Peterossi, A. 1978. Improving Memory Utilization in Transforming Recursive Programs. *In: Seventh International Symposium on Mathematical Foundations of Computer Science.*

Pleban, U.F. 1990. *Preexecution Analysis Based on Denotational Semantics.* Ph.D. thesis, University of Kansas.

Ruggieri, C., & Murtagh, T.P. 1988 (Jan.). Lifetime Analysis of Dynamically Allocated Objects. *Pages 285–293 of: Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.*

Sastry, A.V.S., Clinger, W., & Ariola, Z. 1993. Order-of-evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates. *Pages 266–275 of: Proceedings of the Sixth International Conference on Functional Programming Languages and Computer Architecture.*

Schmidt, D.A. 1985. Detecting Global Variables in Denotational Specifications. *ACM Transactions on Programming Languages and Systems*, **7**(2), 299–310.

Schwarz, J. 1978. Verifying the Safe Use of Destructive Operations in Applicative Programs. *Pages 395–411 of: Proceedings of the Third International Symposium on Programming.*

Sestoft, P. 1989. Replacing Function Parameters by Global Variables. *Pages 39–53 of: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture.*

Smetsers, S., Barendsen, E., van Eekelen, M., & Plasmeijer, R. 1993. *Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs.* Technical Report 93-4. University of Nijmegen.

Turchin, V.F. 1986. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, **8**(3), 90–121.

Turchin, V.F. 1988. The Algorithm of Generalization in the Supercompiler. *Pages 531–549 of: Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation.*

Turner, D.A. 1985. Miranda: A Non-Strict Functional Language With Polymorphic Types. *Lecture Notes in Computer Science*, **201**, 1–16.

Ungar, D. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *Pages 157–167 of: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.*

Wadler, P. 1981. Applicative Style of Programming, Program Transformation and List Operators. *Pages 25–32 of: Proceedings of the International Conference on Functional Programming Languages and Computer Architecture.*

Wadler, P. 1984. Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile-Time. *Pages 45–52 of: Proceedings of the ACM Conference on LISP and Functional Programming.*

Wadler, P. 1985. Listlessness is Better than Laziness II: Composing Listless Functions. *Lecture Notes in Computer Science*, **217**, 282–305.

Wadler, P. 1987a (Dec.). *The Concatenate Vanishes.* FP Electronic Mailing List.

Wadler, P. 1987b. Efficient Compilation of Pattern Matching. *Pages 78–103 of:* Jones, S.L. Peyton (ed), *The Implementation of Functional Programming Languages*. Prentice Hall.

Wadler, P. 1990a (June). Comprehending Monads. *Pages 61–78 of: Proceedings of the ACM Conference on Lisp and Functional Programming.*

Wadler, P. 1990b. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, **73**, 231–248.

Wadler, P. 1990c. Linear Types Can Change the World! *In:* Broy, M., & Jones, C. (eds), *IFIP Working Conference on Programming Concepts and Methods*. North Holland. Sea of Galilee, Israel.

Wakeling, D., & Runciman, C. 1991. Linearity and Laziness. *Lecture Notes in Computer Science*, **523**, 215–240.

Waters, R.C. 1991. Automatic Transformation of Series Expressions into Loops. *ACM Transactions on Programming Languages and Systems*, **13**(1), 52–98.

Wise, D.S., & Friedman, P. 1977. The One-Bit Reference Count. *BIT*, **17**(4), 351–359.

Wright, D.A., & Baker-Finch, C.A. 1993 (Sept.). Usage Analysis With Natural Reduction Types. *Pages 254–266 of: Third International Workshop on Static Analysis*. Lecture Notes in Computer Science, vol. 724.

Appendix A

Proofs for Language Semantics

A.1 Congruence of Expressions

for all $\rho_{\mathcal{E}^{store}} \in \text{Bve}_{\mathcal{E}^{store}}$, $\phi_{\mathcal{E}^{store}} \in \text{Fve}_{\mathcal{E}^{store}}$, $\sigma_{\mathcal{E}^{store}} \in \text{Store}_{\mathcal{E}^{store}}$, $\phi_{\mathcal{E}} \in \text{Fve}_{\mathcal{E}}$, $e \in \text{Exp}$:

if for all $f \in \text{dom}(\phi_{\mathcal{E}^{store}})$:

$$\Phi(\phi_{\mathcal{E}^{store}}[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_{\mathcal{E}^{store}}) = \phi_{\mathcal{E}}[\![f]\!]\ (\Phi(loc_1, \sigma_{\mathcal{E}^{store}})) \ldots (\Phi(loc_n, \sigma_{\mathcal{E}^{store}}))$$

then for all $v \in \text{dom}(\rho_{\mathcal{E}^{store}})$:

$$\Phi(\mathcal{E}^{store}[\![e]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) = \mathcal{E}[\![e]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}$$

Proof

The proof is by structural induction on the expression e.

Base Cases

Case 1: $e ::= k$

$$\mathcal{E}[\![k]\!]\ \rho\ \phi \quad = \quad k$$

$$\mathcal{E}^{store}[\![k]\!]\ \rho\ \phi\ \sigma \quad = \quad alloc(k,\sigma)$$

$$\begin{array}{lrcl} & \Phi(\mathcal{E}^{store}[\![k]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) & = & k \\ \Rightarrow & \Phi(\mathcal{E}^{store}[\![k]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) & = & \mathcal{E}[\![k]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}} \end{array}$$

Case 2: $e ::= v$

$$
\begin{array}{lcll}
\mathcal{E}[\![v]\!]\ \rho\ \phi & = & \rho[\![v]\!] & \\
\mathcal{E}^{store}[\![v]\!]\ \rho\ \phi\ \sigma & = & (loc, \sigma'[loc/\rho[\![v]\!]]), & \text{if } (\sigma\ (\rho[\![v]\!])) \in \text{Closure} \\
& & \text{where} & \\
& & (loc, \sigma') = (\sigma\ (\rho[\![v]\!]))\ \sigma & \\
& = & ((\sigma\ (\rho[\![v]\!])), \sigma), & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lrcl}
& \Phi(\mathcal{E}^{store}[\![v]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) & = & \Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}}) \\
\Rightarrow & \Phi(\mathcal{E}^{store}[\![v]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) & = & \mathcal{E}[\![v]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}
\end{array}
$$

Inductive Cases

Case 1: $e ::= b\ e_1 \ldots e_n$

$$
\begin{array}{lcl}
\mathcal{E}[\![b\ e_1 \ldots e_n]\!]\ \rho\ \phi & = & \mathcal{B}[\![b]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi)\ \ldots\ (\mathcal{E}[\![e_n]\!]\ \rho\ \phi) \\
\mathcal{E}^{store}[\![b\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma & = & \mathcal{B}_{\mathcal{E}^{store}}[\![b]\!]\ loc_1 \ldots loc_n\ \sigma_n \\
& & \text{where} \\
& & (loc_1, \sigma_1) = \mathcal{E}^{store}[\![e_1]\!]\ \rho\ \phi\ \sigma \\
& & \qquad \vdots \\
& & (loc_n, \sigma_n) = \mathcal{E}^{store}[\![e_n]\!]\ \rho\ \phi\ \sigma_{n-1}
\end{array}
$$

$$
\begin{array}{ll}
& \Phi(\mathcal{E}^{store}[\![b\ e_1 \ldots e_n]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) \\
= & \mathcal{B}[\![b]\!]\ \Phi(\mathcal{E}^{store}[\![e_1]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}})\ \ldots \\
& \Phi(\mathcal{E}^{store}[\![e_n]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) \\
= & \mathcal{B}[\![b]\!]\ (\mathcal{E}[\![e_1]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}})\ \ldots \\
& (\mathcal{E}[\![e_n]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}) \\
& \text{(by inductive hypothesis)}
\end{array}
$$

$$= \mathcal{E}[\![b\ e_1 \ldots e_n]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}$$

$$\Rightarrow \Phi(\mathcal{E}^{store}[\![b\ e_1 \ldots e_n]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}})$$

$$= \mathcal{E}[\![b\ e_1 \ldots e_n]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}$$

Case 2: $e ::= c\ e_1 \ldots e_n$

$$\mathcal{E}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi = \mathcal{C}[\![c]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{E}[\![e_n]\!]\ \rho\ \phi)$$

$$\mathcal{E}^{store}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma = \mathcal{C}_{\mathcal{E}^{store}}[\![c]\!]\ loc_1 \ldots loc_n\ \sigma_n$$

where

$$(loc_1, \sigma_1) = alloc((\mathcal{E}^{store}[\![e_1]\!]\ \rho\ \phi), \sigma)$$

$$\vdots$$

$$(loc_n, \sigma_n) = alloc((\mathcal{E}^{store}[\![e_n]\!]\ \rho\ \phi), \sigma_{n-1})$$

$$\Phi(\mathcal{E}^{store}[\![c\ e_1 \ldots e_n]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}})$$

$$= \mathcal{C}[\![c]\!]\ \Phi(\mathcal{E}^{store}[\![e_1]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) \ldots \Phi(\mathcal{E}^{store}[\![e_n]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}})$$

$$= \mathcal{C}[\![c]\!]\ (\mathcal{E}[\![e_1]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}) \ldots (\mathcal{E}[\![e_n]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}})$$

(by inductive hypothesis)

$$= \mathcal{E}[\![c\ e_1 \ldots e_n]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}$$

$$\Rightarrow \Phi(\mathcal{E}^{store}[\![c\ e_1 \ldots e_n]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}})$$

$$= \mathcal{E}[\![c\ e_1 \ldots e_n]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}$$

Case 3: $e ::= f\ e_1 \ldots e_n$

$$\mathcal{E}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi = \phi[\![f]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{E}[\![e_n]\!]\ \rho\ \phi)$$

$$\mathcal{E}^{store}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma = \phi[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_n$$

where

$$(loc_1, \sigma_1) = alloc((\mathcal{E}^{store}[\![e_1]\!]\ \rho\ \phi), \sigma)$$

$$\vdots$$

$$(loc_n, \sigma_n) = alloc((\mathcal{E}^{store}[\![e_n]\!]\ \rho\ \phi), \sigma_{n-1})$$

$$
\begin{aligned}
& \Phi(\mathcal{E}^{store}[\![f\ e_1 \ldots e_n]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) \\
= \quad & \phi_{\mathcal{E}}[\![f]\!]\ \Phi(\mathcal{E}^{store}[\![e_1]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) \ldots \\
& \Phi(\mathcal{E}^{store}[\![e_n]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) \\
& \text{(by assumptions for } \phi_{\mathcal{E}^{store}} \text{ and } \phi_{\mathcal{E}} \text{ in Lemma 2.5.2)} \\
= \quad & \phi_{\mathcal{E}}[\![f]\!]\ (\mathcal{E}[\![e_1]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}) \ldots \\
& (\mathcal{E}[\![e_n]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}) \\
& \text{(by inductive hypothesis)} \\
\Rightarrow \quad & \Phi(\mathcal{E}^{store}[\![f\ e_1 \ldots e_n]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) \\
= \quad & \mathcal{E}[\![f\ e_1 \ldots e_n]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}
\end{aligned}
$$

Case 4: e ::= **case** e_0 **of** $p_1 : e_1 \mid \ldots \mid p_k : e_k$

$$
\begin{aligned}
& \mathcal{E}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!] \rho\ \phi \\
= \quad & \mathcal{E}[\![e_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi \\
& \text{where} \\
& x = \mathcal{E}[\![e_0]\!]\ \rho\ \phi \\
& p_i = c\ v_1 \ldots v_n \text{ and } match(x, c)
\end{aligned}
$$

$$
\begin{aligned}
& \mathcal{E}^{store}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]\ \rho\ \phi\ \sigma \\
= \quad & \mathcal{E}^{store}[\![e_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi\ \sigma' \\
& \text{where} \\
& (loc, \sigma') = \mathcal{E}^{store}[\![e_0]\!]\ \rho\ \phi\ \sigma \\
& x = \sigma'\ loc \\
& p_i = c\ v_1 \ldots v_n \text{ and } match(x, c)
\end{aligned}
$$

$$
\begin{aligned}
& \Phi(\mathcal{E}^{store}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) \\
= \quad & \Phi(\mathcal{E}^{store}[\![e_i]\!]\ \rho_{\mathcal{E}^{store}}[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi_{\mathcal{E}^{store}}\ \sigma') \\
& \text{where} \\
& (loc, \sigma') = \mathcal{E}^{store}[\![e_0]\!]\ \rho\ \phi\ \sigma \\
& x = \sigma'\ loc \\
& p_i = c\ v_1 \ldots v_n \text{ and } match(x, c)
\end{aligned}
$$

$$
\begin{aligned}
&= \mathcal{E}[\![e_i]\!]\ \rho_{\mathcal{E}}[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi_{\mathcal{E}} \\
&\quad \text{where} \\
&\quad x = \mathcal{E}[\![e_0]\!]\ \rho_{\mathcal{E}}\ \phi_{\mathcal{E}} \\
&\quad \rho_{\mathcal{E}} = [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v] \\
&\quad p_i = c\ v_1 \ldots v_n \text{ and } match(x, c) \\
&\quad \text{(by inductive hypothesis)} \\
&= \mathcal{E}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}} \\
\Rightarrow\ & \Phi(\mathcal{E}^{store}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]\ \rho_{\mathcal{E}^{store}}\ \phi_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) \\
&= \mathcal{E}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]\ [\Phi(\rho_{\mathcal{E}^{store}}[\![v]\!], \sigma_{\mathcal{E}^{store}})/v]\ \phi_{\mathcal{E}}
\end{aligned}
$$

□

A.2 Congruence of Function Variable Environments

for all $p \in$ Prog:

if $\mathcal{E}_p[\![p]\!] = \mathcal{E}[\![e]\!]\ (\lambda v.\bot)\ \phi_{\mathcal{E}}$

and $\mathcal{E}_p^{store}[\![p]\!] = force(\mathcal{E}^{store}[\![e]\!]\ (\lambda v.\bot)\ \phi_{\mathcal{E}^{store}}\ (\lambda loc.\text{UNB}))$

then for all $f \in \text{dom}(\phi_{\mathcal{E}^{store}})$, $\sigma_{\mathcal{E}^{store}} \in \text{Store}_{\mathcal{E}^{store}}$:

$$\Phi(\phi_{\mathcal{E}^{store}}[\![f]\!]\ loc_1 \dots loc_n\ \sigma_{\mathcal{E}^{store}}) = \phi_{\mathcal{E}}[\![f]\!]\ (\Phi(loc_1, \sigma_{\mathcal{E}^{store}})) \dots (\Phi(loc_n, \sigma_{\mathcal{E}^{store}}))$$

Proof

The proof is by fixpoint induction.

Base Case

The first approximations to each function variable environment are as follows:

$$\phi^0_{\mathcal{E}} = [(\lambda x_1 \dots \lambda x_{k_j}.\bot)/f_j]$$

$$\phi^0_{\mathcal{E}^{store}} = [(\lambda loc_1 \dots \lambda loc_{k_j}.\lambda\sigma.\bot)/f_j]$$

$$\begin{aligned}
&\Phi(\phi^0_{\mathcal{E}^{store}}[\![f_j]\!]\ loc_1 \dots loc_n\ \sigma_{\mathcal{E}^{store}}) \\
&\quad = \bot \\
&\quad = \phi^0_{\mathcal{E}}[\![f_j]\!]\ (\Phi(loc_1, \sigma_{\mathcal{E}^{store}})) \dots (\Phi(loc_n, \sigma_{\mathcal{E}^{store}}))
\end{aligned}$$

Inductive Case

$$\phi^{n+1}_{\mathcal{E}} = [(\lambda x_1 \dots \lambda x_{k_j}.\mathcal{E}[\![e_j]\!]\ [x_1/v_{j1}, \dots, x_{k_j}/v_{jk_j}]\ \phi^n_{\mathcal{E}})/f_j]$$

where f_j is defined by $f_j\ v_{j1} \dots v_{jk_j} = e_j$

$$\phi^{n+1}_{\mathcal{E}^{store}} = [(\lambda loc_1 \dots \lambda loc_{k_j}.\lambda\sigma.\mathcal{E}^{store}[\![e_j]\!]\ [loc_1/v_{j1}, \dots, loc_{k_j}/v_{jk_j}]\ \phi^n_{\mathcal{E}^{store}}\ \sigma)/f_j]$$

where f_j is defined by $f_j\ v_{j1} \dots v_{jk_j} = e_j$

$$
\begin{aligned}
&\Phi(\phi^{n+1}_{\mathcal{E}^{store}}[\![f_j]\!]\ loc_{j1} \ldots loc_{jk_j}\ \sigma_{\mathcal{E}^{store}}) \\
&\quad = \quad \Phi(\mathcal{E}^{store}[\![e_j]\!]\ [loc_{j1}/v_{j1}, \ldots, loc_{jk_j}/v_{jk_j}]\ \phi^{n}_{\mathcal{E}^{store}}\ \sigma_{\mathcal{E}^{store}}) \\
&\quad = \quad \mathcal{E}[\![e_j]\!]\ [\Phi(loc_{j1}, \sigma_{\mathcal{E}^{store}})/v_{j1}, \ldots, \Phi(loc_{jk_j}, \sigma_{\mathcal{E}^{store}})/v_{jk_j}]\ \phi^{n}_{\mathcal{E}} \\
&\quad \qquad \text{(by inductive hypothesis and Lemma 2.5.2)} \\
&\quad = \quad \phi^{n+1}_{\mathcal{E}}[\![f_j]\!]\ (\Phi(loc_{j1}, \sigma_{\mathcal{E}^{store}})) \ldots (\Phi(loc_{jk_j}, \sigma_{\mathcal{E}^{store}}))
\end{aligned}
$$

□

Appendix B

Proofs for Compile-Time Garbage Detection

B.1 Correctness of Usage Counting Analysis

for all $\rho_{\mathcal{E}^{use}} \in \text{Bve}_{\mathcal{E}^{use}}$, $\phi_{\mathcal{E}^{use}} \in \text{Fve}_{\mathcal{E}^{use}}$, $\sigma_{\mathcal{E}^{use}} \in \text{Store}_{\mathcal{E}^{use}}$, $\phi_{\mathcal{U}} \in \text{Fve}_{\mathcal{U}}$, $p \in \text{Prog}$, $e \in \text{Exp}$:

if $\mathcal{E}_p^{use}[\![p]\!] = (loc''', \sigma'''_{\mathcal{E}^{use}})$

and for all $f \in \text{dom}(\phi_{\mathcal{E}^{use}})$:

  if $\phi_{\mathcal{E}^{use}}[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$

  and $\delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$

  then **if** $\phi_{\mathcal{E}^{use}}[\![f]\!]\ loc'_1 \ldots loc'_n\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$

    and $(\phi_{\mathcal{U}}[\![\mathcal{U} f\#i]\!]\ u) \sqsubseteq \delta(loc'_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

    then $u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

and $\mathcal{E}^{use}[\![e]\!]\ \rho_{\mathcal{E}^{use}}\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$

and $\delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$

then for all $x_i \in \text{dom}(\rho_{\mathcal{E}^{use}})$:

  if $\mathcal{E}^{use}[\![e]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$

  and $(\mathcal{U}[\![e]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

  then $u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

Proof

The proof is by structural induction.

Base Cases

Case 1: $e ::= k$

$\mathcal{U}[\![k]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} = \text{ABS}$

$\mathcal{E}^{use}[\![k]\!]\ \rho\ \phi\ \sigma = alloc((0,k),\sigma)$

$\mathcal{E}^{use}_{p}[\![p]\!] = (loc''', \sigma'''_{\mathcal{E}^{use}})$

if	$\mathcal{E}^{use}[\![e]\!]\ \rho_{\mathcal{E}^{use}}\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$		
and	$\delta(loc',\sigma'_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}}) = u$		
then	**if**	$\mathcal{E}^{use}[\![e]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$	
	and	$(\mathcal{U}[\![e]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$	
	then	$\delta(loc'',\sigma''_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}}) = u$	
		(since no part of x_i appears in the result of e)	

$\Rightarrow\quad u \sqsubseteq \delta(loc'',\sigma''_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}})$

Case 2: $e ::= v$

$$
\begin{aligned}
\mathcal{U}[\![v]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} &= u, && \text{if } v = x\\
&= \text{ABS}, && \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}^{use}[\![v]\!]\ \rho\ \phi\ \sigma &= (loc, \sigma'[loc/\rho[\![v]\!]]), && \text{if } (\sigma\ (\rho[\![v]\!])) \in \text{Closure}\\
&\quad \text{where}\\
&\quad (loc, \sigma') = (\sigma\ (\rho[\![v]\!]))\ \sigma\\
&= ((\sigma\ (\rho[\![v]\!])), \sigma), && \text{otherwise}
\end{aligned}
$$

$\mathcal{E}^{use}_{p}[\![p]\!] = (loc''', \sigma'''_{\mathcal{E}^{use}})$

if	$\mathcal{E}^{use}[\![e]\!]\ \rho_{\mathcal{E}^{use}}\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$
and	$\delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$
then	**if** $\mathcal{E}^{use}[\![e]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$
	and $(\mathcal{U}[\![e]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$
	then $u \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$, if $v = x_i$
	and $\text{ABS} \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$, otherwise
$\Rightarrow$	$u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$, if $v = x_i$
and	$\delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$, otherwise
	(since no part of x_i appears in the result of e)

$\Rightarrow\ u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

Inductive Cases

Case 1: $e ::= b\ e_1 \ldots e_n$

$$\mathcal{U}[\![b\ e_1 \ldots e_n]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} = u \rightarrow (\mathcal{U}[\![e_1]\!][\![x]\!]\ 1\ \phi_{\mathcal{U}}\ \&\ \ldots\ \&\ \mathcal{U}[\![e_n]\!][\![x]\!]\ 1\ \phi_{\mathcal{U}})$$

$$\begin{aligned}\mathcal{E}^{use}[\![b\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma = \mathcal{B}^{use}[\![b]\!]\ & loc_1 \ldots loc_n\ \sigma_n \\ & \text{where} \\ & (loc_1, \sigma_1) = inc(\mathcal{E}^{use}[\![e_1]\!]\ \rho\ \phi\ \sigma) \\ & \quad \vdots \\ & (loc_n, \sigma_n) = inc(\mathcal{E}^{use}[\![e_n]\!]\ \rho\ \phi\ \sigma_{n-1})\end{aligned}$$

$$\mathcal{E}_p^{use}[\![p]\!] = (loc''', \sigma'''_{\mathcal{E}^{use}})$$

if	$\mathcal{E}^{use}[\![e]\!]\ \rho_{\mathcal{E}^{use}}\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$
and	$\delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$
and	$\mathcal{E}^{use}[\![e]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$
and	$(\mathcal{U}[\![e]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$
then	$(u \rightarrow (\mathcal{U}[\![e_1]\!][\![x_i]\!]\ 1\ \phi_{\mathcal{U}}\ \&\ \ldots\ \&\ \mathcal{U}[\![e_n]\!][\![x_i]\!]\ 1\ \phi_{\mathcal{U}})) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

$\Rightarrow$ **if** $\mathcal{E}^{use}[\![e_j]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_j = (loc'_j, \sigma'_j)$
then $(\mathcal{U}[\![e_j]\!][\![x_i]\!]\ 1\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i,\sigma_j,\sigma'''_{\mathcal{E}^{use}})$, if $u \neq$ ABS
$\Rightarrow$ $1 \sqsubseteq \delta(loc'_j,\sigma'_j,\sigma'''_{\mathcal{E}^{use}})$, if $u \neq$ ABS
(by inductive hypothesis)
$\Rightarrow$ $u \sqsubseteq \delta(loc'',\sigma''_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}})$

Case 2: $e ::= c\ e_1 \ldots e_n$

$\mathcal{U}[\![c\ e_1 \ldots e_n]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} = u \rightarrow (\mathcal{U}[\![e_1]\!][\![x]\!]\ u_1\ \phi_{\mathcal{U}}\ \&\ \ldots\ \&\ \mathcal{U}[\![e_n]\!][\![x]\!]\ u_n\ \phi_{\mathcal{U}})$
where
$u_1 = \mathcal{U}c\#1\ u$
$\vdots$
$u_n = \mathcal{U}c\#n\ u$

$\mathcal{E}^{use}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma = \mathcal{C}^{use}[\![c]\!]\ loc_1 \ldots loc_n\ \sigma_n$
where
$(loc_1, \sigma_1) = alloc((\mathcal{E}^{use}[\![e_1]\!]\ \rho\ \phi), \sigma)$
$\vdots$
$(loc_n, \sigma_n) = alloc((\mathcal{E}^{use}[\![e_n]\!]\ \rho\ \phi), \sigma_{n-1})$

$\mathcal{E}^{use}_p[\![p]\!] = (loc'', \sigma''_{\mathcal{E}^{use}})$

if $\mathcal{E}^{use}[\![e]\!]\ \rho_{\mathcal{E}^{use}}\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$
and $\delta(loc',\sigma'_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}}) = u$
and $\mathcal{E}^{use}[\![e]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$
and $(\mathcal{U}[\![e]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i,\sigma_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}})$
then $(u \rightarrow (\mathcal{U}[\![e_1]\!][\![x_i]\!]\ u_1\ \phi_{\mathcal{U}}\ \&\ \ldots\ \&\ \mathcal{U}[\![e_n]\!][\![x_i]\!]\ u_n\ \phi_{\mathcal{U}})) \sqsubseteq \delta(loc_i,\sigma_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}})$
$\Rightarrow$ **if** $\mathcal{E}^{use}[\![e_j]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_j = (loc'_j, \sigma'_j)$
then $(\mathcal{U}[\![e_j]\!][\![x_i]\!]\ u_j\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i,\sigma_j,\sigma'''_{\mathcal{E}^{use}})$, if $u \neq$ ABS
$\Rightarrow$ $u_j \sqsubseteq \delta(loc'_j,\sigma'_j,\sigma'''_{\mathcal{E}^{use}})$, if $u \neq$ ABS
(by inductive hypothesis)
$\Rightarrow$ $u \sqsubseteq \delta(loc'',\sigma''_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}})$

Case 3: $e ::= f\ e_1 \ldots e_n$

$\mathcal{U}[\![f\ e_1 \ldots e_n]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} = u \rightarrow (\mathcal{U}[\![e_1]\!][\![x]\!]\ u_1\ \phi_{\mathcal{U}}\ \&\ \ldots\ \&\ \mathcal{U}[\![e_n]\!][\![x]\!]\ u_n\ \phi_{\mathcal{U}})$

where

$$
\begin{aligned}
u_1 &= \phi_{\mathcal{U}}[\![\mathcal{U}f\#1]\!]\ u \\
&\vdots \\
u_n &= \phi_{\mathcal{U}}[\![\mathcal{U}f\#n]\!]\ u
\end{aligned}
$$

$\mathcal{E}^{use}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi\ \sigma = \phi[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_n$

where

$$
\begin{aligned}
(loc_1, \sigma_1) &= alloc((\mathcal{E}^{use}[\![e_1]\!]\ \rho\ \phi), \sigma) \\
&\vdots \\
(loc_n, \sigma_n) &= alloc((\mathcal{E}^{use}[\![e_n]\!]\ \rho\ \phi), \sigma_{n-1})
\end{aligned}
$$

$\mathcal{E}_p^{use}[\![p]\!] = (loc'', \sigma''_{\mathcal{E}^{use}})$

if $\mathcal{E}^{use}[\![e]\!]\ \rho_{\mathcal{E}^{use}}\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$

and $\delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$

and $\mathcal{E}^{use}[\![e]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$

and $(\mathcal{U}[\![e]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

then $(u \rightarrow (\mathcal{U}[\![e_1]\!][\![x_i]\!]\ u_1\ \phi_{\mathcal{U}}\ \&\ \ldots\ \&\ \mathcal{U}[\![e_n]\!][\![x_i]\!]\ u_n\ \phi_{\mathcal{U}})) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

$\Rightarrow$ **if** $\mathcal{E}^{use}[\![e_j]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_j = (loc'_j, \sigma'_j)$

then $(\mathcal{U}[\![e_j]\!][\![x_i]\!]\ u_j\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i, \sigma_j, \sigma'''_{\mathcal{E}^{use}})$, if $u \neq$ ABS

$\Rightarrow$ $u_j \sqsubseteq \delta(loc'_j, \sigma'_j, \sigma'''_{\mathcal{E}^{use}})$, if $u \neq$ ABS

(by inductive hypothesis)

$\Rightarrow$ $u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

(by assumptions for $\phi_{\mathcal{E}^{use}}$ and $\phi_{\mathcal{U}}$ in Theorem 3.5.2)

Case 4: e ::= **case** e_0 **of** $p_1 : e_1 \mid \ldots \mid p_k : e_k$

$\mathcal{U}[\![$**case** e_0 **of** $p_1 : e_1 \mid \ldots \mid p_k : e_k]\!][\![x]\!]\ u\ \phi_{\mathcal{U}} =$
$u \rightarrow (((\mathcal{U}[\![e_0]\!][\![x]\!]\ u_1\ \phi_{\mathcal{U}})\ \&\ (\mathcal{U}[\![e_1]\!][\![x]\!]\ u\ \phi_{\mathcal{U}})) \sqcup \ldots$
$\sqcup\ ((\mathcal{U}[\![e_0]\!][\![x]\!]\ u_k\ \phi_{\mathcal{U}})\ \&\ (\mathcal{U}[\![e_k]\!][\![x]\!]\ u\ \phi_{\mathcal{U}})))$
where

$$
\begin{array}{lll}
p_1 & = & c_1\ v_{11} \ldots v_{1n_1} \\
\vdots & & \\
p_k & = & c_k\ v_{k1} \ldots v_{kn_k} \\
u_1 & = & \mathcal{U}c_1(1, \mathcal{U}[\![e_1]\!][\![v_{11}]\!]\ u\ \phi_{\mathcal{U}}, \ldots, \mathcal{U}[\![e_1]\!][\![v_{1n_1}]\!]\ u\ \phi_{\mathcal{U}}) \\
\vdots & & \\
u_k & = & \mathcal{U}c_k(1, \mathcal{U}[\![e_k]\!][\![v_{k1}]\!]\ u\ \phi_{\mathcal{U}}, \ldots, \mathcal{U}[\![e_k]\!][\![v_{kn_k}]\!]\ u\ \phi_{\mathcal{U}})
\end{array}
$$

$\mathcal{E}^{use}[\![$**case** e_0 **of** $p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]\ \rho\ \phi\ \sigma =$
$\mathcal{E}^{use}[\![e_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi\ \sigma'$
where

$$
\begin{array}{lll}
(loc, \sigma') & = & inc(\mathcal{E}^{use}[\![e_0]\!]\ \rho\ \phi\ \sigma) \\
(u, x) & = & \sigma'\ loc \\
p_i & = & c\ v_1 \ldots v_n \ \text{ and } match(x, c)
\end{array}
$$

$\mathcal{E}^{use}_p[\![p]\!] = (loc'', \sigma''_{\mathcal{E}^{use}})$

$$
\begin{array}{ll}
\textbf{if} & \mathcal{E}^{use}[\![e]\!]\ \rho_{\mathcal{E}^{use}}\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}}) \\
\textbf{and} & \delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u \\
\textbf{and} & \mathcal{E}^{use}[\![e]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}}) \\
\textbf{and} & (\mathcal{U}[\![e]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) \\
\textbf{then} & u \rightarrow (((\mathcal{U}[\![e_0]\!][\![x_i]\!]\ u_1\ \phi_{\mathcal{U}})\ \&\ (\mathcal{U}[\![e_1]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}})) \sqcup \ldots \\
 & \sqcup\ ((\mathcal{U}[\![e_0]\!][\![x_i]\!]\ u_k\ \phi_{\mathcal{U}})\ \&\ (\mathcal{U}[\![e_k]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}}))) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})
\end{array}
$$

$\Rightarrow$ **if** the branch $p_j : e_j$ is selected

and $\mathcal{E}^{use}[\![e_0]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc'_0, \sigma'_0)$

and $\mathcal{E}^{use}[\![e_j]\!]\ [loc_i/x_i]\ \phi_{\mathcal{E}^{use}}\ \sigma'_0 = (loc'_j, \sigma'_j)$

then $(\mathcal{U}[\![e_0]\!][\![x_i]\!]\ u_j\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

and $(\mathcal{U}[\![e_j]\!][\![x_i]\!]\ u\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc_i, \sigma'_0, \sigma'''_{\mathcal{E}^{use}})$

$\Rightarrow$ $u_j \sqsubseteq \delta(loc'_0, \sigma'_0, \sigma'''_{\mathcal{E}^{use}})$

and $u \sqsubseteq \delta(loc'_j, \sigma'_j, \sigma'''_{\mathcal{E}^{use}})$

(by inductive hypothesis)

$\Rightarrow$ $u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

□

B.2 Correctness of Usage Counting Analysis Function Variable Environment

for all $p \in$ Prog:

if $\mathcal{E}_p^{use}[\![p]\!] = force(\mathcal{E}[\![e]\!]\ (\lambda v.\bot)\ \phi_{\mathcal{E}^{use}}\ (\lambda loc.\text{UNB})) = (loc''', \sigma'''_{\mathcal{E}^{use}})$

and $\mathcal{U}_p[\![p]\!] = \phi_{\mathcal{U}}$

then for all $f \in \text{dom}(\phi_{\mathcal{E}^{use}})$, $\sigma_{\mathcal{E}^{use}} \in \text{Store}_{\mathcal{E}^{use}}$:

  if $\phi_{\mathcal{E}^{use}}[\![f]\!]\ loc_1 \ldots loc_n\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$

  and $\delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$

  then **if** $\phi_{\mathcal{E}^{use}}[\![f]\!]\ loc'_1 \ldots loc'_n\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$

    and $(\phi_{\mathcal{U}}[\![\mathcal{U} f \# i]\!]\ u) \sqsubseteq \delta(loc'_i, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

    then $u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

Proof

The proof is by recursion induction.

Base Case

$\mathcal{E}_p^{use}[\![p]\!] = (loc''', \sigma'''_{\mathcal{E}^{use}})$

if $\phi_{\mathcal{E}^{use}}[\![f_j]\!]\ loc_{j1} \ldots loc_{jk_j}\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$

then $\mathcal{E}^{use}[\![e_j]\!]\ [loc_{j1}/v_{j1}, \ldots, loc_{jk_j}/v_{jk_j}]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc', \sigma'_{\mathcal{E}^{use}})$

where f_j is defined by $f_j\ v_{j1} \ldots v_{jk_j} = e_j$

if $\delta(loc', \sigma'_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}}) = u$

then **if** $\phi_{\mathcal{E}^{use}}[\![f]\!]\ loc'_{j1} \ldots loc'_{jk_j}\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$

  and $(\phi_{\mathcal{U}}[\![\mathcal{U} f_j \# k]\!]\ u) \sqsubseteq \delta(loc'_{jk}, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

  then $\mathcal{E}^{use}[\![e_j]\!]\ [loc'_{j1}/v_{j1}, \ldots, loc'_{jk_j}/v_{jk_j}]\ \phi_{\mathcal{E}^{use}}\ \sigma_{\mathcal{E}^{use}} = (loc'', \sigma''_{\mathcal{E}^{use}})$

  and $(\mathcal{U}[\![e_j]\!][\![v_{jk}]\!]\ u\ \phi_{\mathcal{U}}) \sqsubseteq \delta(loc'_{jk}, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

$\Rightarrow$ $u \sqsubseteq \delta(loc'', \sigma''_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

(by Theorem 3.5.2, since the function f is not recursive)

Inductive Case

$$\phi_{\mathcal{U}}^{n+1} \quad = \quad [(\lambda u.\mathcal{U}[\![e_j]\!][\![v_{jk}]\!]\ u\ \phi_{\mathcal{U}}^{n})/\mathcal{U}f_j\#k]$$

where f_j is defined by $f_j\ v_{j1}\ldots v_{jk_j} = e_j$

$$\phi_{\mathcal{E}^{use}}^{n+1} \quad = \quad [(\lambda loc_{j1}\ldots\lambda loc_{jk_j}.\lambda\sigma_{\mathcal{E}^{use}}.\mathcal{E}^{use}[\![e_j]\!]\ [loc_{j1}/v_{j1},\ldots,loc_{jk_j}/v_{jk_j}]\ \phi_{\mathcal{E}^{use}}^{n}\ \sigma_{\mathcal{E}^{use}})/f_j]$$

where f_j is defined by $f_j\ v_{j1}\ldots v_{jk_j} = e_j$

$$\mathcal{E}_p^{use}[\![p]\!] = (loc''', \sigma'''_{\mathcal{E}^{use}})$$

if $\phi_{\mathcal{E}^{use}}^{n+1}[\![f_j]\!]\ loc_{j1}\ldots loc_{jk_j}\ \sigma_{\mathcal{E}^{use}} = (loc',\sigma'_{\mathcal{E}^{use}})$

then $\mathcal{E}^{use}[\![e_j]\!]\ [loc_{j1}/v_{j1},\ldots,loc_{jk_j}/v_{jk_j}]\ \phi_{\mathcal{E}^{use}}^{n}\ \sigma_{\mathcal{E}^{use}} = (loc',\sigma'_{\mathcal{E}^{use}})$

if $\delta(loc',\sigma'_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}}) = u$

then **if** $\phi_{\mathcal{E}^{use}}^{n+1}[\![f]\!]\ loc'_{j1}\ldots loc'_{jk_j}\ \sigma_{\mathcal{E}^{use}} = (loc'',\sigma''_{\mathcal{E}^{use}})$

and $(\phi_{\mathcal{U}}^{n+1}[\![\mathcal{U}f_j\#k]\!]\ u) \sqsubseteq \delta(loc'_{jk}, \sigma_{\mathcal{E}^{use}}, \sigma'''_{\mathcal{E}^{use}})$

then $\mathcal{E}^{use}[\![e_j]\!]\ [loc'_{j1}/v_{j1},\ldots,loc'_{jk_j}/v_{jk_j}]\ \phi_{\mathcal{E}^{use}}^{n}\ \sigma_{\mathcal{E}^{use}} = (loc'',\sigma''_{\mathcal{E}^{use}})$

and $(\mathcal{U}[\![e_j]\!][\![v_{jk}]\!]\ u\ \phi_{\mathcal{U}}^{n}) \sqsubseteq \delta(loc'_{jk},\sigma_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}})$

$\Rightarrow$ $u \sqsubseteq \delta(loc'',\sigma''_{\mathcal{E}^{use}},\sigma'''_{\mathcal{E}^{use}})$

(by inductive hypothesis and Theorem 3.5.2)

□

Appendix C

Proofs for Compile-Time Garbage Avoidance

C.1 Proof of Deforestation Theorem

C.1.1 Proof of Lemma 5.1.3

Prove: $\mathcal{E}[\![\mathcal{T}[\![e]\!]]\!]\ \rho\ \phi = \mathcal{E}[\![e]\!]\ \rho\ \phi$

The proof is by recursion induction over the transformation rules $\mathcal{T}$.

Base Cases

Case for Rule 1:

$\mathcal{T}[\![k]\!] \quad = \quad k$

Nothing to prove as the expressions are identical.

Case for Rule 2:

$\mathcal{T}[\![v]\!] \quad = \quad v$

Nothing to prove as the expressions are identical.

Inductive Cases

Case for Rule 3:

$\mathcal{T}[\![c\ e_1 \ldots e_n]\!] \quad = \quad c\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]$

$$\begin{array}{rcl}
\mathcal{E}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi & = & \mathcal{C}[\![c]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi)\ \ldots\ (\mathcal{E}[\![e_n]\!]\ \rho\ \phi) \\
& = & \mathcal{C}[\![c]\!]\ (\mathcal{E}[\![\mathcal{T}[\![e_1]\!]]\!]\ \rho\ \phi)\ \ldots\ (\mathcal{E}[\![\mathcal{T}[\![e_n]\!]]\!]\ \rho\ \phi) \\
& & \text{(by inductive hypothesis)} \\
& = & \mathcal{E}[\![c\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]]\!]\ \rho\ \phi \\
\Rightarrow\ \mathcal{E}[\![\mathcal{T}[\![c\ e_1 \ldots e_n]\!]]\!]\ \rho\ \phi & = & \mathcal{E}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \phi
\end{array}$$

Case for Rule 4:

$$\begin{array}{rcl}
\mathcal{T}[\![f\ e_1 \ldots e_n]\!] & = & f'\ v'_1 \ldots v'_k \\
\text{where} & & \\
f'\ v'_1 \ldots v'_k & = & \mathcal{T}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!] \\
& & \text{where } f \text{ is defined by } f\ v_1 \ldots v_n = e
\end{array}$$

and $v'_1 \ldots v'_k$ are the free variables in $(f\ e_1 \ldots e_n)$

$$\begin{array}{rcl}
\mathcal{E}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi & = & \phi[\![f]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi)\ \ldots\ (\mathcal{E}[\![e_n]\!]\ \rho\ \phi) \\
& = & \mathcal{E}[\![e]\!]\ [(\mathcal{E}[\![e_1]\!]\ \rho\ \phi)/v_1,\ldots,(\mathcal{E}[\![e_n]\!]\ \rho\ \phi)/v_n]\ \phi \\
& = & \mathcal{E}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!]\ \rho\ \phi \\
& = & \mathcal{E}[\![\mathcal{T}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!]]\!]\ \rho\ \phi \\
& & \text{(by inductive hypothesis)} \\
& = & \mathcal{E}[\![f'\ v'_1 \ldots v'_k]\!]\ \rho\ \phi \\
& & \text{where} \\
& & f'\ v'_1 \ldots v'_k = \mathcal{T}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!] \\
& & \text{and } v'_1 \ldots v'_k \text{ are the free variables in } (f\ e_1 \ldots e_n) \\
\Rightarrow\ \mathcal{E}[\![\mathcal{T}[\![f\ e_1 \ldots e_n]\!]]\!]\ \rho\ \phi & = & \mathcal{E}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi
\end{array}$$

Case for Rule 5:

$$\begin{array}{l}
\mathcal{T}[\![\mathbf{case}\ v\ \mathbf{of}\ p'_1 : e'_1\ | \ldots |\ p'_k : e'_k]\!] \\
\quad = \quad \mathbf{case}\ v\ \mathbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!]\ | \ldots |\ p'_k : \mathcal{T}[\![e'_k]\!]
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\mathbf{case}\ v\ \mathbf{of}\ p'_1 : e'_1\ | \ldots |\ p'_k : e'_k]\!]\ \rho\ \phi \\
\quad = \quad \mathcal{E}[\![e'_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi \\
\qquad\quad \text{where} \\
\qquad\quad x \ = \ \mathcal{E}[\![v]\!]\ \rho\ \phi \\
\qquad\quad p'_i \ = \ c\ v_1 \ldots v_n \ \text{ and } \mathit{match}(x, c)
\end{array}$$

$$= \mathcal{E}[\![\mathcal{T}[\![e'_i]\!]]\!] \ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n] \ \phi$$

where

$$x = \mathcal{E}[\![v]\!] \ \rho \ \phi$$

$$p'_i = c \ v_1 \ldots v_n \ \text{ and } \mathit{match}(x, c)$$

(by inductive hypothesis)

$$= \mathcal{E}[\![\textbf{case} \ v \ \textbf{of} \ p'_1 : \mathcal{T}[\![e'_1]\!] \ | \ldots | \ p'_k : \mathcal{T}[\![e'_k]\!]]\!] \ \rho \ \phi$$

$$\Rightarrow \mathcal{E}[\![\mathcal{T}[\![\textbf{case} \ v \ \textbf{of} \ p'_1 : e'_1 \ | \ldots | \ p'_k : e'_k]\!]]\!] \ \rho \ \phi$$

$$= \mathcal{E}[\![\textbf{case} \ v \ \textbf{of} \ p'_1 : e'_1 \ | \ldots | \ p'_k : e'_k]\!] \ \rho \ \phi$$

Case for Rule 6:

$$\mathcal{T}[\![\textbf{case} \ (c \ e_1 \ldots e_n) \ \textbf{of} \ p'_1 : e'_1 \ | \ldots | \ p'_k : e'_k]\!]$$

$$= \mathcal{T}[\![e'_i[e_1/v_1, \ldots, e_n/v_n]]\!]$$

where $p'_i = c \ v_1 \ldots v_n$

$$\mathcal{E}[\![\textbf{case} \ (c \ e_1 \ldots e_n) \ \textbf{of} \ p'_1 : e'_1 \ | \ldots | \ p'_k : e'_k]\!] \ \rho \ \phi$$

$$= \mathcal{E}[\![e'_i]\!] \ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n] \ \phi$$

where

$$x = \mathcal{E}[\![c \ e_1 \ldots e_n]\!] \ \rho \ \phi$$

$$p'_i = c \ v_1 \ldots v_n \ \text{ and } \mathit{match}(x, c)$$

$$= \mathcal{E}[\![e'_i]\!] \ \rho[(\mathcal{E}[\![e_1]\!] \ \rho \ \phi)/v_1, \ldots, (\mathcal{E}[\![e_n]\!] \ \rho \ \phi)/v_n] \ \phi$$

$$= \mathcal{E}[\![e'_i[e_1/v_1, \ldots, e_n/v_n]]\!] \ \rho \ \phi$$

$$= \mathcal{E}[\![\mathcal{T}[\![e'_i[e_1/v_1, \ldots, e_n/v_n]]\!]]\!] \ \rho \ \phi$$

(by inductive hypothesis)

$$\Rightarrow \mathcal{E}[\![\mathcal{T}[\![\textbf{case} \ (c \ e_1 \ldots e_n) \ \textbf{of} \ p'_1 : e'_1 \ | \ldots | \ p'_k : e'_k]\!]]\!] \ \rho \ \phi$$

$$= \mathcal{E}[\![\textbf{case} \ (c \ e_1 \ldots e_n) \ \textbf{of} \ p'_1 : e'_1 \ | \ldots | \ p'_k : e'_k]\!] \ \rho \ \phi$$

Case for Rule 7:

$$\mathcal{T}[\![\textbf{case} \ (f \ e_1 \ldots e_n) \ \textbf{of} \ p'_1 : e'_1 \ | \ldots | \ p'_k : e'_k]\!] = f' \ v'_1 \ldots v'_k$$

where

$$f' \ v'_1 \ldots v'_k = \mathcal{T}[\![\textbf{case} \ (e[e_1/v_1, \ldots, e_n/v_n]) \ \textbf{of} \ p'_1 : e'_1 \ | \ldots | \ p'_k : e'_k]\!]$$

where f is defined by $f \ v_1 \ldots v_n = e$

and $v'_1 \ldots v'_k$ are the free variables in (**case** $(f \ e_1 \ldots e_n)$ **of** $p'_1 : e'_1 \ | \ldots | \ p'_k : e'_k$)

$$
\begin{array}{ll}
& \mathcal{E}[\![\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho\ \phi \\
= & \mathcal{E}[\![e'_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi \\
& \text{where} \\
& x \ = \ \mathcal{E}[\![f\ e_1 \ldots e_n]\!]\ \rho\ \phi \\
& p'_i \ = \ c\ v_1 \ldots v_n \ \text{ and } \mathit{match}(x, c) \\
= & \mathcal{E}[\![e'_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi \\
& \text{where} \\
& x \ = \ \phi[\![f]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{E}[\![e_n]\!]\ \rho\ \phi) \\
& p'_i \ = \ c\ v_1 \ldots v_n \ \text{ and } \mathit{match}(x, c) \\
= & \mathcal{E}[\![e'_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi \\
& \text{where} \\
& x \ = \ \mathcal{E}[\![e]\!]\ [(\mathcal{E}[\![e_1]\!]\ \rho\ \phi)/v_1, \ldots, (\mathcal{E}[\![e_n]\!]\ \rho\ \phi)/v_n]\ \phi \\
& p'_i \ = \ c\ v_1 \ldots v_n \ \text{ and } \mathit{match}(x, c) \\
= & \mathcal{E}[\![e'_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi \\
& \text{where} \\
& x \ = \ \mathcal{E}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!]\ \rho\ \phi \\
& p'_i \ = \ c\ v_1 \ldots v_n \ \text{ and } \mathit{match}(x, c) \\
= & \mathcal{E}[\![\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho\ \phi \\
= & \mathcal{E}[\![\mathcal{T}[\![\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!]\ \rho\ \phi \\
& \text{(by inductive hypothesis)} \\
= & \mathcal{E}[\![f'\ v'_1 \ldots v'_k]\!]\ \rho\ \phi \\
& \text{where} \\
& f'\ v'_1 \ldots v'_k = \mathcal{T}[\![\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
& \text{and } v'_1 \ldots v'_k \text{ are the free variables in } (\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \\
\Rightarrow & \mathcal{E}[\![\mathcal{T}[\![\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!]\ \rho\ \phi \\
= & \mathcal{E}[\![\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho\ \phi
\end{array}
$$

Case for Rule 8:

$$
\begin{array}{ll}
& \mathcal{T}[\![\textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_n : e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
= & \mathcal{T}[\![\textbf{case}\ e_0\ \textbf{of} \\
& \qquad p_1 \ : \ \textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\
& \qquad \vdots \\
& \qquad p_n \ : \ \textbf{case}\ e_n\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]
\end{array}
$$

$$
\begin{array}{ll}
& \mathcal{E}[\![\textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_n : e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho\ \phi \\
= & \mathcal{E}[\![e'_i]\!]\ \rho[x' \downarrow 1/v'_1, \ldots, x' \downarrow m/v'_m]\ \phi \\
& \text{where} \\
& \quad x' = \mathcal{E}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_n : e_n]\!]\ \rho\ \phi \\
& \quad p'_i = c'\ v'_1 \ldots v'_m\ \text{ and } \mathit{match}(x', c') \\
= & \mathcal{E}[\![e'_i]\!]\ \rho[x' \downarrow 1/v'_1, \ldots, x' \downarrow m/v'_m]\ \phi \\
& \text{where} \\
& \quad x' = \mathcal{E}[\![e_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow l/v_l]\ \phi \\
& \quad p'_i = c'\ v'_1 \ldots v'_m\ \text{ and } \mathit{match}(x', c') \\
& \quad x = \mathcal{E}[\![e_0]\!]\ \rho\ \phi \\
& \quad p_i = c\ v_1 \ldots v_l\ \text{ and } \mathit{match}(x, c) \\
= & \mathcal{E}[\![\textbf{case}\ e_i\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow l/v_l]\ \phi \\
& \text{where} \\
& \quad x = \mathcal{E}[\![e_0]\!]\ \rho\ \phi \\
& \quad p_i = c\ v_1 \ldots v_l\ \text{ and } \mathit{match}(x, c) \\
& \text{(since there is no nameclash between the variables in the patterns } p_1 \ldots p_n \\
& \text{and the free variables in the expressions } e'_1 \ldots e'_k) \\
= & \mathcal{E}[\![\textbf{case}\ e_0\ \textbf{of} \\
& \quad p_1 : \textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\
& \quad \vdots \\
& \quad p_n : \textbf{case}\ e_n\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho\ \phi \\
= & \mathcal{E}[\![\mathcal{T}[\![\textbf{case}\ e_0\ \textbf{of} \\
& \qquad p_1 : \textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\
& \qquad \vdots \\
& \qquad p_n : \textbf{case}\ e_n\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!]\ \rho\ \phi \\
& \text{(by inductive hypothesis)} \\
\Rightarrow & \mathcal{E}[\![\mathcal{T}[\![\textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_n : e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!]\ \rho\ \phi \\
= & \mathcal{E}[\![\textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_n : e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho\ \phi
\end{array}
$$

□

C.1.2 Proof of Lemma 5.1.4

Prove: $\mathcal{T}[\![e]\!] \in tf$

The proof is by recursion induction over the transformation rules $\mathcal{T}$.

Base Cases

Case for Rule 1:

$\mathcal{T}[\![k]\!] \quad = \quad k$

Nothing to prove as $k \in tf$

Case for Rule 2:

$\mathcal{T}[\![v]\!] \quad = \quad v$

Nothing to prove as $v \in tf$

Inductive Cases

Case for Rule 3:

$$\mathcal{T}[\![c\ e_1 \ldots e_n]\!] \quad = \quad c\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]$$

$\mathcal{T}[\![e_i]\!] \in tf, \forall\ i \in \{1 \ldots n\}$
(by inductive hypothesis)
$\Rightarrow \quad (c\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]) \in tf$
$\Rightarrow \quad \mathcal{T}[\![c\ e_1 \ldots e_n]\!] \in tf$

Case for Rule 4:

$\mathcal{T}[\![f\ e_1 \ldots e_n]\!] \quad = \quad f'\ v'_1 \ldots v'_k$
where
$f'\ v'_1 \ldots v'_k \quad = \quad \mathcal{T}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!]$
where f is defined by $f\ v_1 \ldots v_n = e$
and $v'_1 \ldots v'_k$ are the free variables in $(f\ e_1 \ldots e_n)$

$(f'\ v'_1 \ldots v'_k) \in tf$

$(\mathcal{T}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!]) \in tf$

(by inductive hypothesis)

$\Rightarrow \quad \mathcal{T}[\![f\ e_1 \ldots e_n]\!] \in tf$

Case for Rule 5:

$$\mathcal{T}[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] = \textbf{case}\ v\ \textbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!]$$

$\mathcal{T}[\![e'_i]\!] \in tf, \forall\ i \in \{1 \ldots k\}$

(by inductive hypothesis)

$\Rightarrow \quad (\textbf{case}\ v\ \textbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!]) \in tf$

$\Rightarrow \quad \mathcal{T}[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \in tf$

Case for Rule 6:

$$\mathcal{T}[\![\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] = \mathcal{T}[\![e'_i[e_1/v_1, \ldots, e_n/v_n]]\!]$$

where $p'_i = c\ v_1 \ldots v_n$

$(\mathcal{T}[\![e'_i[e_1/v_1, \ldots, e_n/v_n]]\!]) \in tf$

(by inductive hypothesis)

$\Rightarrow \quad \mathcal{T}[\![\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \in tf$

Case for Rule 7:

$\mathcal{T}[\![\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] = f'\ v'_1 \ldots v'_k$

where

$f'\ v'_1 \ldots v'_k \quad = \quad \mathcal{T}[\![\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$

where f is defined by $f\ v_1 \ldots v_n = e$

and $v'_1 \ldots v'_k$ are the free variables in $(\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k)$

$f'\ v'_1 \ldots v'_k \in tf$

$(\mathcal{T}[\![\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]) \in tf$

(by inductive hypothesis)

$\Rightarrow \quad \mathcal{T}[\![\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \in tf$

Case for Rule 8:

$$\mathcal{T}[\![\textbf{case } (\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$

$$\begin{aligned} = \quad & \mathcal{T}[\![\textbf{case } e_0 \textbf{ of} \\ & \qquad p_1 \; : \; \textbf{case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\ & \qquad \vdots \\ & \qquad p_n \; : \; \textbf{case } e_n \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \end{aligned}$$

$$\begin{aligned} & (\mathcal{T}[\![\textbf{case } e_0 \textbf{ of} \\ & \qquad p_1 \; : \; \textbf{case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\ & \qquad \vdots \\ & \qquad p_n \; : \; \textbf{case } e_n \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]) \; \in \; \mathit{tf} \end{aligned}$$

(by inductive hypothesis)

$$\Rightarrow \quad \mathcal{T}[\![\textbf{case } (\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \in \mathit{tf}$$

□

C.1.3 Proof of Lemma 5.1.5

Assume $\mathcal{R}[\![e]\!]$ is a measure of the number of steps required to reduce the expression e to a fully forced form. One expression is considered to be more efficient than another if, for every possible instantiation of the free variables, the first requires fewer steps to reduce than the second.

Prove: $\mathcal{R}[\![\mathcal{T}[\![e]\!]]\!] \leq \mathcal{R}[\![e]\!]$

The proof is by recursion induction over the transformation rules $\mathcal{T}$.

Base Cases

Case for Rule 1:

$$\mathcal{T}[\![k]\!] \quad = \quad k$$

Nothing to prove as the expressions are identical.

Case for Rule 2:

$$\mathcal{T}[\![v]\!] \quad = \quad v$$

Nothing to prove as the expressions are identical.

Inductive Cases

Case for Rule 3:

$$\begin{array}{lll} & \mathcal{T}[\![c\ e_1 \ldots e_n]\!] & = \ c\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!] \\ & \mathcal{R}[\![\mathcal{T}[\![e_i]\!]]\!] & \leq \ \mathcal{R}[\![e_i]\!],\ \forall\ i \in \{1 \ldots n\} \\ & & \quad \text{(by inductive hypothesis)} \\ \Rightarrow & \mathcal{R}[\![c\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]]\!] & \leq \ \mathcal{R}[\![c\ e_1 \ldots e_n]\!] \\ \Rightarrow & \mathcal{R}[\![\mathcal{T}[\![c\ e_1 \ldots e_n]\!]]\!] & \leq \ \mathcal{R}[\![c\ e_1 \ldots e_n]\!] \end{array}$$

Case for Rule 4:

$$\mathcal{T}[\![f\ e_1 \ldots e_n]\!] \quad = \quad f'\ v'_1 \ldots v'_k$$

where

$$f'\ v'_1 \ldots v'_k \quad = \quad \mathcal{T}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!]$$

where f is defined by $f\ v_1 \ldots v_n = e$

and $v'_1 \ldots v'_k$ are the free variables in $(f\ e_1 \ldots e_n)$

$$\begin{array}{llcl}
 & \mathcal{R}[\![e[e_1/v_1,\ldots,e_n/v_n]]\!] & < & \mathcal{R}[\![f\ e_1\ldots e_n]\!] \\
 & & & \text{(since } e \text{ is linear in all variables and} \\
 & & & \text{a function call has been removed)} \\
 & \mathcal{R}[\![\mathcal{T}[\![e[e_1/v_1,\ldots,e_n/v_n]]\!]]\!] & \leq & \mathcal{R}[\![e[e_1/v_1,\ldots,e_n/v_n]]\!] \\
 & & & \text{(by inductive hypothesis)} \\
\Rightarrow & \mathcal{R}[\![\mathcal{T}[\![e[e_1/v_1,\ldots,e_n/v_n]]\!]]\!] & < & \mathcal{R}[\![f\ e_1\ldots e_n]\!] \\
\Rightarrow & \mathcal{R}[\![f'\ v'_1\ldots v'_k]\!] & \leq & \mathcal{R}[\![f\ e_1\ldots e_n]\!] \\
 & & & \text{(since a function call is introduced only when} \\
 & & & \text{another function call has been removed)} \\
\Rightarrow & \mathcal{R}[\![\mathcal{T}[\![f\ e_1\ldots e_n]\!]]\!] & \leq & \mathcal{R}[\![f\ e_1\ldots e_n]\!]
\end{array}$$

Case for Rule 5:

$$\begin{array}{l}
\mathcal{T}[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
\qquad = \quad \textbf{case}\ v\ \textbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!]
\end{array}$$

$$\begin{array}{lll}
 & \mathcal{R}[\![\mathcal{T}[\![e'_i]\!]]\!] \quad \leq & \mathcal{R}[\![e'_i]\!],\ \forall\ i \in \{1\ldots k\} \\
 & & \text{(by inductive hypothesis)} \\
\Rightarrow & \mathcal{R}[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!]]\!] & \\
 & \qquad \leq & \mathcal{R}[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
\Rightarrow & \mathcal{R}[\![\mathcal{T}[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] & \\
 & \qquad \leq & \mathcal{R}[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]
\end{array}$$

Case for Rule 6:

$$\begin{array}{l}
\mathcal{T}[\![\textbf{case}\ (c\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
\quad = \quad \mathcal{T}[\![e'_i[e_1/v_1,\ldots,e_n/v_n]]\!] \\
\qquad\quad \text{where } p'_i = c\ v_1\ldots v_n
\end{array}$$

$$\begin{array}{ll}
 & \mathcal{R}[\![e'_i[e_1/v_1,\ldots,e_n/v_n]]\!] \\
 & \quad < \quad \mathcal{R}[\![\textbf{case}\ (c\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
 & \qquad \text{(since } e'_i \text{ is linear in all variables and a} \\
 & \qquad \text{constructor application has been removed)} \\
 & \mathcal{R}[\![\mathcal{T}[\![e'_i[e_1/v_1,\ldots,e_n/v_n]]\!]]\!] \\
 & \quad \leq \quad \mathcal{R}[\![e'_i[e_1/v_1,\ldots,e_n/v_n]]\!] \\
 & \qquad \text{(by inductive hypothesis)} \\
\Rightarrow & \mathcal{R}[\![\mathcal{T}[\![e'_i[e_1/v_1,\ldots,e_n/v_n]]\!]]\!] \\
 & \quad < \quad \mathcal{R}[\![\textbf{case}\ (c\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
\Rightarrow & \mathcal{R}[\![\mathcal{T}[\![\textbf{case}\ (c\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \\
 & \quad \leq \quad \mathcal{R}[\![\textbf{case}\ (c\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]
\end{array}$$

Case for Rule 7:

$$\begin{array}{l}
\mathcal{T}[\![\textbf{case}\ (f\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] = f'\ v'_1\ldots v'_k \\
\text{where} \\
f'\ v'_1\ldots v'_k \quad = \quad \mathcal{T}[\![\textbf{case}\ (e[e_1/v_1,\ldots,e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
\qquad\qquad\qquad \text{where } f \text{ is defined by } f\ v_1\ldots v_n = e \\
\text{and } v'_1\ldots v'_k \text{ are the free variables in } (\textbf{case}\ (f\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k)
\end{array}$$

$$\begin{array}{ll}
 & \mathcal{R}[\![\textbf{case}\ (e[e_1/v_1,\ldots,e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
 & \quad < \quad \mathcal{R}[\![\textbf{case}\ (f\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
 & \qquad \text{(since } e \text{ is linear in all variables and} \\
 & \qquad \text{a function call has been removed)} \\
 & \mathcal{R}[\![\mathcal{T}[\![\textbf{case}\ (e[e_1/v_1,\ldots,e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \\
 & \quad \leq \quad \mathcal{R}[\![\textbf{case}\ (e[e_1/v_1,\ldots,e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
 & \qquad \text{(by inductive hypothesis)} \\
\Rightarrow & \mathcal{R}[\![\mathcal{T}[\![\textbf{case}\ (e[e_1/v_1,\ldots,e_n/v_n]\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \\
 & \quad < \quad \mathcal{R}[\![\textbf{case}\ (f\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
\Rightarrow & \mathcal{R}[\![f'\ v'_1\ldots v'_k]\!] \\
 & \quad \leq \quad \mathcal{R}[\![\textbf{case}\ (f\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
 & \qquad \text{(since a function call is introduced only when} \\
 & \qquad \text{another function call has been removed)} \\
\Rightarrow & \mathcal{R}[\![\mathcal{T}[\![\textbf{case}\ (f\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \\
 & \quad \leq \quad \mathcal{R}[\![\textbf{case}\ (f\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]
\end{array}$$

Case for Rule 8:

$$\mathcal{T}[\![\textbf{case } (\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$
$$\begin{aligned} = \quad & \mathcal{T}[\![\textbf{case } e_0 \textbf{ of} \\ & \quad p_1 \;:\; \textbf{case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\ & \quad \vdots \\ & \quad p_n \;:\; \textbf{case } e_n \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \end{aligned}$$

$$\begin{aligned} & \mathcal{R}[\![\textbf{case } e_0 \textbf{ of} \\ & \quad p_1 \;:\; \textbf{case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\ & \quad \vdots \\ & \quad p_n \;:\; \textbf{case } e_n \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\ & \leq \quad \mathcal{R}[\![\textbf{case } (\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \end{aligned}$$

$$\begin{aligned} & \mathcal{R}[\![\mathcal{T}[\![\textbf{case } e_0 \textbf{ of} \\ & \quad p_1 \;:\; \textbf{case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\ & \quad \vdots \\ & \quad p_n \;:\; \textbf{case } e_n \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \\ & \leq \quad \mathcal{R}[\![\textbf{case } e_0 \textbf{ of} \\ & \qquad p_1 \;:\; \textbf{case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\ & \qquad \vdots \\ & \qquad p_n \;:\; \textbf{case } e_n \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \end{aligned}$$

(by inductive hypothesis)

$$\begin{aligned} \Rightarrow \quad & \mathcal{R}[\![\mathcal{T}[\![\textbf{case } e_0 \textbf{ of} \\ & \quad p_1 \;:\; \textbf{case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\ & \quad \vdots \\ & \quad p_n \;:\; \textbf{case } e_n \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \\ & \leq \quad \mathcal{R}[\![\textbf{case } (\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\ \Rightarrow \quad & \mathcal{R}[\![\mathcal{T}[\![\textbf{case } (\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \\ & \leq \quad \mathcal{R}[\![\textbf{case } (\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_n : e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \end{aligned}$$

□

C.1.4 Proof of Lemma 5.1.10

Prove: $\forall\ e \in dg^s(x,y),\ x \leq s,\ y \leq n$:

$$\mathcal{T}[\![e]\!] = \ldots \mathcal{T}[\![e']\!] \ldots \Rightarrow e' \in dg^s(s,n)$$

The proof is by inspection of the transformation rules $\mathcal{T}$

Case for Rule 1:

$\mathcal{T}[\![k]\!] \quad = \quad k$

Nothing to prove.

Case for Rule 2:

$\mathcal{T}[\![v]\!] \quad = \quad v$

Nothing to prove.

Case for Rule 3:

$$\mathcal{T}[\![c\ e_1 \ldots e_n]\!] \quad = \quad c\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]$$

$(c\ e_1 \ldots e_n) \in dg^s(x,y),\ x \leq s,\ y \leq n$

$\Rightarrow \quad e_i \in dg^s(x-1,y),\ \forall\ i \in \{1 \ldots n\}$

$\Rightarrow \quad e_i \in dg^s(s,n),\ \forall\ i \in \{1 \ldots n\}$

(since $x \leq s,\ y \leq n$)

Case for Rule 4:

$$\mathcal{T}[\![f\ e_1 \ldots e_n]\!] \quad = \quad f'\ v'_1 \ldots v'_k$$

where

$$f'\ v'_1 \ldots v'_k \quad = \quad \mathcal{T}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!]$$

where f is defined by $f\ v_1 \ldots v_n = e$

and $v'_1 \ldots v'_k$ are the free variables in $(f\ e_1 \ldots e_n)$

$(f\ e_1 \ldots e_n) \in dg^s(x,y),\ x \leq s,\ y \leq n$

$\Rightarrow \quad e_i \in dg^s(0,y),\ \forall\ i \in \{1 \ldots n\}$

and $\quad e \in dg^s(s,1)$

$\Rightarrow \quad e[e_1/v_1, \ldots, e_n/v_n] \in dg^s(s, y)$

$\Rightarrow \quad e[e_1/v_1, \ldots, e_n/v_n] \in dg^s(s, n)$

(since $y \leq n$)

Case for Rule 5:

$$\mathcal{T}[\![\textbf{case } v \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$
$$= \textbf{case } v \textbf{ of } p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!]$$

$(\textbf{case } v \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in dg^s(x, y),\ x \leq s,\ y \leq n$

$\Rightarrow \quad e'_i \in dg^s(x-y, y)\ \forall\ i \in \{1 \ldots k\}$

$\Rightarrow \quad e'_i \in dg^s(s, n)\ \forall\ i \in \{1 \ldots k\}$

(since $x \leq s,\ y \leq n$)

Case for Rule 6:

$$\mathcal{T}[\![\textbf{case } (c\ e_1 \ldots e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$
$$= \mathcal{T}[\![e'_i[e_1/v_1, \ldots, e_n/v_n]]\!]$$

where $p'_i = c\ v_1 \ldots v_n$

$(\textbf{case } (c\ e_1 \ldots e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in dg^s(x, y),\ x \leq s,\ y \leq n$

$\Rightarrow \quad e_i \in dg^s(s-1, y-1),\ \forall\ i \in \{1 \ldots n\}$

and $\quad e'_i \in dg^s(x-y, y),\ \forall\ i \in \{1 \ldots k\}$

$\Rightarrow \quad e'_i[e_1/v_1, \ldots, e_n/v_n] \in dg^s(x-y, y)$

(since $v_1, \ldots, v_n \notin fv$)

$\Rightarrow \quad e'_i[e_1/v_1, \ldots, e_n/v_n] \in dg^s(s, n)$

(since $x \leq s,\ y \leq n$)

Case for Rule 7:

$\mathcal{T}[\![\textbf{case } (f\ e_1 \ldots e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] = f'\ v'_1 \ldots v'_k$

where

$f'\ v'_1 \ldots v'_k \quad = \quad \mathcal{T}[\![\textbf{case } (e[e_1/v_1, \ldots, e_n/v_n]) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$

where f is defined by $f\ v_1 \ldots v_n = e$

and $v'_1 \ldots v'_k$ are the free variables in $(\textbf{case } (f\ e_1 \ldots e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k)$

$$
\begin{array}{ll}
 & (\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in dg^s(x,y),\ x \leq s,\ y \leq n \\
\Rightarrow & e_i \in dg^s(0, y-1),\ \forall\ i \in \{1 \ldots n\} \\
\text{and} & e'_i \in dg^s(x-y, y),\ \forall\ i \in \{1 \ldots k\} \\
\text{and} & e \in dg^s(s, 1) \\
\Rightarrow & (\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in dg^s(x,y) \\
\Rightarrow & (\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in dg^s(s,n) \\
 & (\text{since } x \leq s,\ y \leq n)
\end{array}
$$

Case for Rule 8:

$$
\begin{array}{l}
\mathcal{T}[\![\textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_n : e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
\quad = \ \mathcal{T}[\![\textbf{case}\ e_0\ \textbf{of} \\
\qquad\qquad p_1 \ :\ \textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\
\qquad\qquad \vdots \\
\qquad\qquad p_n \ :\ \textbf{case}\ e_n\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]
\end{array}
$$

$$
\begin{array}{ll}
 & (\textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_n : e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in dg^s(x,y), \\
 & \hfill x \leq s,\ y \leq n \\
\Rightarrow & e_i \in dg^s(s-y+1, y-1),\ \forall\ i \in \{1 \ldots n\} \\
\text{and} & e'_i \in dg^s(x-y, y),\ \forall\ i \in \{1 \ldots k\} \\
\text{and} & e_0 \in dg^s(0, y-1) \\
\Rightarrow & (\textbf{case}\ e_0\ \textbf{of} \\
 & \quad p_1 \ :\ \textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\
 & \quad \vdots \\
 & \quad p_n \ :\ \textbf{case}\ e_n\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in dg^s(x,y) \\
\Rightarrow & (\textbf{case}\ e_0\ \textbf{of} \\
 & \quad p_1 \ :\ \textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\
 & \quad \vdots \\
 & \quad p_n \ :\ \textbf{case}\ e_n\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in dg^s(s,n) \\
 & (\text{since } x \leq s,\ y \leq n)
\end{array}
$$

□

C.1.5 Proof of Lemma 5.1.11

Prove: $\forall\ e \in dg^s(x,y),\ x \geq 0,\ s > 0,\ y > 0$:

$\mathcal{S}[\![e]\!] \leq x + (s \times (y-1))$

(Lemma 5.1.11 is a corollary of this)

The proof is by induction on the variable y.

Base Cases: $y = 1$

The proof of the base cases is by induction on the variable x.

Base Cases: $x = 0$

Case 1: $dg^s(x,y) ::= k$ if $x \geq 0$ and $y > 0$

$$\begin{aligned} \mathcal{S}[\![k]\!] &= 0 \\ &\leq x + (s \times (y-1)) \\ &\quad \text{(since } x = 0,\ y = 1 \text{ and } s > 0) \end{aligned}$$

Case 2: $dg^s(x,y) ::= v$ if $x \geq 0$ and $y > 0$

$$\begin{aligned} \mathcal{S}[\![v]\!] &= 0 \\ &\leq x + (s \times (y-1)) \\ &\quad \text{(since } x = 0,\ y = 1 \text{ and } s > 0) \end{aligned}$$

Inductive Cases: $x > 0$

Case 1: $dg^s(x,y) ::= c\ dg^s_1(x-1,y) \ldots dg^s_n(x-1,y)$ if $x > 0$ and $y > 0$

$$\begin{aligned} \mathcal{S}[\![c\ e_1 \ldots e_n]\!] &= 1 + max(\mathcal{S}[\![e_1,\ \ldots,\ e_n]\!]) \\ &\leq 1 + (x-1) + (s \times (y-1)), \text{ if } (c\ e_1 \ldots e_n) \in dg^s(x,y) \\ &\quad \text{(by inductive hypothesis for } x) \\ &\leq x + (s \times (y-1)) \end{aligned}$$

Case 2: $dg^s(x,y) ::= f\ dg_1^s(0,y) \ldots dg_n^s(0,y)$ if $x > 0$ and $y > 0$
where f is defined by $f\ v_1 \ldots v_n = e$ and $e \in dg^s(s,1)$

$$\begin{array}{rcl}\mathcal{S}[\![f\ e_1 \ldots e_n]\!] & = & 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\ & = & 1, \text{ if } (f\ e_1 \ldots e_n) \in dg^s(x,y) \\ & \leq & x + (s \times (y-1)) \\ & & \text{(since } x > 0, y = 1) \end{array}$$

Case 3: $dg^s(x,y) ::= \textbf{case}\ dg_0^s(0,y)\ \textbf{of}\ p_1 : dg_1^s(x-y,y) \mid \ldots \mid p_k : dg_k^s(x-y,y)$
if $x > 0$ and $y > 0$

$$\begin{array}{ll}\mathcal{S}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!] & \\ \quad = & 1 + max(\mathcal{S}[\![e_0]\!], \ldots, \mathcal{S}[\![e_k]\!]) \\ \quad \leq & 1 + ((x-y) + (s \times (y-1)))), \\ & \quad \text{if } (\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k) \in dg^s(x,y) \\ & \text{(by inductive hypothesis for } x) \\ \quad \leq & x + (s \times (y-1)) \\ \text{(since } y = 1) & \end{array}$$

Case 4: $dg^s(x,y) ::= dg^s(x-1,y)$ if $x > 0$ and $y > 0$

$$\begin{array}{rcl}\mathcal{S}[\![e]\!] & \leq & x - 1 + (s \times (y-1)), \forall\ e \in dg^s(x-1,y) \\ & & \text{(by inductive hypothesis for } x) \\ & \leq & x + (s \times (y-1)) \end{array}$$

Inductive Cases: $y > 1$

The proof of the inductive cases is by induction on the variable x.

Base Cases: $x = 0$

Case 1: $dg^s(x,y) ::= k$ if $x \geq 0$ and $y > 0$

$$\begin{aligned}
\mathcal{S}[\![k]\!] &= 0 \\
&\leq x + (s \times (y-1)) \\
&\quad \text{(since } x = 0,\ y > 1 \text{ and } s > 0\text{)}
\end{aligned}$$

Case 2: $dg^s(x,y) ::= v$ if $x \geq 0$ and $y > 0$

$$\begin{aligned}
\mathcal{S}[\![v]\!] &= 0 \\
&\leq x + (s \times (y-1)) \\
&\quad \text{(since } x = 0,\ y > 0 \text{ and } s > 0\text{)}
\end{aligned}$$

Inductive Cases: $x > 0$

Case 1: $dg^s(x,y) ::= c\ dg^s_1(x-1,y) \ldots dg^s_n(x-1,y)$ if $x > 0$ and $y > 0$

$$\begin{aligned}
\mathcal{S}[\![c\ e_1 \ldots e_n]\!] &= 1 + max(\mathcal{S}[\![e_1, \ldots, e_n]\!]) \\
&\leq 1 + (x-1) + (s \times (y-1)),\ \text{if } (c\ e_1 \ldots e_n) \in dg^s(x,y) \\
&\quad \text{(by inductive hypothesis for } x\text{)} \\
&\leq x + (s \times (y-1))
\end{aligned}$$

Case 2: $dg^s(x,y) ::= f\ dg^s_1(0,y) \ldots dg^s_n(0,y)$ if $x > 0$ and $y > 0$
where f is defined by $f\ v_1 \ldots v_n = e$ and $e \in dg^s(s,1)$

$$\begin{aligned}
\mathcal{S}[\![f\ e_1 \ldots e_n]\!] &= 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
&\leq 1 + s + (s \times (y-2)),\ \text{if } (f\ e_1 \ldots e_n) \in dg^s(x,y) \\
&\quad \text{(by inductive hypothesis for } y\text{)} \\
&\leq x + (s \times (y-1)) \\
&\quad \text{(since } x > 0\text{)}
\end{aligned}$$

Case 3: $dg^s(x,y) ::=$ **case** $dg^s_0(0,y)$ **of** $p_1 : dg^s_1(x-y,y) \mid \ldots \mid p_k : dg^s_k(x-y,y)$
if $x > 0$ and $y > 0$

$$
\begin{aligned}
&\mathcal{S}[\![\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k]\!] \\
&\quad = \; 1 + max(\mathcal{S}[\![e_0]\!], \ldots, \mathcal{S}[\![e_k]\!]) \\
&\quad \leq \; 1 + max((s + (s \times (y-2))),((x-y) + (s \times (y-1)))), \\
&\qquad\qquad \text{if } (\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k) \in dg^s(x,y) \\
&\qquad \text{(by inductive hypotheses for } x \text{ and } y) \\
&\quad \leq \; x + (s \times (y-1)) \\
&\qquad \text{(since } x > 0,\ y > 1)
\end{aligned}
$$

Case 4: $dg^s(x,y) ::= dg^s(x-1,y)$ if $x > 0$ and $y > 0$

$$
\begin{aligned}
\mathcal{S}[\![e]\!] \;&\leq\; x - 1 + (s \times (y-1)),\ \forall\ e \in dg^s(x-1,y) \\
&\quad \text{(by inductive hypothesis for } x) \\
&\leq\; x + (s \times (y-1))
\end{aligned}
$$

Case 5: $dg^s(x,y) ::= fg^s(s,y-1)$ if $x \geq 0$ and $y > 1$

$$
\begin{aligned}
\mathcal{S}[\![e]\!] \;&\leq\; s + (s \times (y-2)),\ \forall\ e \in fg^s(s,y-1) \\
&\quad \text{(by inductive hypothesis for } y \text{, since } e \in fg^s(s,y-1) \Rightarrow e \in dg^s(s,y-1)) \\
&\leq\; s \times (y-1) \\
&\leq\; x + (s \times (y-1)) \\
&\quad \text{(since } x > 0)
\end{aligned}
$$

□

C.2 Proof of Extended Deforestation Theorem

C.2.1 Proof of Lemma 5.2.12

Prove: $\forall\, e \in edg^{s,n}(x,y,z),\ x \le s,\ y \le f,\ z \le n$:

$$\mathcal{T}[\![e]\!] = \ldots \mathcal{T}[\![e']\!] \ldots \Rightarrow e' \in edg^{s,n}(s,f,n)$$

The proof is by inspection of the transformation rules $\mathcal{T}$

Case for Rule 1:

$\mathcal{T}[\![k]\!] \quad = \quad k$

Nothing to prove.

Case for Rule 2:

$\mathcal{T}[\![v]\!] \quad = \quad v$

Nothing to prove.

Case for Rule 3:

$$\mathcal{T}[\![c\ e_1 \ldots e_n]\!] \quad = \quad c\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]$$

$(c\ e_1 \ldots e_n) \in edg^{s,n}(x,y,z),\ x \le s,\ y \le f,\ z \le n$

$\Rightarrow \quad e_i \in edg^{s,n}(x-1,y,z),\ \forall\, i \in \{1 \ldots n\}$

$\Rightarrow \quad e_i \in edg^{s,n}(s,f,n),\ \forall\, i \in \{1 \ldots n\}$

(since $x \le s,\ y \le f,\ z \le n$)

Case for Rule 4:

$\mathcal{T}[\![f\ e_1 \ldots e_n]\!] \quad = \quad f'\ v'_1 \ldots v'_k$

where

$f'\ v'_1 \ldots v'_k \quad = \quad \mathcal{T}[\![e[e_1/v_1, \ldots, e_n/v_n]]\!]$

where f is defined by $f\ v_1 \ldots v_n = e$

and $v'_1 \ldots v'_k$ are the free variables in $(f\ e_1 \ldots e_n)$

$$
\begin{array}{lll}
 & (f\ e_1 \ldots e_n) \in edg^{s,n}(x,y,z),\ x \leq s,\ y \leq f,\ z \leq n & \\
\Rightarrow & e \in edg^{s,n}(s,0,1) & \\
\text{and} & e_i\ \ \in v, & \text{if } y = f \\
 & \quad \in edg^{s,n}(0,0,z), & \text{if } 0 \leq y < f \text{ and } e_i \text{ is a transient structure} \\
 & \quad \in edg^{s,n}(x-1,y,z), & \text{if } 0 \leq y < f \text{ and } e_i \text{ is not a transient structure} \\
\Rightarrow & (e[e_1/v_1,\ldots,e_n/v_n])\ \in edg^{s,n}(s,0,1), & \text{if } y = f \\
 & \qquad\qquad\qquad\qquad\ \ \in edg^{s,n}(s,y+1,z), & \text{if } 0 \leq y < f
\end{array}
$$

(since only transient structures can be the selectors in **case** expressions)

$$
\begin{array}{ll}
\Rightarrow & (e[e_1/v_1,\ldots,e_n/v_n]) \in edg^{s,n}(s,f,n)
\end{array}
$$

(since $y \leq f,\ z \leq n$)

Case for Rule 5:

$$
\begin{array}{l}
\mathcal{T}[\![\textbf{case}\ v\ \textbf{of}\ p'_1 : e'_1\ |\ldots|\ p'_k : e'_k]\!] \\
\quad = \quad \textbf{case}\ v\ \textbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!]\ |\ldots|\ p'_k : \mathcal{T}[\![e'_k]\!]
\end{array}
$$

$$
\begin{array}{ll}
 & (\textbf{case}\ v\ \textbf{of}\ p'_1 : e'_1\ |\ldots|\ p'_k : e'_k) \in edg^{s,n}(x,y,z),\ x \leq s,\ y \leq f,\ z \leq n \\
\Rightarrow & e'_i \in edg^{s,n}(x-z,y,z),\ \forall\ i \in \{1\ldots k\} \\
\Rightarrow & e'_i \in edg^{s,n}(s,f,n),\ \forall\ i \in \{1\ldots k\}
\end{array}
$$

(since $x \leq s,\ y \leq f,\ z \leq n$)

Case for Rule 6:

$$
\begin{array}{l}
\mathcal{T}[\![\textbf{case}\ (c\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1\ |\ldots|\ p'_k : e'_k]\!] \\
\quad = \quad \mathcal{T}[\![e'_i[e_1/v_1,\ldots,e_n/v_n]]\!] \\
\qquad\quad\ \ \text{where } p'_i = c\ v_1\ldots v_n
\end{array}
$$

$$
\begin{array}{ll}
 & (\textbf{case}\ (c\ e_1\ldots e_n)\ \textbf{of}\ p'_1 : e'_1\ |\ldots|\ p'_k : e'_k) \in edg^{s,n}(x,y,z),\ x \leq s,\ y \leq f,\ z \leq n \\
\Rightarrow & e_i \in edg^{s,n}(s-1,f,z-1),\ \forall\ i \in \{1\ldots n\} \\
\text{and} & e'_i \in edg^{s,n}(x-z,y,z),\ \forall\ i \in \{1\ldots k\} \\
\Rightarrow & (e'_i[e_1/v_1,\ldots,e_n/v_n]) \in edg^{s,n}(x-z,y,z)
\end{array}
$$

(since $v_1\ldots v_n \notin fv$)

$$
\begin{array}{ll}
\Rightarrow & (e'_i[e_1/v_1,\ldots,e_n/v_n]) \in edg^{s,n}(s,f,n)
\end{array}
$$

(since $x \leq s,\ y \leq f,\ z \leq n$)

Case for Rule 7:

$\mathcal{T}[\![\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] = f'\ v'_1 \ldots v'_k$

where

$f'\ v'_1 \ldots v'_k \quad = \quad \mathcal{T}[\![\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$

where f is defined by $f\ v_1 \ldots v_n = e$

and $v'_1 \ldots v'_k$ are the free variables in $(\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k)$

$(\textbf{case}\ (f\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in edg^{s,n}(x,y,z),\ x \leq s,\ y \leq f,\ z \leq n$

$\Rightarrow \quad e'_i \in edg^{s,n}(x-z, y, z),\ \forall\ i \in \{1 \ldots k\}$

and $\quad (f\ e_1 \ldots e_n) \in edg^{s,n}(x', y', z'),\ x' \leq s,\ y' \leq f,\ z' < z$

and $\quad e \in edg^{s,n}(s, 0, 1)$

$$\Rightarrow \quad e_i \begin{array}{ll} \in v, & \text{if } y' = f \\ \in edg^{s,n}(0,0,z'-1), & \text{if } 0 \leq y' < f \text{ and } e_i \text{ is a transient structure} \\ \in edg^{s,n}(x'-1,y',z'), & \text{if } 0 \leq y' < f \text{ and } e_i \text{ is not a transient structure} \end{array}$$

$$\Rightarrow \quad (e[e_1/v_1, \ldots, e_n/v_n]) \begin{array}{ll} \in edg^{s,n}(s,0,1), & \text{if } y' = f \\ \in edg^{s,n}(s,y'+1,z'), & \text{if } 0 \leq y' < f \end{array}$$

(since only transient structures can be the selectors in **case** expressions)

$\Rightarrow \quad (e[e_1/v_1, \ldots, e_n/v_n]) \in edg^{s,n}(s, f, z-1)$

$\Rightarrow \quad (\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in edg^{s,n}(x,y,z)$

$\Rightarrow \quad (\textbf{case}\ (e[e_1/v_1, \ldots, e_n/v_n])\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in edg^{s,n}(s,f,n)$

(since $x \leq s,\ y \leq f,\ z \leq n$)

Case for Rule 8:

$$\begin{array}{l} \mathcal{T}[\![\textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_n : e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\ = \quad \mathcal{T}[\![\textbf{case}\ e_0\ \textbf{of} \\ \qquad\qquad p_1 \ : \ \textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k \\ \qquad\qquad \vdots \\ \qquad\qquad p_n \ : \ \textbf{case}\ e_n\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \end{array}$$

$$(\textbf{case}\ (\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1\ | \ldots |\ p_n : e_n)\ \textbf{of}\ p'_1 : e'_1\ | \ldots |\ p'_k : e'_k) \in edg^{s,n}(x,y,z),\ x \leq s,\ y \leq f,\ z \leq n$$

$$\begin{array}{ll}
\Rightarrow & e_i \in edg^{s,n}(s-z+1, f, z-1),\ \forall\ i \in \{1 \ldots n\} \\
\text{and} & e'_i \in edg^{s,n}(x-z, y, z),\ \forall\ i \in \{1 \ldots k\} \\
\text{and} & e_0 \in edg^{s,n}(0,0,z-1) \\
\Rightarrow & (\textbf{case}\ e_0\ \textbf{of} \\
& \quad p_1 \ :\ \textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1\ | \ldots |\ p'_k : e'_k \\
& \quad \vdots \\
& \quad p_n \ :\ \textbf{case}\ e_n\ \textbf{of}\ p'_1 : e'_1\ | \ldots |\ p'_k : e'_k) \in edg^{s,n}(x,y,z) \\
\Rightarrow & (\textbf{case}\ e_0\ \textbf{of} \\
& \quad p_1 \ :\ \textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1\ | \ldots |\ p'_k : e'_k \\
& \quad \vdots \\
& \quad p_n \ :\ \textbf{case}\ e_n\ \textbf{of}\ p'_1 : e'_1\ | \ldots |\ p'_k : e'_k) \in edg^{s,n}(s,f,n) \\
& (\text{since } x \leq s,\ y \leq f,\ z \leq n)
\end{array}$$

□

C.2.2 Proof of Lemma 5.2.13

Prove: $\forall\ e \in edg^{s,n}(x,y,z),\ x \geq 0,\ s > 0,\ y \leq 0,\ f \geq 0,\ z > 0$:

$\mathcal{S}[\![e]\!] \leq x + (s \times y) + (s \times (f+1) \times (z-1))$

(Lemma 5.3.9 is a corollary of this)

The proof is by induction on the variable z.

Base Cases: $z = 1$

The proof of the base cases is by induction on the variable y.

Base Cases: $y = 0$

The proof of the base cases is by induction on the variable x.

Base Cases: $x = 0$

Case 1: $edg^{s,n}(x,y,z) ::= k$ if $x \geq 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned} \mathcal{S}[\![k]\!] &= 0 \\ &\leq x + (s \times y) + (s \times (f+1) \times (z-1)) \\ &\quad \text{(since } x = 0,\ y = 0,\ z = 1,\ s > 0 \text{ and } f \geq 0) \end{aligned}$$

Case 2: $edg^{s,n}(x,y,z) ::= v$ if $x \geq 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned} \mathcal{S}[\![v]\!] &= 0 \\ &\leq x + (s \times y) + (s \times (f+1) \times (z-1)) \\ &\quad \text{(since } x = 0,\ y = 0,\ z = 1,\ s > 0 \text{ and } f \geq 0) \end{aligned}$$

Inductive Cases: $x > 0$

Case 1: $edg^{s,n}(x,y,z) ::= c\ edg_1^{s,n}(x-1,y,z) \ldots edg_n^{s,n}(x-1,y,z)$

if $x > 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned}
\mathcal{S}[\![c\ e_1 \ldots e_n]\!] &= 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
&\leq 1 + (x-1) + (s \times y) + (s \times (f+1) \times (z-1)), \\
&\qquad \text{if } (c\ e_1 \ldots e_n) \in edg^{s,n}(x,y,z) \\
&\qquad \text{(by inductive hypothesis for } x\text{)} \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1))
\end{aligned}$$

Case 2: $edg^{s,n}(x,y,z) ::= f\ e_1 \ldots e_n$ if $x > 0$, $0 \leq y < f$ and $z > 0$
where f is defined by $f\ v_1 \ldots v_n = e$ and $e \in edg^{s,n}(s,0,1)$
and $e_i \in edg^{s,n}(0,0,z)$, if e_i is a transient structure
$\in edg^{s,n}(x-1,y,z)$, otherwise

$$\begin{aligned}
\mathcal{S}[\![f\ e_1 \ldots e_n]\!] &= 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
&\leq 1 + (x-1) + (s \times (y-1)) + (s \times f \times (z-1))), \\
&\qquad \text{if } (f\ e_1 \ldots e_n) \in edg^{s,n}(x,y,z) \\
&\qquad \text{(by inductive hypothesis for } x\text{)} \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1))
\end{aligned}$$

Case 3: $edg^{s,n}(x,y,z) ::= f\ v_1 \ldots v_n$ if $x > 0$, $y = f$ and $z > 0$
where f is defined by $f\ v'_1 \ldots v'_n = e$ and $e \in edg^{s,n}(s,0,1)$

$$\begin{aligned}
\mathcal{S}[\![f\ v_1 \ldots v_n]\!] &= 1 + max(\mathcal{S}[\![v_1]\!], \ldots, \mathcal{S}[\![v_n]\!]) \\
&= 1 \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1)) \\
&\qquad \text{(since } x > 0,\ y = 0,\ z = 1,\ s > 0 \text{ and } f \geq 0\text{)}
\end{aligned}$$

Case 4: $edg^{s,n}(x,y,z) ::=$
case $edg_0^{s,n}(0,0,z)$ **of** $p_1 : edg_1^{s,n}(x-z,y,z) \mid \ldots \mid p_k : edg_k^{s,n}(x-z,y,z)$
if $x > 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned}
&\mathcal{S}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!] \\
&\quad = \quad 1 + max(\mathcal{S}[\![e_0]\!], \ldots, \mathcal{S}[\![e_k]\!]) \\
&\quad \leq \quad 1 + (x - z) + (s \times y) + (s \times (f + 1) \times (z - 1))), \\
&\qquad \text{if } \textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k \in edg^{s,n}(x, y, z) \\
&\qquad \text{(by inductive hypothesis for } x\text{)} \\
&\quad \leq \quad x + (s \times y) + (s \times (f + 1) \times (z - 1)) \\
&\text{(since } z = 1\text{)}
\end{aligned}$$

Case 5: $edg^{s,n}(x, y, z) ::= edg^{s,n}(x - 1, y, z)$ if $x > 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned}
\mathcal{S}[\![e]\!] &\leq (x - 1) + (s \times y) + (s \times (f + 1) \times (z - 1)),\ \forall\ e \in edg^{s,n}(x - 1, y, z) \\
&\quad \text{(by inductive hypothesis for } x\text{)} \\
&\leq x + (s \times y) + (s \times (f + 1) \times (z - 1))
\end{aligned}$$

Inductive Cases: $y > 0$

The proof of the inductive cases is by induction on the variable x.

Base Cases: $x = 0$

Case 1: $edg^{s,n}(x, y, z) ::= k$ if $x \geq 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned}
\mathcal{S}[\![k]\!] &= 0 \\
&\leq x + (s \times y) + (s \times (f + 1) \times (z - 1)) \\
&\quad \text{(since } x = 0,\ y > 0,\ z = 1,\ s > 0 \text{ and } f \geq 0\text{)}
\end{aligned}$$

Case 2: $edg^{s,n}(x, y, z) ::= v$ if $x \geq 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned}
\mathcal{S}[\![v]\!] &= 0 \\
&\leq x + (s \times y) + (s \times (f + 1) \times (z - 1)) \\
&\quad \text{(since } x = 0,\ y > 0,\ z = 1,\ s > 0 \text{ and } f \geq 0\text{)}
\end{aligned}$$

Inductive Cases: $x > 0$

Case 1: $edg^{s,n}(x,y,z) ::= c\ edg_1^{s,n}(x-1,y,z) \ldots edg_n^{s,n}(x-1,y,z)$
if $x > 0$, $y \geq 0$ and $z > 0$

$$
\begin{array}{rcl}
\mathcal{S}[\![c\ e_1 \ldots e_n]\!] & = & 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
& \leq & 1 + (x-1) + (s \times y) + (s \times (f+1) \times (z-1)), \\
& & \text{if } (c\ e_1 \ldots e_n) \in edg^{s,n}(x,y,z) \\
& & \text{(by inductive hypothesis for } x) \\
& \leq & x + (s \times y) + (s \times (f+1) \times (z-1))
\end{array}
$$

Case 2: $edg^{s,n}(x,y,z) ::= f\ e_1 \ldots e_n$ if $x > 0$, $0 \leq y < f$ and $z > 0$
where f is defined by $f\ v_1 \ldots v_n = e$ and $e \in edg^{s,n}(s,0,1)$

$$
\begin{array}{lll}
\text{and } e_i & \in edg^{s,n}(0,0,z), & \text{if } e_i \text{ is a transient structure} \\
& \in edg^{s,n}(x-1,y,z), & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}[\![f\ e_1 \ldots e_n]\!] & = & 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
& \leq & 1 + (x-1) + (s \times y) + (s \times (f+1) \times (z-1))), \\
& & \text{if } (f\ e_1 \ldots e_n) \in edg^{s,n}(x,y,z) \\
& & \text{(by inductive hypothesis for } x) \\
& \leq & x + (s \times y) + (s \times (f+1) \times (z-1))
\end{array}
$$

Case 3: $edg^{s,n}(x,y,z) ::= f\ v_1 \ldots v_n$ if $x > 0$, $y = f$ and $z > 0$
where f is defined by $f\ v'_1 \ldots v'_n = e$ and $e \in edg^{s,n}(s,0,1)$

$$
\begin{array}{rcl}
\mathcal{S}[\![f\ v_1 \ldots v_n]\!] & = & 1 + max(\mathcal{S}[\![v_1]\!], \ldots, \mathcal{S}[\![v_n]\!]) \\
& = & 1 \\
& \leq & x + (s \times y)) + (s \times (f+1) \times (z-1)) \\
& & \text{(since } x > 0,\ y > 0,\ z = 1,\ s > 0 \text{ and } f \geq 0)
\end{array}
$$

Case 4: $edg^{s,n}(x, y, z)$ ::=

case $edg_0^{s,n}(0, 0, z)$ **of** $p_1 : edg_1^{s,n}(x-z, y, z) \mid \ldots \mid p_k : edg_k^{s,n}(x-z, y, z)$

if $x > 0$, $y \geq 0$ and $z > 0$

$$
\begin{aligned}
&\mathcal{S}[\![\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k]\!] \\
&= 1 + max(\mathcal{S}[\![e_0]\!], \ldots, \mathcal{S}[\![e_k]\!]) \\
&\leq 1 + (x - z) + (s \times y) + (s \times (f + 1) \times (z - 1))), \\
&\qquad \text{if } \textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots \mid p_k : e_k \in edg^{s,n}(x, y, z) \\
&\quad \text{(by inductive hypothesis for } x) \\
&\leq x + (s \times y) + (s \times (f + 1) \times (z - 1)) \\
&\quad \text{(since } x > 0,\ y > 0,\ z = 1,\ s > 0)
\end{aligned}
$$

Case 5: $edg^{s,n}(x, y, z)$ ::= $edg^{s,n}(x - 1, y, z)$ if $x > 0$, $y \geq 0$ and $z > 0$

$$
\begin{aligned}
\mathcal{S}[\![e]\!] &\leq (x - 1) + (s \times y) + (s \times (f + 1) \times (z - 1)), \ \forall\ e \in edg^{s,n}(x - 1, y, z) \\
&\quad \text{(by inductive hypothesis for } x) \\
&\leq x + (s \times y) + (s \times (f + 1) \times (z - 1))
\end{aligned}
$$

Case 6: $edg^{s,n}(x, y, z)$ ::= $edg^{s,n}(s, y - 1, z)$ if $x \geq 0$, $y > 0$ and $z > 0$

$$
\begin{aligned}
\mathcal{S}[\![e]\!] &\leq s + (s \times (y - 1)) + (s \times (f + 1) \times (z - 1)), \ \forall\ e \in edg^{s,n}(s, y - 1, z) \\
&\quad \text{(by inductive hypothesis for } y) \\
&\leq (s \times y) + (s \times (f + 1) \times (z - 1)) \\
&\leq x + (s \times y) + (s \times (f + 1) \times (z - 1)) \\
&\quad \text{(since } x > 0)
\end{aligned}
$$

Inductive Cases: $z > 1$

The proof of the inductive cases is by induction on the variable y.

Base Cases: $y = 0$

The proof of the base cases is by induction on the variable x.

Base Cases: $x = 0$

Case 1: $edg^{s,n}(x, y, z)$ ::= k if $x \geq 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned} \mathcal{S}[\![k]\!] &= 0 \\ &\leq x + (s \times y) + (s \times (f+1) \times (z-1)) \\ &\quad \text{(since } x = 0,\ y = 0,\ z > 1,\ s > 0 \text{ and } f \geq 0) \end{aligned}$$

Case 2: $edg^{s,n}(x, y, z)$ ::= v if $x \geq 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned} \mathcal{S}[\![v]\!] &= 0 \\ &\leq x + (s \times y) + (s \times (f+1) \times (z-1)) \\ &\quad \text{(since } x = 0,\ y = 0,\ z > 1,\ s > 0 \text{ and } f \geq 0) \end{aligned}$$

Inductive Cases: $x > 0$

Case 1: $edg^{s,n}(x, y, z)$ ::= $c\ edg_1^{s,n}(x-1, y, z) \ldots edg_n^{s,n}(x-1, y, z)$
if $x > 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned} \mathcal{S}[\![c\ e_1 \ldots e_n]\!] &= 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\ &\leq 1 + (x-1) + (s \times y) + (s \times (f+1) \times (z-1)), \\ &\qquad \text{if } (c\ e_1 \ldots e_n) \in edg^{s,n}(x, y, z) \\ &\quad \text{(by inductive hypothesis for } x) \\ &\leq x + (s \times y) + (s \times (f+1) \times (z-1)) \end{aligned}$$

Case 2: $edg^{s,n}(x, y, z)$ ::= $f\ e_1 \ldots e_n$ if $x > 0$, $0 \leq y < f$ and $z > 0$
where f is defined by $f\ v_1 \ldots v_n = e$ and $e \in edg^{s,n}(s, 0, 1)$

$$\begin{aligned} \text{and } e_i &\in edg^{s,n}(0, 0, z), && \text{if } e_i \text{ is a transient structure} \\ &\in edg^{s,n}(x-1, y, z), && \text{otherwise} \end{aligned}$$

$$
\begin{aligned}
\mathcal{S}[\![f\ e_1 \ldots e_n]\!] &= 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
&\leq 1 + max((\mathrm{s} + (s \times f) + (s \times (f+1) \times (z-2))), \\
&\quad ((x-1) + (s \times y) + (s \times (f+1) \times (z-1)))), \\
&\qquad \text{if } (f\ e_1 \ldots e_n) \in edg^{s,n}(x,y,z) \\
&\quad \text{(by inductive hypotheses for } x \text{ and } z) \\
&\leq 1 + (x-1) + (s \times y) + (s \times (f+1) \times (z-1)) \\
&\quad \text{(since } x > 0,\ y = 0) \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1))
\end{aligned}
$$

Case 3: $edg^{s,n}(x,y,z) ::= f\ v_1 \ldots v_n$ if $x > 0$, $y = f$ and $z > 0$
where f is defined by $f\ v'_1 \ldots v'_n = e$ and $e \in edg^{s,n}(s,0,1)$

$$
\begin{aligned}
\mathcal{S}[\![f\ v_1 \ldots v_n]\!] &= 1 + max(\mathcal{S}[\![v_1]\!], \ldots, \mathcal{S}[\![v_n]\!]) \\
&= 1 \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1)) \\
&\quad \text{(since } x > 0,\ y = 0,\ z > 1,\ s > 0 \text{ and } f \geq 0)
\end{aligned}
$$

Case 4: $edg^{s,n}(x,y,z) ::=$
case $edg_0^{s,n}(0,0,z)$ **of** $p_1 : edg_1^{s,n}(x-z,y,z) \mid \ldots \mid p_k : edg_k^{s,n}(x-z,y,z)$
if $x > 0$, $y \geq 0$ and $z > 0$

$\mathcal{S}[\![$**case** e_0 **of** $p_1 : e_1 \mid \ldots \mid p_k : e_k]\!]$

$$
\begin{aligned}
&= 1 + max(\mathcal{S}[\![e_0]\!], \ldots, \mathcal{S}[\![e_k]\!]) \\
&\leq 1 + max((s + (s \times f) + (s \times (f+1) \times (z-2))), \\
&\quad ((x-z) + (s \times y) + (s \times (f+1) \times (z-1)))),
\end{aligned}
$$

if **case** e_0 **of** $p_1 : e_1 \mid \ldots \mid p_k : e_k \in edg^{s,n}(x,y,z)$

$$
\begin{aligned}
&\quad \text{(by inductive hypotheses for } x \text{ and } z) \\
&\leq 1 + (x-z) + (s \times y) + (s \times (f+1) \times (z-1)) \\
&\quad \text{(since } x > 0,\ y = 0,\ s > 0) \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1))
\end{aligned}
$$

(since $z > 1$)

Case 5: $edg^{s,n}(x,y,z)$ ::= $edg^{s,n}(x-1,y,z)$ if $x > 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned}
\mathcal{S}[\![e]\!] &\leq (x-1) + (s \times y) + (s \times (f+1) \times (z-1)),\ \forall\, e \in edg^{s,n}(x-1,y,z) \\
&\quad \text{(by inductive hypothesis for } x\text{)} \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1))
\end{aligned}$$

Inductive Cases: $y > 0$

The proof of the inductive cases is by induction on the variable x.

Base Cases: $x = 0$

Case 1: $edg^{s,n}(x,y,z)$ ::= k if $x \geq 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned}
\mathcal{S}[\![k]\!] &= 0 \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1)) \\
&\quad \text{(since } x = 0,\ y > 0,\ z > 1,\ s > 0 \text{ and } f \geq 0\text{)}
\end{aligned}$$

Case 2: $edg^{s,n}(x,y,z)$ ::= v if $x \geq 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned}
\mathcal{S}[\![v]\!] &= 0 \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1)) \\
&\quad \text{(since } x = 0,\ y > 0,\ z > 1,\ s > 0 \text{ and } f \geq 0\text{)}
\end{aligned}$$

Inductive Cases: $x > 0$

Case 1: $edg^{s,n}(x,y,z)$ ::= $c\ edg_1^{s,n}(x-1,y,z) \ldots edg_n^{s,n}(x-1,y,z)$ if $x > 0$, $y \geq 0$ and $z > 0$

$$\begin{aligned}
\mathcal{S}[\![c\ e_1 \ldots e_n]\!] &= 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
&\leq 1 + (x-1) + (s \times y) + (s \times (f+1) \times (z-1)), \\
&\qquad \text{if } (c\ e_1 \ldots e_n) \in edg^{s,n}(x,y,z) \\
&\quad \text{(by inductive hypothesis for } x\text{)} \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1))
\end{aligned}$$

Case 2: $edg^{s,n}(x,y,z)$ ::= $f\ e_1 \ldots e_n$ if $x > 0$, $0 \leq y < f$ and $z > 0$

where f is defined by $f\ v_1 \ldots v_n = e$ and $e \in edg^{s,n}(s,0,1)$

and $e_i \in edg^{s,n}(0,0,z)$, if e_i is a transient structure

$\in edg^{s,n}(x-1,y,z)$, otherwise

$$\begin{aligned}
\mathcal{S}[\![f\ e_1 \ldots e_n]\!] &= 1 + max(\mathcal{S}[\![e_1]\!], \ldots, \mathcal{S}[\![e_n]\!]) \\
&\leq 1 + max((\mathrm{s} + (s \times f) + (s \times (f+1) \times (z-2))), \\
&\quad ((x-1) + (s \times y) + (s \times (f+1) \times (z-1)))), \\
&\quad \text{if } (f\ e_1 \ldots e_n) \in edg^{s,n}(x,y,z) \\
&\quad \text{(by inductive hypotheses for } x \text{ and } z\text{)} \\
&\leq 1 + (x-1) + (s \times y) + (s \times (f+1) \times (z-1)) \\
&\quad \text{(since } x > 0,\ y > 0\text{)} \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1))
\end{aligned}$$

Case 3: $edg^{s,n}(x,y,z)$ ::= $f\ v_1 \ldots v_n$ if $x > 0$, $y = f$ and $z > 0$

where f is defined by $f\ v'_1 \ldots v'_n = e$ and $e \in edg^{s,n}(s,0,1)$

$$\begin{aligned}
\mathcal{S}[\![f\ v_1 \ldots v_n]\!] &= 1 + max(\mathcal{S}[\![v_1]\!], \ldots, \mathcal{S}[\![v_n]\!]) \\
&= 1 \\
&\leq x + (s \times y) + (s \times (f+1) \times (z-1)) \\
&\quad \text{(since } x > 0,\ y > 0,\ z > 1,\ s > 0 \text{ and } f \geq 0\text{)}
\end{aligned}$$

Case 4: $edg^{s,n}(x,y,z)$ ::=

case $edg_0^{s,n}(0,0,z)$ **of** $p_1 : edg_1^{s,n}(x-z,y,z) \mid \ldots \mid p_k : edg_k^{s,n}(x-z,y,z)$

if $x > 0$, $y \geq 0$ and $z > 0$

$$
\begin{array}{ll}
\mathcal{S}[\![\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k]\!] & \\
\quad = & 1 + max(\mathcal{S}[\![e_0]\!], \ldots, \mathcal{S}[\![e_k]\!]) \\
\quad \leq & 1 + max((s + (s \times f) + (s \times (f+1) \times (z-2))), \\
 & ((x-z) + (s \times y) + (s \times (f+1) \times (z-1)))), \\
 & \text{if } \textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 \mid \ldots \mid p_k : e_k \in edg^{s,n}(x,y,z) \\
 & \text{(by inductive hypotheses for } x \text{ and } z\text{)} \\
\quad \leq & 1 + (x-z) + (s \times y) + (s \times (f+1) \times (z-1)) \\
 & \text{(since } x > 0,\ y > 0,\ s > 0\text{)} \\
\quad \leq & x + (s \times y) + (s \times (f+1) \times (z-1))
\end{array}
$$

(since $z > 1$)

Case 5: $edg^{s,n}(x,y,z) ::= edg^{s,n}(x-1,y,z)$ if $x > 0$, $y \geq 0$ and $z > 0$

$$
\begin{array}{lll}
\mathcal{S}[\![e]\!] & \leq & (x-1) + (s \times y) + (s \times (f+1) \times (z-1)),\ \forall\ e \in edg^{s,n}(x-1,y,z) \\
 & & \text{(by inductive hypothesis for } x\text{)} \\
 & \leq & x + (s \times y) + (s \times (f+1) \times (z-1))
\end{array}
$$

Case 6: $edg^{s,n}(x,y,z) ::= edg^{s,n}(s,y-1,z)$ if $x \geq 0$, $y > 0$ and $z > 0$

$$
\begin{array}{lll}
\mathcal{S}[\![e]\!] & \leq & s + (s \times (y-1)) + (s \times (f+1) \times (z-1)),\ \forall\ e \in edg^{s,n}(s,y-1,z) \\
 & & \text{(by inductive hypothesis for } y\text{)} \\
 & \leq & (s \times y) + (s \times (f+1) \times (z-1)) \\
 & \leq & x + (s \times y) + (s \times (f+1) \times (z-1)) \\
 & & \text{(since } x > 0\text{)}
\end{array}
$$

Case 7: $edg^{s,n}(x,y,z) ::= efg^{s,n}(s,f,z-1)$ if $x \geq 0$, $y \geq 0$ and $z > 1$

$$
\begin{array}{lll}
\mathcal{S}[\![e]\!] & \leq & s + (s \times f) + (s \times (f+1) \times (z-2)),\ \forall\ e \in efg^{s,n}(s,f,z-1) \\
 & & \text{(by inductive hypothesis for } z\text{, since} \\
 & & e \in efg^{s,n}(s,f,z-1) \Rightarrow e \in edg^{s,n}(s,f,z-1)) \\
 & \leq & (s \times (f+1) \times (z-1)) \\
 & \leq & x + (s \times y) + (s \times (f+1) \times (z-1)) \\
 & & \text{(since } x > 0,\ y > 0,\ s > 0\text{)}
\end{array}
$$

□

C.3 Proof of Generalised Deforestation Theorem

C.3.1 Proof of Lemma 5.3.3

In order to prove Lemma 5.3.3, the proof of Lemma 5.1.3 must be extended to include the three new transformation rules. The following inductive cases must be added to the recursion induction proof of Lemma 5.1.3.

Case for Rule 9:

$$\mathcal{T}[\![b\ e_1 \ldots e_n]\!] = b\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]$$

$$\begin{array}{rcl}
\mathcal{E}[\![b\ e_1 \ldots e_n]\!]\ \rho\ \phi & = & \mathcal{B}[\![b]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi)\ \ldots\ (\mathcal{E}[\![e_n]\!]\ \rho\ \phi) \\
& = & \mathcal{B}[\![b]\!]\ (\mathcal{E}[\![\mathcal{T}[\![e_1]\!]]\!]\ \rho\ \phi)\ \ldots\ (\mathcal{E}[\![\mathcal{T}[\![e_n]\!]]\!]\ \rho\ \phi) \\
& & \text{(by inductive hypothesis)} \\
& = & \mathcal{E}[\![b\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]]\!]\ \rho\ \phi \\
\Rightarrow\ \mathcal{E}[\![\mathcal{T}[\![b\ e_1 \ldots e_n]\!]]\!]\ \rho\ \phi & = & \mathcal{E}[\![b\ e_1 \ldots e_n]\!]\ \rho\ \phi
\end{array}$$

Case for Rule 10:

$$\begin{array}{l}
\mathcal{T}[\![\mathbf{case}\ (b\ e_1 \ldots e_n)\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\
\quad = \ \mathbf{case}\ (b\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!])\ \mathbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!]
\end{array}$$

$$\begin{array}{l}
\mathcal{E}[\![\mathbf{case}\ (b\ e_1 \ldots e_n)\ \mathbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho\ \phi \\
\quad = \ \mathcal{E}[\![e'_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi \\
\qquad \mathbf{where} \\
\qquad x \ = \ \mathcal{E}[\![b\ e_1 \ldots e_n]\!]\ \rho\ \phi \\
\qquad p'_i \ = \ c\ v_1 \ldots v_n\ \mathbf{and}\ match(x, c) \\
\quad = \ \mathcal{E}[\![e'_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi \\
\qquad \mathbf{where} \\
\qquad x \ = \ \mathcal{B}[\![b]\!]\ (\mathcal{E}[\![e_1]\!]\ \rho\ \phi) \ldots (\mathcal{E}[\![e_n]\!]\ \rho\ \phi) \\
\qquad p'_i \ = \ c\ v_1 \ldots v_n\ \mathbf{and}\ match(x, c)
\end{array}$$

$$= \mathcal{E}[\![\mathcal{T}[\![e'_i]\!]]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi$$

where

$$x = \mathcal{B}[\![b]\!]\ (\mathcal{E}[\![\mathcal{T}[\![e_1]\!]]\!]\ \rho\ \phi) \ldots (\mathcal{E}[\![\mathcal{T}[\![e_n]\!]]\!]\ \rho\ \phi)$$

$$p'_i = c\ v_1 \ldots v_n \textbf{ and } match(x, c)$$

(by inductive hypothesis)

$$= \mathcal{E}[\![\textbf{case}\ (b\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!])\ \textbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!]]\!]\ \rho\ \phi$$

$$\Rightarrow \mathcal{E}[\![\mathcal{T}[\![\textbf{case}\ (b\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!]\ \rho\ \phi$$

$$= \mathcal{E}[\![\textbf{case}\ (b\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho\ \phi$$

Case for Rule 11:

$$\mathcal{T}[\![\textbf{let}\ v = e_0\ \textbf{in}\ e_1]\!] = \textbf{let}\ v = \mathcal{T}[\![e_0]\!]\ \textbf{in}\ \mathcal{T}[\![e_1]\!]$$

$$\mathcal{E}[\![\textbf{let}\ v = e_0\ \textbf{in}\ e_1]\!]\ \rho\ \phi = \mathcal{E}[\![e_1]\!]\ \rho[(\mathcal{E}[\![e_0]\!]\ \rho\ \phi)/v]\ \phi$$

$$= \mathcal{E}[\![\mathcal{T}[\![e_1]\!]]\!]\ \rho[(\mathcal{E}[\![\mathcal{T}[\![e_0]\!]]\!]\ \rho\ \phi)/v]\ \phi$$

(by inductive hypothesis)

$$= \mathcal{E}[\![\textbf{let}\ v = \mathcal{T}[\![e_0]\!]\ \textbf{in}\ \mathcal{T}[\![e_1]\!]]\!]\ \rho\ \phi$$

$$\Rightarrow \mathcal{E}[\![\mathcal{T}[\![\textbf{let}\ v = e_0\ \textbf{in}\ e_1]\!]]\!]\ \rho\ \phi = \mathcal{E}[\![\textbf{let}\ v = e_0\ \textbf{in}\ e_1]\!]\ \rho\ \phi$$

Case for Rule 12:

$$\mathcal{T}[\![\textbf{case}\ (\textbf{let}\ v = e_0\ \textbf{in}\ e_1)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$

$$= \textbf{let}\ v = \mathcal{T}[\![e_0]\!]\ \textbf{in}\ \mathcal{T}[\![\textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$

$$\mathcal{E}[\![\textbf{case}\ (\textbf{let}\ v = e_0\ \textbf{in}\ e_1)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho\ \phi$$

$$= \mathcal{E}[\![e'_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi$$

where

$$x = \mathcal{E}[\![\textbf{let}\ v = e_0\ \textbf{in}\ e_1]\!]\ \rho\ \phi$$

$$p'_i = c\ v_1 \ldots v_n \text{ and } match(x, c)$$

$$= \mathcal{E}[\![e'_i]\!]\ \rho[x \downarrow 1/v_1, \ldots, x \downarrow n/v_n]\ \phi$$

where

$$x = \mathcal{E}[\![e_1]\!]\ \rho[(\mathcal{E}[\![e_0]\!]\ \rho\ \phi)/v]\ \phi$$

$$p'_i = c\ v_1 \ldots v_n \text{ and } match(x, c)$$

$$= \mathcal{E}[\![\textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]\ \rho[(\mathcal{E}[\![e_0]\!]\ \rho\ \phi)/v]\ \phi$$

(since v does not occur free in $e'_1 \ldots e'_k$)

$$= \mathcal{E}[\![\mathcal{T}[\![\textbf{case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \; \rho[(\mathcal{E}[\![\mathcal{T}[\![e_0]\!]]\!] \; \rho \; \phi)/v] \; \phi$$

(by inductive hypothesis)

$$= \mathcal{E}[\![\textbf{let } v = \mathcal{T}[\![e_0]\!] \textbf{ in } \mathcal{T}[\![\textbf{case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \; \rho \; \phi$$

$$\Rightarrow \mathcal{E}[\![\mathcal{T}[\![\textbf{case } (\textbf{let } v = e_0 \textbf{ in } e_1) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \; \rho \; \phi$$

$$= \mathcal{E}[\![\textbf{case } (\textbf{let } v = e_0 \textbf{ in } e_1) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \; \rho \; \phi$$

□

C.3.2 Proof of Lemma 5.3.4

In order to prove Lemma 5.3.4, the proof of Lemma 5.1.5 must be extended to include the three new transformation rules. The following inductive cases must be added to the recursion induction proof of Lemma 5.1.5.

Case for Rule 9:

$$\mathcal{T}[\![b \; e_1 \ldots e_n]\!] = b \; \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]$$

$$\mathcal{R}[\![\mathcal{T}[\![e_i]\!]]\!] \leq \mathcal{R}[\![e_i]\!], \forall \, i \in \{1 \ldots n\}$$

(by inductive hypothesis)

$$\Rightarrow \mathcal{R}[\![b \; \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]]\!] \leq \mathcal{R}[\![b \; e_1 \ldots e_n]\!]$$

$$\Rightarrow \mathcal{R}[\![\mathcal{T}[\![b \; e_1 \ldots e_n]\!]]\!] \leq \mathcal{R}[\![b \; e_1 \ldots e_n]\!]$$

Case for Rule 10:

$$\mathcal{T}[\![\textbf{case } (b \; e_1 \ldots e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$
$$= \textbf{case } (b \; \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]) \textbf{ of } p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!]$$

$$\mathcal{R}[\![\mathcal{T}[\![e_i]\!]]\!] \leq \mathcal{R}[\![e_i]\!], \forall \, i \in \{1 \ldots n\}$$

(by inductive hypothesis)

$$\mathcal{R}[\![\mathcal{T}[\![e'_i]\!]]\!] \leq \mathcal{R}[\![e'_i]\!], \forall \, i \in \{1 \ldots k\}$$

(by inductive hypothesis)

$$\Rightarrow \mathcal{R}[\![\textbf{case } (b \; \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]) \textbf{ of } p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!]]\!]$$
$$\leq \mathcal{R}[\![\textbf{case } (b \; e_1 \ldots e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$

$$\Rightarrow \mathcal{R}[\![\mathcal{T}[\![\textbf{case } (b \; e_1 \ldots e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!]$$
$$\leq \mathcal{R}[\![\textbf{case } (b \; e_1 \ldots e_n) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$

Case for Rule 11:

$$\mathcal{T}[\![\textbf{let } v = e_0 \textbf{ in } e_1]\!] = \textbf{let } v = \mathcal{T}[\![e_0]\!] \textbf{ in } \mathcal{T}[\![e_1]\!]$$

$$\begin{array}{lll} & \mathcal{R}[\![\mathcal{T}[\![e_i]\!]]\!] & \leq \mathcal{R}[\![e_i]\!], \forall\, i \in \{0,1\} \\ & & \text{(by inductive hypothesis)} \\ \Rightarrow & \mathcal{R}[\![\textbf{let } v = \mathcal{T}[\![e_0]\!] \textbf{ in } \mathcal{T}[\![e_1]\!]]\!] & \\ & & \leq \mathcal{R}[\![\textbf{let } v = e_0 \textbf{ in } e_1]\!] \\ \Rightarrow & \mathcal{R}[\![\mathcal{T}[\![\textbf{let } v = e_0 \textbf{ in } e_1]\!]]\!] & \\ & & \leq \mathcal{R}[\![\textbf{let } v = e_0 \textbf{ in } e_1]\!] \end{array}$$

Case for Rule 12:

$$\begin{array}{l} \mathcal{T}[\![\textbf{case } (\textbf{let } v = e_0 \textbf{ in } e_1) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\ \quad = \textbf{let } v = \mathcal{T}[\![e_0]\!] \textbf{ in } \mathcal{T}[\![\textbf{case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \end{array}$$

$$\begin{array}{ll} & \mathcal{R}[\![\textbf{let } v = e_0 \textbf{ in case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\ & \quad \leq \mathcal{R}[\![\textbf{case } (\textbf{let } v = e_0 \textbf{ in } e_1) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\ & \mathcal{R}[\![\mathcal{T}[\![\textbf{let } v = e_0 \textbf{ in case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \\ & \quad \leq \mathcal{R}[\![\textbf{let } v = e_0 \textbf{ in case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\ & \quad \text{(by inductive hypothesis)} \\ \Rightarrow & \mathcal{R}[\![\mathcal{T}[\![\textbf{let } v = e_0 \textbf{ in case } e_1 \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \\ & \quad \leq \mathcal{R}[\![\textbf{case } (\textbf{let } v = e_0 \textbf{ in } e_1) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \\ \Rightarrow & \mathcal{R}[\![\mathcal{T}[\![\textbf{case } (\textbf{let } v = e_0 \textbf{ in } e_1) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]]\!] \\ & \quad \leq \mathcal{R}[\![\textbf{case } (\textbf{let } v = e_0 \textbf{ in } e_1) \textbf{ of } p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!] \end{array}$$

□

C.3.3 Proof of Lemma 5.3.8

In order to prove Lemma 5.3.8, the proof of Lemma 5.2.12 must be extended to include the three new transformation rules. The following inductive cases must be added to the proof of Lemma 5.2.12.

Case for Rule 9:

$$\mathcal{T}[\![b\ e_1 \ldots e_n]\!] \quad = \quad b\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!]$$

$(b\ e_1 \ldots e_n) \in edg^{s,n}(x, y, z),\ x \leq s,\ y \leq f,\ z \leq n$

$\Rightarrow\quad e_i \in edg^{s,n}(x-1, y, z),\ \forall\ i \in \{1 \ldots n\}$

$\Rightarrow\quad e_i \in edg^{s,n}(s, f, n),\ \forall\ i \in \{1 \ldots n\}$

(since $x \leq s,\ y \leq f,\ z \leq n$)

Case for Rule 10:

$$\mathcal{T}[\![\textbf{case}\ (b\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$
$$= \quad \textbf{case}\ (b\ \mathcal{T}[\![e_1]\!] \ldots \mathcal{T}[\![e_n]\!])\ \textbf{of}\ p'_1 : \mathcal{T}[\![e'_1]\!] \mid \ldots \mid p'_k : \mathcal{T}[\![e'_k]\!]$$

$(\textbf{case}\ (b\ e_1 \ldots e_n)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in edg^{s,n}(x, y, z),\ x \leq s,\ y \leq f,\ z \leq n$

$\Rightarrow\quad e_i \in edg^{s,n}(s-1, y, z-1),\ \forall\ i \in \{1 \ldots n\}$

and $\quad e'_i \in edg^{s,n}(x-z, y, z),\ \forall\ i \in \{1 \ldots k\}$

$\Rightarrow\quad e_i \in edg^{s,n}(s, f, n),\ \forall\ i \in \{1 \ldots n\}$

and $\quad e'_i \in edg^{s,n}(s, f, n),\ \forall\ i \in \{1 \ldots k\}$

(since $x \leq s,\ y \leq f,\ z \leq n$)

Case for Rule 11:

$$\mathcal{T}[\![\textbf{let}\ v = e_0\ \textbf{in}\ e_1]\!] \quad = \quad \textbf{let}\ v = \mathcal{T}[\![e_0]\!]\ \textbf{in}\ \mathcal{T}[\![e_1]\!]$$

$(\textbf{let}\ v = e_0\ \textbf{in}\ e_1) \in edg^{s,n}(x, y, z),\ x \leq s,\ y \leq f,\ z \leq n$

$\Rightarrow\quad e_i \in edg^{s,n}(x-1, y, z),\ \forall\ i \in \{0,1\}$

$\Rightarrow\quad e_i \in edg^{s,n}(s, f, n),\ \forall\ i \in \{0,1\}$

(since $x \leq s,\ y \leq f,\ z \leq n$)

Case for Rule 12:

$$\mathcal{T}[\![\textbf{case}\ (\textbf{let}\ v = e_0\ \textbf{in}\ e_1)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$
$$= \quad \textbf{let}\ v = \mathcal{T}[\![e_0]\!]\ \textbf{in}\ \mathcal{T}[\![\textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k]\!]$$

$$(\textbf{case}\ (\textbf{let}\ v = e_0\ \textbf{in}\ e_1)\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in edg^{s,n}(x,y,z),$$
$$x \leq s,\ y \leq f,\ z \leq n$$

$\Rightarrow$ $e_i \in edg^{s,n}(s-1, f, z-1),\ \forall\ i \in \{0,1\}$

and $e'_i \in edg^{s,n}(x-z, y, z),\ \forall\ i \in \{1 \ldots k\}$

$\Rightarrow$ $(\textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in edg^{s,n}(x,y,z)$

$\Rightarrow$ $e_0 \in edg^{s,n}(s,f,n)$

and $(\textbf{case}\ e_1\ \textbf{of}\ p'_1 : e'_1 \mid \ldots \mid p'_k : e'_k) \in edg^{s,n}(s,f,n)$

(since $x \leq s,\ y \leq f,\ z \leq n$)

□